# Clocks as First Class Abstract Types

*or*

*how to turn the clock calculus*
*into a (quasi) ML type system*


Jean-Louis Colaço [a]

Esterel Technologies

Marc Pouzet [b]

Université Paris 6 (LIP6)



Synchron 2003 [c]



[a] Jean-Louis.Colaco@esterel-technologies.com

[b] Marc.Pouzet@lip6.fr



[c] Part of this work have been presented at Emsoft 2003.

**The context.**

- synchronous data-flow languages
- need to compose processes on different rates or *clocks*
- slow and fast processes by means of sampled streams
- **but...** the parallel composition may need synchronisation at run-time involving possibly unbounded buffering
- they must be rejected when dealing with real-time computing
- this is the purpose of the **clock-calculus**

## What is a clock?

- an information about the behavior of a sampled stream
- it tells *when* the value of the stream is *available* on the channel
- statically reject non-synchronous programs (`x + x when half`)

| `x` | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|---|---|---|---|---|---|---|---|
| `half` | $t$ | $f$ | $t$ | $f$ | $t$ | $f$ | $t$ |
| `x when half` | $x_0$ | | $x_2$ | | $x_4$ | | $x_6$ |

## Goals.

- automatic clock inference (*mandatory in graphical tools*)
- fast, syntactic approach and modular (*real-size programs*)
- good diagnostic (*can I easilly understand why the program is not clocked?*)

## Approach.

- the clock is a (dependent) type
- what can we learn from this relationship?

**Define a Clock Calculus for two Compilers.**

**Lucid Synchrone** *(Univ. Paris 6)*

- synchronous data-flow + ML
- type and clock inference, polymorphism
- causality, init. analysis
- new features (e.g., merge, pattern-matching, data-types)

**The ReLuC Compiler** *(Esterel Technologies)*

- first-order
- prototype compiler for **Scade-Lustre**
- improved static analysis and code generation
- techniques closed to the ones of Lucid Synchrone
  (e.g. clocks, program analysis, code generation)
- new features (e.g., merge, array iterators)

**Data-flow programming.**

- constants lifted to streams
- pointwise application of primitives
- `pre` is the unitary delay (*it is an un-initialised register*)

| | | | | | |
|---|---|---|---|---|---|
| `1` | $1$ | $1$ | $1$ | $1$ | $\ldots$ |
| `x` | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $\ldots$ |
| `y` | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $\ldots$ |
| `x + y` | $x_0 + y_0$ | $x_1 + y_1$ | $x_2 + y_2$ | $x_3 + y_3$ | $\ldots$ |
| `x -> y` | $x_0$ | $y_1$ | $y_2$ | $y_3$ | $\ldots$ |
| `pre y` | $nil$ | $y_0$ | $y_1$ | $y_2$ | $\ldots$ |

**Programs are defined by set of equations between streams**

```
let sum x = s where
    rec s = x -> pre s + x
node sum :  int -> int
node sum ::  'a -> 'a
```

**They can be composed**

```
let sum2 = sum (sum x)
node sum2 :  int -> int
node sum2 ::  'a -> 'a
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `1` | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| `sum 1` | 1 | 2 | 3 | 4 | 5 | 6 | ... |
| `sum (sum 1)` | 1 | 3 | 6 | 10 | 15 | 21 | ... |

**These are length-preserving functions**

## Up and down-sampling streams.

- `when` is the *down*-sampling operator

| | | | | | |
|---|---|---|---|---|---|
| `c` | $f$ | $t$ | $f$ | $t$ | $\ldots$ |
| `x` | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $\ldots$ |
| `x when c` | | $x_1$ | | $x_3$ | $\ldots$ |

- `merge` combines two *complementary* sub-streams

| | | | | | |
|---|---|---|---|---|---|
| `c` | $f$ | $t$ | $f$ | $t$ | $\ldots$ |
| `y` | | $y_0$ | | $y_1$ | $\ldots$ |
| `z` | $z_0$ | | $z_1$ | | $\ldots$ |
| `merge c y z` | $z_0$ | $y_0$ | $z_1$ | $y_1$ | $\ldots$ |

**Computes the sum of x according to a condition clk.**

```
let sampled_sum clk x = sum (x when clk)
node sampled_sum :  clock -> int -> int
node sampled_sum ::  (_c0:'a) -> 'a -> 'a on _c0
```

**Sets the output of the sampled sum on the base clock.**

```
let cond_act_sum clk x = o where
    rec o = merge clk (sampled_sum clk x) po
    and po = (0 -> pre o) whenot clk
node cond_act_sum :  int -> clock -> int
node cond_act_sum ::  'a -> (_c0:'a) -> 'a
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `1` | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| `half` | $t$ | $f$ | $t$ | $f$ | $t$ | $f$ | ... |
| `sum half 1` | 1 | | 2 | | 3 | | ... |
| `cond_act_sum half 1` | 1 | 1 | 2 | 2 | 3 | 3 | ... |

## Rejected programs.

```
let unclocked x = x + (x when half)
> x + (x when half)
>      ^^^^^^^^^^^^
This expression has clock 'b on half,
but is used with clock 'b.
```

## Clocks.

- a clock is a (dependent) type since it may contain expressions
- the clock calculus can be expressed as a type system

## Questions.

- what kind of expression is allowed in clocks?
- which (decidable) equality between clocks do we choose ?
- what about unification?

**A synchronous kernel.**

Data-flow primitives + functions

$$
\begin{array}{lcl}
e & ::= & i \mid x \mid op(e,e) \mid \texttt{pre}\ e \mid e\ \texttt{->}\ e \\
 & & \mid e\ \texttt{when}\ e \mid \texttt{merge}\ e\ e\ e \\
 & & \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid \texttt{rec}\ x = e \\
 & & \mid \lambda x.e \\
 & & \mid e(e) \\
 & & \\
i & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid \ldots \\
op & ::= & + \mid \ldots
\end{array}
$$

We give it a synchronous semantics

- clocks may contain program expressions
- defined as a dependent type system (ICFP'96)

**Clocks (as dependent types).**

- embedded into the proof system **Coq** (LPAR'01)
- used in the **ReLuC** compiler of **Scade/Lustre** (2000)
- experimented on real size programs (50000 lines of code)
- **but...** the implementation is quite tricky
- **ReLuC** only implements verification, no clock inference
- restrictions to keep the implementation practible are not totally satisfactory (e.g, syntactic equality between clocks)
  - *the substitution principle is lost*
  - *the system is sensitive to the introduction/erasing of intermediate names*

**Questions.**

- can we identify a (sub-system only) reminicent to ML type systems?
- what do we loose? (is-it enough for every day programming?)

**A (quasi) ML type system.**

- avoid expressions in clocks, avoid substitution in clocks
- be unsensitive to the introduction of intermediate names
- use first-order unification (efficient, well known implementation techniques)
- can I simulate it in the Caml type system?

**The idea.**

- explicit introduction of a **clock name** through a special construction
- clock equivalence is based on name equality
- only those names can be used for sampling a signal
- this corresponds to the classical Laufer & Odersky extension of ML

**Examples.**

```
(* a sampler that counts modulo n *)
let sample n = ok where
    rec cpt = 0 -> if pre cpt = n - 1
                   then 0 else pre cpt + 1
    and ok = cpt = 0
```

```
(* defining a 1/60 clock *)
let clock sixty = sample 60

let sum x = o where
    rec o = x -> pre o + x

let sum_60 x = sum (x when sixty)
node sum_60 :  int -> int
node sum_60 ::  'a -> 'a on sixty
```

**Nesting.**

```
let hour_minute_second second =
    let minute = second when sixty in
    let hour = minute when sixty in
    hour,minute,second
```

```
node hour_minute_second :  'a -> 'a * 'a * 'a
node hour_minute_second ::
     'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

**Fast/Slow computation (Alain Girault).**

Suppose that `slow` has signature:

```
node slow ::  'a -> 'a
```

```
let filtre ck input =
    let rec
        out = slow input
        and fast = 0 -> pre fast + 1
        and res = merge ck ((0 when ck) -> pre out)
                           ((0 -> pre res) whenot ck)
    in res, out, fast
```

*`node filtre ::  (ck:'a) -> 'a on ck -> 'a * 'a on ck * 'a`*

`slow` is used at a slow rate (it receives the clock instance `'a on ck -> 'a on ck`)

**Playing with higher-order.**

A macro operator which “merge” two blocks.

```
let merge_blocks ck f g i1 i2 = o where
    o = merge ck (f (i1 when ck)) (g (i2 whenot ck))
```

```
node merge_blocks ::  (ck:'a) -> ('a on ck -> 'a on ck)
                              -> ('a on not ck -> 'a on not ck)
                              -> 'a
```

**Rmk:** such an operator can be hard coded inside a compiler where inputs of the two blocks are implicitely filtered.

**Examples (cont).**

Higher-order: the activation condition, a common data-flow primitive.

```
let condact clk f input init = u where
    rec u = merge clk (f (input when clk))
                      ((init fby u) whenot clk)
node condact :  clock -> ('a -> 'b) -> 'a -> 'b -> 'b
node condact ::  (clk:'a) -> ('a on clk -> 'a on clk)
                          -> 'a -> 'a -> 'a
```

**The symetric operator: the iterator.**

```
let iter clk init f input =
    (* read input when clk is true *)
    let i = merge clk input ((init fby i) whenot clk) in
    let rec o = f i po
    and po = merge clk (init when clk) ((init fby o) whenot clk) in
    (* emit output when clk is true *)
    o when clk
```

```
node iter ::
    (clk:'a) -> 'a -> ('a -> 'a -> 'a) -> 'a on clk -> 'a on clk
```

**Computing the power of a sequence.**

```
let power clk x = o where
   o = iter clk 1 (fun i acc -> i * acc) x
```

| clk | $t$ | $f$ | $f$ | $t$ | $f$ | $t$ | $f$ | $f$ | $f$ | $t$ | $t$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | $x_0$ | | | $x_1$ | | $x_2$ | | | | $x_3$ | $x_4$ | ... |
| k | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 | 4 | 1 | 1 | ... |
| y | 1 | | | $x_0^3$ | | $x_1^2$ | | | | $x_2^4$ | $x_3$ | ... |

## Unicity of clock names.

when a clock name is introduced, the name is considered to be unique no matter is the right-hand side

```
let clock c = true in
let v = 1 when c in
let clock c = true in
let w = 1 when c in
v + w
    ^^
This expression has clock 'a on ?_c0,
but is used with clock 'a on ?_c1.
```

This is a good programming discipline

## Language extension.

$$e \quad ::= \quad \texttt{let clock}\ x = e\ \texttt{in}\ e \mid \ldots$$

**The clock type language.**

$$
\begin{array}{lcl}
\sigma & ::= & \forall \beta_1, ..., \beta_n. \forall \alpha_1, ..., \alpha_m. \forall X_1, ..., X_k. cl \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \\
 & & \mid \beta \mid s \mid (c : s) \\
s & ::= & s \ \texttt{on} \ c \mid s \ \texttt{on not} \ c \mid \alpha \\
c & ::= & n \mid X
\end{array}
$$

$$
H \quad ::= \quad [x_1 : cl_1, ..., x_n : cl_n]
$$

$$
\textsf{judgment} \quad ::= \quad H \vdash e : cl
$$

**Initial conditions.**

$$\texttt{pre} : \forall\alpha.\alpha \to \alpha \qquad \texttt{when} : \forall\alpha.\forall X.\alpha \to (X : \alpha) \to \alpha \ \texttt{on}\ X$$

**The system.**

$$\frac{H \vdash e_1 : s \quad H \vdash e_2 : s}{H \vdash op(e_1, e_2) : s} \qquad \frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : cl_1 \to cl_2}$$

$$\frac{H \vdash e_1 : cl_1 \to cl_2 \quad H \vdash e_2 : cl_2}{H \vdash e_1\,(e_2) : cl_1}$$

This is standard ML typing

## Clock system (cont.)

$$\frac{H \vdash e_1 : s \quad H, x : (n : s) \vdash e_2 : cl_2 \quad n \notin names(H, cl_2)}{H \vdash \texttt{let clock } x = e_1 \texttt{ in } e_2 : cl_2}$$

## Conclusion.

- the `let clock` construction introduces a fresh name (a new label)
- name equivalence: $s_1 \texttt{ on } n \equiv s_2 \texttt{ on } n$ if $s_1 \equiv s_2$
- clock inference
- classical ML typing with existential data-types (Laufer & Odersky)
- standard theory, standard ML implementation (e.g, destructive unification)
- very efficient (as efficient as ML type inference)

**Scope restrictions.**

**We have lost something!**

```
let escape x =
    let clock c = (x = 0) in
    x when c
```

```
>....escape x =
> let clock c = (x = 0) in
> x when c
The clock of this expression depends on ?c_0 which escape its scope.
```

Old answer (e.g. Lustre, Lucid Synchrone v1.x):

> A clock can be either an input or computed locally. In the later case, it must be returned by the function.

This corresponds to *dependent pairs*: $\Sigma(ck : cl_1).cl_2$ is a pair made of a boolean stream `ck` with clock $cl_1$ and a value with clock $cl_2$. $cl_2$ may depend on $ck$.

A (very) restricted case of this can be achieved in the Laufer & Odersky system by introducing a particular type (e.g, $\texttt{box}(\alpha)$ for pairs made of a boolean stream $ck$ (with clock $\alpha$ and a value $v$ on clock $\alpha$ on $ck$.

**Related works.**

**Lustre.**

- **Lucid synchrone** and **Lustre** are based on the same semantics
- first-order
- clocks are similar; syntactic equality two
- clock verification instead of clock inference
- fix-point iteration techniques *vs* unification
- no polymorphism of constants: the folowing program is rejected

  ```
  (x when clk) + 1
  ```

- restriction on clock schemes: one clock per node, named the base clock of the node

  ```
  let f (x,y) = (x+1, y+2)
  ```

  *node f : ('a * 'b) -> ('a * 'b)* ← *here*

  *node f : ('a * 'a) -> ('a * 'a)* ← *in* **Lustre**

**Signal.**

- different semantics, asynchronous operators (e.g, `when`, `default`)
- first-order
- clock inference two
- far more expressive clock calculus
- over-sampling

The proposed calculus allows some restricted forms of oversampling

```
let stuttering x = o where
    rec o = merge half x (0 -> pre o) whenot half)
node stuttering ::  'a on half -> 'a
```

**Conclusion.**

- clocks as (quasi)-ML types
- clocks are simpler and easy to understand
- automatic inference
- fast, one pass analysis
- small (2000 lines in all)
- some *modest* forms of oversampling
- reduce scope restrictions
- data-structures (record types, sum types)