Forward: an Array Iteration in a Stream Language
or
iterating in space with an iteration in time

Grégoire Bussone[1] Jean-Louis Colaço[2] Baptiste Pauget[2]
Marc Pouzet[1]

2. ANSYS, Toulouse

1. ENS/PSL/INRIA, Paris

Marc.Pouzet@ens.fr

Workshop SYNCHRON
Bamberg University, Germany
Nov. 2024

The context: certified real-time software

Users: software dev. submitted to independent certification authorities.

E.g., avionic, railways.

Correctness and determinacy must be demonstrated w.r.t a reference spec.

Automation tools like compilers are concerned by certification.

"Efficient" means "fast enough": does the system reacts on time?

"Dynamic" (e.g., run-time exception) means "too late".

Consequences for PL/compiler developers: no out-of-the-bounds, no access to un-initialized variables, no type errors; the code must executes with statically known bounded memory and time, no GC... and even no malloc.

with generated code that is simple enough to compute the WCET.

The beautiful idea of Lustre [Caspi et al., 1987]

A PL to write a mathematical executable specification of a control system.

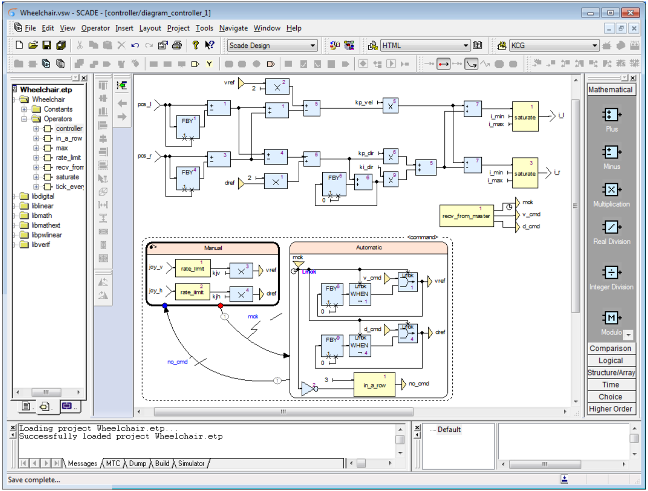
An ideal synchronous and deterministic model.

A signal is a stream; a system is a length-preserving stream function.

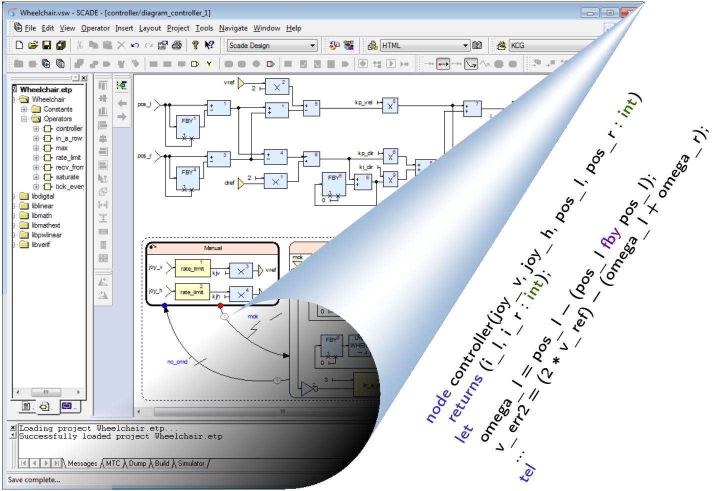
simulated, tested, formally verified (as much as possible), and compiled to C code.

The language Scade 6 [Colaco et al., 2017] is built on Lustre principles and is state-of-the-art for certified software.

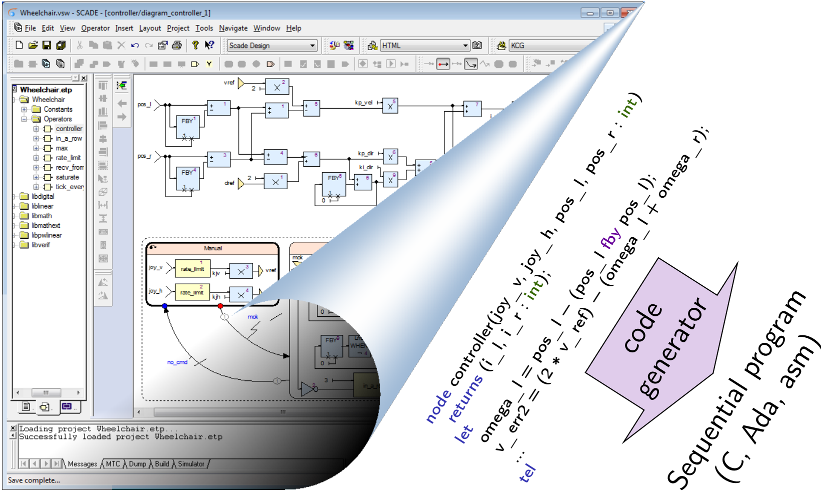
The language Scade 6



The language Scade 6



The language Scade 6



Some typical stream functions [1]

```
(* backward Euleur integration *)
(* x(0) = x0(0) /\ x(n) = x(n-1) + h(n) * xprime(n) *)
let node backward_euler(h, x0, xprime) = x where rec
  x = x0 -> pre(x) +. h *. xprime

(* a PI controller *)
let node pi(h, p, i, error) = u where rec
  u = p *. error + backward_euler(h, 0.0, i *. error)

(* a system with two modes *)
let node controller(h, p, i, auto, toggle, input, measure) = u
where rec
  automaton
  | Direct -> do u = input until auto then Auto
  | Auto -> do u = pid(h, p, i, input -. measure)
            until not auto then Direct
  end
```

[1] Written in Zélus: zelus.di.ens.fr

A question

Address applications that mix complex control and computer intensive array-based operations.

E.g., Kalman filtering, optimization-based control algorithms, ML algorithms, etc.

Importing imperative libraries is possible but the certification is lost.

How to conservatively extend the language?

Write a purely functional specification with generated code that is efficient enough.

Study the consequences for the compiler to pass certification requirements.

Array operations in Lustre

Many solutions have been done to mix Lustre and arrays.

- Lustre arrays (V4/V5) [Rocheteau and Halbwachs, 1991].

  Compilation by maximal extension; good for circuit synthesis. Inadequate for software.

- HDL like Jazz [Vuillemin, 1993, Bourdoncle and Merz, 1997], HDL embedded in Haskell (e.g., Lava [Bjesse et al., 1998]) went further in term of expressiveness.

  Good for circuit synthesis; inadequate for software code.

- Array iterators (e.g., map/fold of FP) [Morel, 2007].

  Easy to generate sequential code; but too much copies in the code.

  Some primitives are lacking; what is the minimal set?

- Array iterators of Scade 6 [Colaco et al., 2017].

  Same advantages but same weaknesses.

  In practice, the generated code has too much copies

Map/fold iterators

E.g., FP languages (Haskell, DSLs in Haskell), DSLs like DEX and Futhark.

Examples below are written in Zélus [2].

```
let sum(x, y) = map (fun (x, y) -> x +. y) (combine x y))

let dot(x, y) =
  fold (fun acc (x, y) -> acc +. x *. y) 0.0 (combine x y)

let node delay(x, y) =
  map (fun (x, y) -> x fby y) (combine x y)
```

Scade 6: map, fold, mapfold, transpose, reverse, concat, window, etc.

Array iterators can be applied to combinational and stateful functions.

[2] Examples in this presentation can be tested with the ZRun interpreter: https://github.com/marcpouzet/zrun/tree/work

An other need: iterate several synchronous steps but observe only one.

that is, iterate in time versus iterate in space.

Time refinement

E.g., implement a 32 bit adder from a one bit adder.

An iterative algorithm (e.g., square root), an optimization control technique.

Hide several synchronous steps as if there were only one.

A stream function whose input clock is slower than the internal clock (or internal is faster).

This feature is not possible in Lustre nor Scade; it is rejected by the clock calculus.

It was expressible in Signal [Benveniste et al., 1991] and Lucid Synchrone, Version 1.1 (1998) [Caspi and Pouzet, 1999].

but never really exploited.

Can we interpret a (finite) stream as an array or, conversely, an array as a (finite) stream?

The `forward` iteration construct

- Different from the "parallel" iteration (e.g., `map`, `fold`).
- The forward iterates one single stream function.
- It interprets an input array as a finite stream.
- The body can be any stateful (sequential) stream function.

The construct is studied by B. Pauget in his PhD. thesis [Pauget, 2023].

An experiment with the Zrun interpreter [3].

[3] https://github.com/marcpouzet/zrun/tree/work

Example: combinational functions

```
(* [x] and [y] are streams; [z] is the point-wise sum *)
(* [z(i) = x(i) + y(i)] for all i in Nat *)
let sum(x, y) = z where z = x +. y

(* [x] and [y] are two streams of arrays of length [n] *)
(* Returns the stream of arrays [z] st: *)
(* forall i in Nat. z(i)[j] = x(i)[j] +. y(i)[j] *)
let sum(x, y) returns (z)
  z = forward ([xj] in x, [yj] in y) returns ([zj])
        zj = xj +. yj
      done
```

``[zj]'' is the array made of the *n* sums.

Here, there is no difference with a parallel version because the body is combinational.

```
let sum(x,y) = map2 (+.) x y
```

Forward of a sequential function

Now, use a sequential function inside the body, e.g., to perform an accumulatiion.

```
(* [forall i in Nat. o(i) = o(i-1) + x(i) * y(i)] *)
let node acc(x, y) returns (o)
  o = (0 fby o) +. x * y

(* [x], [y] are two streams of arrays of length [n] *)
(* returns a stream [o] st: *)
let scalar(x, y) returns (o)
  o = forward ([xj] in x, [yj] in y) returns (o)
        do o = (0 fby o) +. xj *. yj done

let scalar(x, y) returns (o)
  o = forward ([xj] in x, [yj] in y) returns (o)
        do o = acc(xj, yj)
      done
```

`o` is the last computed value of the sequence.

The resulting function is considered to be combinational.

Initialization of an accumulation

An alternative definition:

```
(* [last o] is the last value of the (stream) variable [o] *)
(* i.e., (last o)(i) = o(i-1). *)
(* declaration [o init 0] means (last o)(0) = 0 *)

let node acc(x, y) returns (o init 0)
  o = last o +. x *. y

let scalar(x, y) returns (o)
   o = forward ([xj] in x, [yj] in y) returns (o init 0)
         do o = last o +. xj *. yj
      done
```

Forward and Reset

1. "forward" mimics a "for" loop, interpreting an array as a finite stream .
2. Reset or resume the internal state at the end of the loop iteration?
3. Two choice are possible; both are useful.
4. By default (when nothing is said), the body is reset.
5. The consequence is that `forward` ... is considered a combinatorial expression.
6. In term of an implementation, the state can be stack allocated.

Accumulation without reset

```
(* Euler integration *)
let node euler_backward(h, x0, xprime) returns (x)
  do x = x0 -> pre(x) +. h *. xprime done
```

Imagine the speed comes by chunks of length n.

```
(* [xprime] is a stream of arrays of size n *)
let node euler_pack(h, x0, xprime) returns (x)
  forward resume ([xi] in xprime) returns (x)
    do x = euler_backward(h, x0, xi) done
```

This code models a time acceleration with several consecutive steps performed in a raw.

The resulting function is sequential, not combinatorial (keyword `node`).

By default, the state is reset, following ReactiveML [Mandel et al., 2015].

Automata and Forward

An example by JL. Colaco.

```
(* monotonic returns true when a is monotonic
(increasing, constant, decreasing) *)
let node monotonic (a) returns (o default true)
    automaton
    | S0 ->
        do unless true continue Constant
    | Constant ->
        do unless (a > last a) continue Increasing
           unless (a < last a) continue Decreasing
    | Increasing ->
        do unless (a < last a) continue NonMonotonic
    | Decreasing ->
        do unless (a > last a) continue NonMonotonic
    | NonMonotonic ->
        do o = false done
    end
```

```
let main(a) returns (o)
    o = forward ai in a do monotonic ai done

let main() =
  let a = [| 1; 2; 3; 4; 4; 4; 3; 2; 1; 5; 6 |] in
  main(a)
```

An iterative algorithm

Compute the solution of $f(x) = 0$ by the Newton method.

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

E.g., compute the square root of an input. Do at most max step.

```
let newton(max)(eps)(f)(f')(v) returns (x)
  forward(max) returns (x init v) do
    x = last x -. f(last x) / f'(last x)
  while (x -. last x) >= eps done

(* application to the square root *)
let main(max)(eps)(v) =
  let f x = x *. x -. v in
  let f' x = 2.0 *. x in
  newton(max)(eps)(f)(f')(v)
```

The Boyer & Moore majority voting algorithm [4]

```
(* Boyer and Moore Majority Algorithm *)
(* Find in linear time (two passes) one vote that appears *)
(* strictly more than half of the time; if there is some *)

(* Initialize an element m and a counter c with c = 0
``For each element x of the input sequence:
If c = 0, then assign m = x and c = 1
else if m = x, then assign c = c + 1
else assign c = c - 1
Return m'' *)

(* [x] is a stream of votes *)
let node vote(x) returns (m init 0)
  local c init 0 in
  if last c = 0 then do m = x and c = 1 done
  else if m = x then
         c = last c + 1
       else
         c = last c - 1
```

[4] https://en.wikipedia.org/wiki/Boyer-Moore_majority_vote_algorithm

```
let count_votes (x, a) returns (n)
  n = forward ([ai] in a) returns (s init 0) do
           s = if ai = x then last s + 1 else last s
      done

let main(nb_votes, votes) returns (candidate, has_majority)
  do
    candidate = forward(vi in votes) returns (c)
                  do c = vote(vi) done
  and
    has_majority =
      2 * (count_votes(candidate, votes)) > nb_votes
  done

let nb_votes = 10

let node main () returns (o)
  let votes = [|3;2;3;1;3;2;3;1;3;3|] in
  do o = main(nb_votes, votes) done
```

Related works

- Time refinement [Mikác and Caspi, 2009]; Reactive domains [Mandel et al., 2015] for ReactiveML; Integer clocks [Guatto, 2018].
- Reification: integrate to the language the finite iteration of the transition function produced by the compiler.
- Oversampling in Signal [Benveniste et al., 1991, Amagbegnon et al., 1995]: the clock of the input is slower than the the internal clock.
- Lucid Synchrone [Pouzet, 2006] was also allowing oversampling functions to be defined.
- But their compiler did not generate a simple nested "for" loop.
- Functional circuit languages, e.g., Lava [Bjesse et al., 1998] do have a construct to unfold a sequential function.
- Simulink do have an operator to "iterate" a sequential function.

What the Forward Iteration adds to a Stream Language

- Interpret an array as a (finite) stream.
- Stream (sequential) operations, that iterate on time, can be used to iterate on space.
- A program construct to express temporal refinement.
- It makes a connection between two old PL born at the same time: Lustre and SISAL [Feo et al., 1990][5].
- A side-effect (unexpected) of this work: its gives a purely functional, stream semantics, for the Simulink iteration block.

[5] Stream and Iteration in a Single Assignment Language

What would be a stream semantics of this construct?

Back to KPN [Kahn, 1974]

Express the "forward" loop as a higher-order stream function. A Kahnian interpretation of Lustre primitives [Caspi, 1992, Colaço and Pouzet, 2003].

V^n is the set of sequences of length n; $V^\star = \cup_{n=0}^{\infty} V^n$. the prefix order. $(V^\infty, \epsilon, \leq)$, is a CPO. ϵ models a stream that stuck.
Function below are continuous for the prefix order.

$$
\begin{array}{lcl}
\mathit{const}(v) & = & v.\mathit{const}(v) \\
\mathit{extend}(f.\mathit{fs})(v.s) & = & (f\ v).\mathit{extend}(\mathit{fs})(s) \\
(v_1.s_1)\ \mathit{fby}\ s_2 & = & v_1.s_2 \\
\mathit{when}(v.s, 1.c) & = & v.\mathit{when}(s, c) \\
\mathit{when}(v.s, 0.c) & = & \mathit{when}(s, c) \\
\mathit{merge}(1.c, v.s_1, s_2) & = & v.\mathit{merge}(c, s_1, s_2) \\
\mathit{merge}(0.c, s_1, v.s_2) & = & v.\mathit{merge}(c, s_1, s_2) \\
 & = & \epsilon \text{ otherwise}
\end{array}
$$

$$\begin{array}{rcl} next\,(x.s) & = & s \\ restart\ f\ x & = & (f\ x)\ fby\ (restart\ f\ (next\,(x))) \\ & = & \epsilon \text{ otherwise} \end{array}$$

$next\,(s)$ is the "tail" on (infinite) lists.

If $f : stream(T) \rightarrow stream(T')$, then $restart\ f\ .$ is a combinational function, even if f is not.

Indeed, a function g is combinational if $g(x\ fby\ y) = (g\ x)\ fby\ (g\ y)$

Warning: ϵ is NOT the empty list, it models a stream that stuck, that is:

$$\epsilon\ \texttt{fby}\ x = \epsilon$$

Taking $\epsilon\ \texttt{fby}\ x = x$ would make $.\ \texttt{fby}\ .$ not monotone.

A Stream Definition for the Forward

$$
\begin{array}{rcl}
\mathit{forward\_resume}\langle n\rangle\ f\ x & = & \mathit{let}\ s = \mathit{soa}\langle n\rangle(x)\ \mathit{in} \\
 & & \mathit{let}\ s' = f\ s\ \mathit{in} \\
 & & \mathit{let}\ a = \mathit{aos}\langle n\rangle(s')\ \mathit{in} \\
 & & o \\
\mathit{forward\_restart}\langle n\rangle\ f\ x & = & \mathit{restart} \\
 & & \quad (\mathit{forward\_resume}\langle n\rangle\ f) \\
 & & \quad (x)
\end{array}
$$

where:

$$
\begin{array}{rcl}
\mathit{soa}\langle n\rangle([x_0; ...; x_{n-1}].s) & = & x_0....x_{n-1}.\mathit{soa}\langle n\rangle(s) \\
\mathit{aos}\langle n\rangle(x_0.....x_{n-1}.s) & = & [x_0; ...; x_{n-1}].\mathit{aos}\langle n\rangle(s) \\
 & = & \epsilon \text{ otherwise}
\end{array}
$$

Again: wait to have n input values before outputting a data.

$\mathit{forward\_resume}\langle . \rangle$. and $\mathit{forward\_restart}\langle . \rangle$. are continuous because they compose continous functions.

Mux/Demux

x	$=$	$[x_0; x_1; x_2]_0$			$[x_0; x_1; x_2]_1$			$\ldots$
s	$=$	$x_{0,0}$	$x_{1,0}$	$x_{2,0}$	$x_{0,1}$	$x_{1,1}$	$x_{2,1}$	
s'	$=$	$y_{0,0}$	$y_{1,0}$	$y_{2,0}$	$y_{0,1}$	$y_{1,1}$	$y_{2,1}$	
a	$=$			$[y_0; y_1; y_2]_0$			$[y_0; y_1; y_2]_1$	$\ldots$

That's all: the denotational, Kahnian, semantics of the forward is expressed by composing basic existing primitives.

This is not expressible in Lustre: the internal clock to compute `s` and `s'` is faster than that of `x` and `a`. It is forbidden by the clock calculus.

Can we express it using clocked streams?

```
(* Lucid Synchrone V1.1 (1998) *)
open Array
open Hold

let period n = ok where
  rec cpt = 0 fby (cpt + 1) mod n
  and ok = (cpt = n-1)

let stream_of_array n x =
  let rec i = 0 fby (i + 1) mod n in
  Array.get (hold (true fby (period n)) x) i

let array_of_stream n x =
  let rec i = 0 fby (i + 1) mod n in
  (Array.update (Array.make n x) i x) when (period n)

let forward (static n) f x =
  let s = stream_of_array(n)(x) in
  let ys = f s in
  array_of_stream(n)(ys)
```

Type and clock signatures

The definition before were valid functions in Lucid Synchrone V1.1 (1998)!
In particular, the compiler infers the following type and clock signatures:

```
plume.local[1] lucyc -i forward.ls

node period :  int -> bool
node period :: 'a -> 'a

node stream_of_array :  int -> 'a array -> 'a
node stream_of_array ::
  (n_1:'a) -> 'a on true fby period n_1 -> 'a

node array_of_stream :  int -> 'a -> 'a array
node array_of_stream :: (n_1:'a) -> 'a -> 'a on period n_1

node forward : int -> ('a -> 'b) -> 'a array -> 'b array
node forward ::
  (n_1:'a) -> ('a -> 'a)
  -> 'a on true fby period n_1 -> 'a on period n_1
```

Yet, the clock calculus of V1.1 was not expressive enough. If forces the input and output of the forward to have the same clock.

Use the clock calculus of V2.0 (2002) instead. Clock “static” is given to a constant, i.e., it can be used at any clock.

```
node forward ::
  (n_1: static) -> ('a -> 'b)
  -> 'a on true fby period n_1 -> 'b on period n_1
```

composes a bit better.

Yet, is this embedding useful (more than a curiosity)? I don't know.

An Operational State-based Semantics

We have also extended the constructive synchronous semantics presented in [Colaco et al., 2023].

A forward loop simply iterates the internal state function.

The Zrun interpreter [6] implements it.

[6] `https://github.com/marcpouzet/zrun/tree/work`

An Operational Interpretation

Suppose $f : stream(T) \to stream(T')$ is a stream function that is implemented as a pair made of:

- an initial state: $so : S$;
- a transition function $step : S \to T \to T' \times S$.

The classical map function is simply:

$map\, n\, f\, x = y$ where for all $j \in [0 \ldots n-1]$, $y[j] = f(x[j])$.

That is, for all $j \in [0 \ldots n-1]$ and for all $i \in \mathbb{N}$:

$$(s[j]_0 = so) \wedge (y[j]_i, s[j]_{i+1} = step\,(s[j]_i)\,(x[j]_i))$$

Equivalently, since n is constant, it can also be defined as a function which apply to a stream of arrays $x : \mathit{stream}((T')^n)$ and returns a stream of arrays $y : \mathit{stream}((T')^n)$ such that:

$$\forall j \in [0 \ldots n-1].\ (y_i[j])_{i \in \mathbb{N}} = f\,((x_i[j])_{i \in \mathbb{N}})$$

Operationally, for all $j \in [0 \ldots n-1]$ and for all $i \in \mathbb{N}$:

$$(s_0[j] = so) \wedge (y_i[j], s_{i+1}[j] = \mathit{step}\,(s_i[j])\,(x_i[j]))$$

The Forward

If $f : \mathit{stream}(T) \to \mathit{stream}(T')$, we can convert an array $x : (T)^n$ of size n into a stream, pass it to f and convert the result into an array $y : (T')^n$:

$$\begin{array}{l} \forall j \in [0..n-1]. \\ \quad (s_0 = \mathit{so}) \wedge (y[j], s_{j+1} = \mathit{step}\,(s_j)\,(x[j])) \end{array}$$

More generally, if $x : \mathit{stream}((T)^n)$:

$$\begin{array}{l} \forall i \in \mathbb{N},\ \exists s. \forall j \in [0..n-1]. \\ \quad (s_0 = \mathit{so}) \wedge (y_i[j], s_{j+1} = \mathit{step}\,(s_j)\,(x_i[j])) \end{array}$$

This is the *forward restart* form.

The Forward

$$\begin{aligned}&\exists s.\forall i \in \mathbb{N}.\forall j \in [0..n-1].\\&\quad (s_0 = so) \wedge (y_i[j], s_{n\times i+j+1} = \mathit{step}\,(s_{n\times i+j})\,(x_i[j]))\end{aligned}$$

the *forward restart* with a single output that is accumulated becomes:

$$\begin{aligned}&\forall i \in \mathbb{N},\ \exists s, o.\forall j \in [0..n-1].\\&\quad (s_0 = so) \wedge (o_j, s_{j+1} = \mathit{step}\,(s_j)\,(x_i[j]))\\&\quad \wedge (y_i = o_{n-1})\end{aligned}$$

and for the *forward resume*:

$$\begin{aligned}&\exists s, o.\forall i \in \mathbb{N}.\forall j \in [0..n-1].\\&\quad (s_0 = so) \wedge (o_{n\times i+j}, s_{n\times i+j+1} = \mathit{step}\,(s_{n\times i+j})\,(x_i[j]))\\&\quad \wedge (y_i = o_{n\times i+(n-1)})\end{aligned}$$

Conclusion

- Add an novel construct to a Lustre-like language to iterate a stream function on an array.
- A denotational, stream-based, semantics and an state-based one.
- For sequential code generation, generate a for loop. The state can be stack allocated when the forward is reset.
- This operation can be combined with rich projection functions (reverse, append, slices, transpose, etc.).
- Still, the generation of efficient code that minimizes copies is difficult.
- Read the (beautiful) PhD. thesis of Baptiste Pauget [Pauget, 2023].
- The forward construct will be part of the new Scade language.

References I

Amagbegnon, T., Besnard, L., and Guernic., P. L. (1995).
Implementation of the data-flow synchronous language signal.
In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM.

Benveniste, A., LeGuernic, P., and Jacquemot, C. (1991).
Synchronous programming with events and relations: the SIGNAL language and its semantics.
Science of Computer Programming, 16:103–149.

Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1998).
Lava: Hardware Design in Haskell.
In *International Conference on Functional Programming (ICFP)*. ACM.

Bourdoncle, F. and Merz, S. (1997).
Type checking higher-order polymorphic multi-methods.
In *Annual Symposium on Principles of Programming Languages*, Paris, France. SIGPLAN-SIGACT, ACM.

Caspi, P. (1992).
Clocks in dataflow languages.
Theoretical Computer Science, 94:125–140.

Caspi, P., Halbwachs, N., Pilaud, D., and Plaice, J. (1987).
Lustre: a declarative language for programming synchronous systems.
In *14th ACM Symposium on Principles of Programming Languages*. ACM.

Caspi, P. and Pouzet, M. (1999).
Lucid Synchrone, version 1.01. Tutorial and reference manual.
Laboratoire d'Informatique de Paris 6.

References II

Colaço, J.-L. and Pouzet, M. (2003).
Clocks as First Class Abstract Types.
In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA.

Colaco, J.-L., Mendler, M., Pauget, B., and Pouzet, M. (2023).
A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State machines.
In *International Conference on Embedded Software (EMSOFT'23)*, Hamburg, Germany. ACM.

Colaco, J.-L., Pagano, B., and Pouzet, M. (2017).
Scade 6: A Formal Language for Embedded Critical Software Development.
In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France.

Feo, J. T., Cann, D. C., and Oldehoeft, R. R. (1990).
A report on the Sisal language project.
Journal of Parallel and Distributed Computation, 10:349–366.

Guatto, A. (2018).
A generalized modality for recursion.
In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491.

Kahn, G. (1974).
The semantics of a simple language for parallel programming.
In *IFIP 74 Congress*. North Holland, Amsterdam.

References III

Mandel, L., Pasteur, C., and Pouzet, M. (2015).
Time Refinement in a Functional Synchronous Language.
Science of Computer Programming.
Available online 10 July 2015.

Mikác, J. and Caspi, P. (2009).
Flush: an example of development by refinements in scade/lustre.
Int. J. Softw. Tools Technol. Transf., 11(5):409–418.
see: https://inria.hal.science/inria-00466172.

Morel, L. (2007).
Array iterators in lustre: From a language extension to its exploitation in validation.
EURASIP Journal on Embedded Systems.

Pauget, B. (2023).
Memory Specification in a Data-flow Synchronous Language with Statically Sized Arrays.
PhD thesis, Université PSL - Ecole normale supérieure.

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

Rocheteau, F. and Halbwachs, N. (1991).
POLLUS: A LUSTRE based hardware design environment.
In *Algorithms and Parallel VLSI Architectures*, pages 335–346.

Vuillemin, J. (1993).
On Circuits and Numbers.
Technical report, Digital, Paris Research Laboratory.