Lucid Synchrone
version 1.01

Tutorial and reference manual

Paul Caspi and Marc Pouzet

January 99

Contents

Foreword

This document describes Lucid Synchrone, an experimental synchronous data-flow language dedicated to the implementation of reactive systems. Lucid Synchrone combines the real-time features of synchronous data-flow languages and the functionality of ML languages. The name "Lucid Synchrone" is built from "Lucid" [1, 2] a data-flow language managing streams and from the French word "synchrone" (for "synchronous").

The main features of the language are:

- It is a strongly typed, higher-order functional language built above the host language Objective Caml [7] keeping its syntax and its run-time system.
- The language manages infinite lists — streams — as primitive values. These streams are used for representing input and output signals of reactive systems. These streams are combined with a set of synchronous data-flow operators in the spirit of Lustre [5].
- Programs are compiled into Objective Caml [7] code. A compilation mode is provided to ensure that compiled programs are "real-time" (the compiled code uses bounded memory and have a bounded response time whatever be the program inputs).
- A simple module system is provided for importing values from the host language Objective Caml or from other synchronous modules. A complete system is thus made of several modules, some of them programming the combinatorial part (i.e, the instant), some other programming the dynamical part (i.e, the time).

Availability

The current implementation is written in Objective Caml. The use of the language needs the installation of Objective Caml.

```
Lucid Synchrone, version 1.01:  http://spi.lip6.fr/lucid-synchrone
Objective Caml, version 2.01:   http://pauillac.inria.fr/caml
```

The language is experimental and evolves continuously. Please send comments or bug to `Marc.Pouzet@lip6.fr`.

Copyright notice

Part I

Lucid Synchrone

Chapter 1

An introduction to Lucid Synchrone

This section is a tutorial introduction to LUCID SYNCHRONE. A good familiarity with general programming languages is assumed. Some familiarity with (strict or lazy) ML languages and with existing synchronous data-flow languages would be helpful since the language incorporates features from both families. Some references are given at the end of this document.

For this tutorial, we suppose that programs are written in a file `tutorial.ls`.

1.1 First-order features

Point-wise operations

The language is a functional language, with a OBJECTIVE CAML syntax except that, as in LUSTRE, every ground type or any scalar value imported from the host language OBJECTIVE CAML is implicitly lifted to streams. Thus,

- `int` stands for the type of streams of integers,
- `1` stands for the constant stream of values 1,
- `+` adds point-wise its two input streams. It can be seen as an adder circuit, in the same way as `&` can be seen as an "and" gate.

`c`	t	f	t	...
`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`x+y`	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	...
`if c then x else y`	x_0	y_1	x_2	...

Delays

`fby` (read "followed by") is an initialized delay operator. `->` is the initialization operator. `pre` (for "previous") is the uninitialized delay: at the first instant, it takes any value with a valid type (denoted by *nil*).

`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`x fby y`	x_0	y_0	y_1	...
`pre x`	nil	x_0	x_1	...
`x -> y`	x_0	y_1	y_2	...

Global declarations

A program is made of a sequence of declarations of global values named *nodes*. `let` defines non recursive global values whereas `let rec` define recursive global values. For example, the sequence of naturals is defined by:

```
let rec nat = 0 -> pre nat + 1
```

`0`	0	0	0	0	0	0	...
`1`	1	1	1	1	1	1	...
`pre nat`	nil	0	1	2	3	4	...
`pre nat + 1`	nil	1	2	3	4	5	...
`0 -> pre nat + 1`	0	1	2	3	4	5	...

An edge detector can be defined in the following way:

```
let edge c = c & not (false fby c)
```

`c`	f	f	t	t	f	t	...
`false`	f	f	f	f	f	f	...
`false fby c`	f	f	f	t	t	f	...
`not (false fby c)`	t	t	t	f	f	t	...
`edge c`	f	f	t	f	f	t	...

`edge` is a global function from boolean streams to boolean streams. An integrator is defined like the following:

```
let sum x = s
    where rec s = x + (0 fby s)
```

`sum` is a global function returning a stream `s` defined recursively. Defined functions may be composed. For example:

```
let sum2 x = sum(sum(x))
```

When this text is fed into the compiler `lucyc` with option `-i`, we get the answer:

```
node nat :  int
node nat :: 'a
node edge :  bool -> bool
node edge :: 'a -> 'a
node sum :  int -> int
node sum :: 'a -> 'a
node sum2 :  int -> int
node sum2 :: 'a -> 'a
```

For each declaration, we get both the type and clock inferences. The clock `'a` for the global constant `nat` states that `nat` may be used at any clock. The clock `'a -> 'a` for the node `edge` states that the function is length preserving: it returns a value for every input. Clocks will be explained further.

When a stream is defined recursively in an equation `rec` s `=` e, the compiler checks that there is no causality loop in the definition (i.e, $\mathtt{s}_n$ is defined only in term of $\mathtt{s}_m$ with $m < n$. For example, if we type:

```
let sum x = s
    where rec s = x + s
```

the compiler emits the message:

```
File "tutorial.ls", line 2, characters 20-21:
>  where rec s = x + s
>                    ^
Causality loop.
```

In LUCID SYNCHRONE, causality is ensured by checking that recursions are always on the right of a delay. Moreover, the compiler checks that every delay operator is initialized. With the program:

```
let sum x = s
    where rec s = x + pre s
```

the compiler emits the message:

```
File "tutorial.ls", line 2, characters 24-25:
>  where rec s = x + pre s
>                        ^
This expression must be initialised.
```

Local definitions

Variables may be defined locally with a `let/in` or `let rec/in`. For example:

```
let rec fibo =
  let pfibo = 1 -> pre fibo in
  1 -> pfibo + pre pfibo
```

defines the *Fibonacci* sequence. Local definitions can be given in a similar way with `where` and `where rec`. Local definitions are just regular expression and can be arbitrarily nested.

Mutually recursive streams

Streams may be defined as a set of mutual recursive equations. Consider, for example, a mouse controller. It is parameterized by two input streams `click` and `top` and must emit `double` when it has received two `click` in less than four `tops`. Otherwise, it emits `single`.

```
(* counts the number of events from the last reset res *)
let counter(res,event) = count
where rec count = if res then 0 else x
      and x = 0 -> if event then pre count + 1 else pre count


let mouse (click,top) = (single,double) where
  rec counting = click -> if click & not pre counting then true
                          else if res & pre counting then false
                          else pre counting
  and count = counter(res,top & counting)
  and single = ((0 fby count) = 3) & top & not click
  and double = (false fby counting) & click
  and res = single or double
```

The compiler emits the types and clocks on the standard output:

```
node counter :  bool * bool -> int
node counter :: 'a * 'a -> 'a
node mouse :  bool * bool -> bool * bool
node mouse :: 'a * 'a -> 'a * 'a
```

Reseted application

Lucid Synchrone provides means of reseting the internal states of a function in a modular way: every stream restarts with its initial value. For example:

```
let sum_every_c dx c =
  reset sum dx every c
```

resets the internal states of `sum` every time the condition `c` is true. More precisely, the internal stream `0 fby s` of `sum` restarts with its first value `0` every time `c` is true.

```
node sum_every_c :: 'a -> 'a -> 'a
```

The reseted application has the general form of an application except that it receives a supplementary reset condition.

Sampling

`when` is a sampler which allows fast processes to communicate with slower ones by extracting sub-streams from streams according to a condition, *i.e.* a boolean stream.

`true`	t	t	t	t	...
`c`	f	t	f	t	...
`x`	x_0	x_1	x_2	x_3	...
`x when c`		x_1		x_3	...
`true on c`	f	t	f	t	...

The sampling operators introduce the notion of *clock*. The clock of a stream s is a boolean sequence giving the instants where s is defined. Among these clocks, the base clock stands for the constant stream `true`: a stream on the base clock is present at every instant. In the above example, the current value of `x when c` is present when `x` and `c` are present and `c` is true. Since `x` and `c` are on the base clock `true`, the clock of `x when c` is noted `true on c`.

Now the `sum` function can be used at a slower rate by sampling its input stream:

```
let mywhen x y = x when y

let sampled_sum x c = sum(x when c)

node mywhen :  'a -> bool -> 'a
node mywhen :: 'a -> (x__2:'a) -> 'a on x__2
node sampled_sum :  int -> bool -> int
node sampled_sum :: 'a -> (x__4 : 'a) -> 'a on x__4
```

`sampled_sum` has a function clock which follows its type structure. This clocks says that for any clock `a`, if the first argument of the function has clock `a` and the second argument named `x__4` has clock `a`, then the result is on clock `a on x__4` (every expression variable is renamed in the clock to avoid name conflicts).

As a final remark, note that in general, sampling an input has not the same effect as sampling the output (`(sum (x when c))` $\neq$ `((sum x) when c)`).

Of course, not every program is well-clocked. For example:

```
let wrong_clocked x =
  let half = true -> not (pre half) in
  x + (x when half)
```

adds the stream `x` to the sub-stream of `x` made by filtering one item over two. The system's answer is:

```
File "a.ls", line 3, characters 7-18:
>  x + (x when half)
>       ^^^^^^^^^^^
This expression has clock 'a on x__2,
but is used with clock 'a.
```

`x__2` denotes an expression variable containing the value of `half`.

Such a program is forbidden in LUCID SYNCHRONE because it would need an unbounded memory to be computed (some items of `x` are accumulated because `+` needs to have a value present on its two inputs).

`merge` conversely allows slow processes to communicate with faster ones by merging substreams into "larger" ones:

c	f	t	f	t	...
x		x_0		x_1	...
y	y_0		y_1		...
merge c x y	y_0	x_0	y_1	x_1	...

For instance, it allows us to define an "holding" function (the "current" operator of LUSTRE), which "holds" a signal between two successive samples (here `ydef` is a default value used before any sample has been taken):

```
let mymerge c x y = merge c x y
let hold ydef c x = y
    where rec y = merge c x ((ydef fby y) when (not c))

node mymerge :  bool -> 'a -> 'a -> 'a
node mymerge :: (x__1:'a) -> ('a on x__1) -> ('a on not x__1) -> 'a
node hold :  'a -> bool -> 'a -> 'a
node hold :: 'a -> (x__8 : 'a) -> ('a on x__8) -> 'a
```

It also allows a limited form of oversampling, *i.e.* functions whose output has a larger rate than the input:

```
let rec half = false fby (not h)

let over x = hold 1 half x

node half :  bool
node half :: 'a
node over :  int -> int
node over :: 'a on half -> 'a
```

Because clocks may contain expressions, some restrictions are imposed on programs. These restrictions are illustrated on some examples. Consider the programming of a (real clock). It receives a `second` input signal and returns three signals: `hour`, `minute` and `second`.

```
(* returns true every n instants *)
let counter n = o where
  rec (c,o) = (0,false) -> if pre c = n then (0,true)
                           else pre c + 1,false

let filter n top =
  top when (counter n)

let hour_minute_second second =
  let minute = filter 60 second in
  let hour = filter 60 minute in
```

```
  hour,minute,second

node counter :  int -> bool
node counter :: 'a -> 'a
node filter :  int -> 'a -> 'a
node filter :: (x__3:'a) -> 'a -> 'a on counter x__3
node hour_minute_second :  'a -> 'a * 'a * 'a
node hour_minute_second :: 'a -> (('a on counter 60) on counter 60) *
                               ('a on counter 60) * 'a
```

In this example, the parameter `x__3` is substituted by `60`. It is also possible to hide some expression variables used in clocks by returning them as outputs. For example:

```
let filter n top1 = (top2,o) where
  rec (c,o) = (0,false) -> if pre c = n then (0,true)
                          else pre c + 1,false
  and top2 = top when o

node filter :  int -> bool -> bool * bool
node filter :: 'a -> 'a -> ('a on x__4) * (x__4:'a)
```

Thus, expressions in clocks may contain variables which are, either globals, input parameters or outputs.

1.2 Higher order reactive features

From now on the proposed language appears to be more or less equivalent to Lustre, up to syntactic details.

The language is a functional language: functions are first-class objects which can be given to functions or returned by functions. For example, the `it` function feeds back a network (i.e, it iterates a function on a stream of values):

```
let it f init x = y where
  rec y = f x (init fby y)

node it :: ('a -> 'b -> 'b) -> 'b -> 'a -> 'b
```

such that:

```
sum x = it (+) 0 x
```

One can define some composition operators:

```
let (||) f g x = (f x,g x)

let (>) f g x = g (f x)

let rec half = true -> not (pre half)
```

```
let alternate f g x = merge half (f (x when half)) (g (x when not half))

node (||) :: ('b -> 'a) -> ('b -> 'a) -> 'a * 'a
node (>) :: ('a -> 'c) -> ('c -> 'b) -> 'a -> 'b
node half :: 'a
node alternate :: ('a on half -> 'a on half) ->
                  ('a on not half -> 'a on not half) -> 'a -> 'a
```

The infix operator `(||)` computes the synchronous product of two functions `f` and `g` and `(>)` composes two functions. `alternate` alternates the execution of a function with an other one.

All the programs defined above still define reactive systems: programs are compiled into finite state machines answering in bounding time and space whatever be their input.

1.3 Non reactive higher-order features

Besides this functional facility, we can also define recursive functions, such as the celebrated Eratosthenes sieve:

```
let first x = v
  where rec v = x fby v

let rec sieve x =
  let filter = (x mod (first x))<> 0
  in  merge filter
            (sieve (x when filter))
            (true fby false)

let rec nat = 2 fby (nat + 1)

let primes = nat when sieve nat

node first :  'a -> 'a
node first :: 'a -> 'a
node sieve :  int -> bool
node sieve :: 'a -> 'a
node nat :  int
node nat :: 'a
node primes :  int
node primes :: 'a on sieve nat
```

This program is no more real-time since the time and memory to answer at every instant grows.

A compilation option `realtime` is provided for restricting the language to define only real-time programs.

1.4 Polymorphism

In LUCID SYNCHRONE, stream operator (`pre`, `fby`,...) can be applied to any stream value. For example:

```
let my_fby x y = x fby y

let from2 res =
  let rec (i1,i2) =
     (0,0) -> if res then (0,0)
              else (pre i1 + 1,pre i2 - 1) in
  (i1,i2)

node my_fby :  'a -> 'a -> 'a
node my_fby :: 'a -> 'a -> 'a
node from2 :  int -> int * int
node from2 :: 'a -> 'a * 'a
```

`fby` has a polymorphic type where the type variable `'a` may be instantiated by any value which can be converted to a stream. Streams are thus defined in term of classes of types. Informally, this class verifies:

- A scalar type (e.g, a basic type or an imported type) belongs to the class of streams.
- A product type of types belonging to the class of streams belongs to the class of streams.
- A function type does not belong to the class of streams.

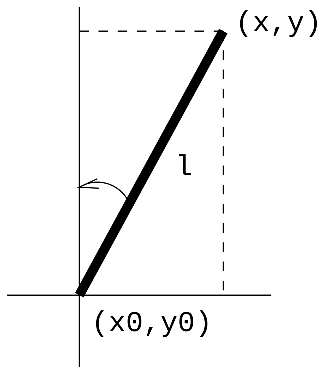
1.5 Compilation

We illustrate the compilation and communication with the host language through the example of an inverted pendulum.

1.5.1 The inverted pendulum

Consider the problem of balancing an inverted pendulum with the hand (through a mouse). The inverted pendulum has a length l, its bottom has coordinates x_0 and y_0 which are continuously controlled by the user and it forms an angle θ with the vertical. This pendulum is submitted to the following law:

$$
\begin{aligned}
&l * \frac{d^2\theta}{dt^2} = sin(\theta) * (\frac{d^2 y_0}{dt^2} + g) - (cos(\theta) * \frac{d^2 x_0}{dt^2})\\
&x = x_0 + l.sin(\theta)\\
&y = y_0 + l.cos(\theta)
\end{aligned}
$$



We suppose that some auxiliary scalar functions have been defined in a Objective Caml module `Draw` with implementation `draw.ml` and interface `draw.mli`. A pendulum is characterized by its bottom and top coordinates. The exported values are defined below:

```
(* file draw.mli *)
type pendulum
val make_pend : float -> float -> float -> float -> pendulum
val clear_pend : pendulum -> unit
val draw_pend : pendulum -> unit
val mouse_pos : unit -> float * float
```

We start by defining a synchronous module for integrating and deriving a signal.

```
(* file misc.ls *)
(* rectangle integration *)
let integr t dx = let rec x = 0.0 -> t *. dx +. pre x in x

(* derivative *)
let deriv t x = 0.0 -> (x -.(pre x))/. t
```

Now, the main module defines global constants and the pendulum law.

```
(* file pendulum.ls *)
open Draw
open Misc

let t = 0.05 (* sampling frequency *)
and l = 10.0 (* length of the pendulum *)
and g = 9.81 (* acceleration *)

let integr dx = Misc.integr t dx
let deriv x = Misc.deriv t x

(* the equation of the pendulum *)
let equation d2x0 d2y0 = theta
where
  rec theta =
      let thetap = 0.0 fby theta
      in integr (integr ((sin thetap)*.(d2y0 +. g)
                        -.(cos thetap)*.d2x0)/.l)

let position x0 y0 =
  let d2x0 = deriv (deriv x0)  in
  let d2y0 = deriv (deriv y0)  in

  let theta = equation d2x0 d2y0 in

  let x = x0 +. l*.(sin theta)  in
```

```
  let y = y0 +. l*.(cos theta)  in
  make_pend x0 y0 x y
```

As in Objective Caml, an `open` *Module* directive makes the names exported by the module *Module* visible in the current module [1]. Imported values may be either used with the dot notation (e.g, `Draw.mouse_pos`) or directly (e.g, `make_pend`) once the module is opened.

Finally the main function continuously reads informations from the mouse, computes the position of pendulum, clear its previous position and draw its current position. We get:

```
let play () =
  let x0,y0 = mouse_pos () in
  let p = position x0 y0 in
  clear_pendulum (p fby p);
  draw_pendulum p;;
```

Now, all the files must be compiled. The compiler of Lucid Synchrone acts as a pre-processor: the compilation of the implementation `misc.ls` produces a file `misc.ml` and a compiled interface `misc.lzi` containing informations about types and clocks of the implementation. Similarly, the compilation of the scalar interface `draw.mli` produces a compiled interface `draw.lzi`. Files are compiled by typing:

```
% lucyc draw.mli          => draw.lzi
% lucyc misc.ls           => misc.ml, misc.lzi
% lucyc -sampling 0.05 -s play pendulum.ls
                          => pendulum.ml, pendulum.lzi, play.ml
% ocamlc draw.mli
% ocamlc draw.ml
% ocamlc pendulum.ml
% ocamlc play.ml
% ocamlc -o play draw.cmo pendulum.cmo play.cmo ...
```

The whole system is obtained by linking all the modules (synchronous and scalars ones) and by sampling the main transition function `play` on a timer (here, 0.05 seconds) giving the base clock (the real-time clock) of the system.

[1] The implemented module system of Lucid Synchrone is very limited and does not compare with the very powerful Objective Caml module system. In fact, it is based on the Caml Light module system, giving the minimal tools for importing names from Objective Caml files or from Lucid Synchrone files.

Part II

Reference manual

Chapter 2

The language

The language is built on top of Objective Caml [7], an ML language developed at INRIA. Many parts of this reference manual are common to Objective Caml and are borrowed from its reference manual, with the permission of the author. The present document should be used in complement with the Objective Caml reference manual.

2.1 Lexical conventions

Lexical conventions for blanks, comments, identifiers, integer literals, floating-point literals, character literals, string literals, prefix and infix symbols are the one of Objective Caml.

Keywords

The following identifiers are keywords.

```
let   and   if    then  pre    or
rec   where fun   else  not    open
when  fby   merge reset every  function
```

The following character sequences are also keywords:

```
->    >     <     =     <>     >=       )     &
+     -     *     /     ;;     <=       (     .
```

2.2 Values

2.2.1 Basic values

Lucid Synchrone only implements the basic values of Objective Caml with the same convention, that is, integer numbers, floating-point numbers, characters and character strings.

2.2.2 Tuples

Lucid Synchrone implements the tuples of Objective Caml, with the same conventions.

Records, arrays and variant values are not provided.

Functions

Mapping from values to values

2.3 Global names

The conventions about names in LUCID SYNCHRONE is inherited from OBJECTIVE CAML with some restrictions [1]. They are listed here:

Names in LUCID SYNCHRONE are decomposed into the following syntactic classes:

- The syntactic class *value-name* for value names
- The syntactic class *typeconstr-name* for type constructors
- The syntactic class *module-name* for module names

2.3.1 Naming values

value-name	::=	*lowercase-ident*
	|	(*operator-name*)
operator-name	::=	*prefix-symbol* | *infix-symbol* | `*` | `=` | `or` | `&`
typeconstr-name	::=	*lowercase-ident*
module-name	::=	*capitalized-ident*

As in OBJECTIVE CAML, the syntactic class of *lowercase-ident* is the set of identifiers starting with a lowercase letter whereas *capitalized-ident* is the set of of identifiers starting with a capital letter.

2.3.2 Referring to named values

value-path	::=	*value-name*
	|	*module-name.value-name*
typeconstr	::=	*typeconstr-name*
	|	*module-name.typeconstr*

A value can be referred either by its name or by qualifying the name with a module name.

2.4 Types

type	::=	' *ident*
	|	(*type*)
	|	*type* `->` *type*
	|	(*type* {* *type*}
	|	*type typeconstr*
	|	(*type* {, *type* }) *typeconstr*
	|	*typeconstr*

[1] In fact, the naming convention are closer to the one of CAML LIGHT but the adopted syntax is the one of OBJECTIVE CAML.

The precedences are given in the following table. The constructions with higher precedences come first.

Operator	Associativity
Type constructor application	-
`*`	-
`->`	right

Type variables

' *ident* denotes the type variable *ident*.

Parenthesized types

(*type*) stands for *type*.

Function type

The type $type_1$ `->` $type_2$ denotes the type of functions mapping arguments of type $type_1$ to results of type $type_2$.

Tuple types

The type $(ty_1 * ... * ty_n)$ is the type of expressions evaluating to $(v_1, ..., v_n)$ where v_i is of type ty_i.

Constructed types

A n-ary type constructor *typeconstr* must be applied to n types. The syntactical conventions are the one of Objective Caml.

2.5 Clocks

clock ::= ' *ident*
| (*clock*)
| *clock* `on` *expr*
| *named-clock* `->` *named-clock*
| (*clock* {* *clock*})

named-clock ::= (*ident* : *clock*)
| (*named-clock* {* *named-clock* })
| *clock*

The precedences are given in the following table. The constructions with higher precedences come first.

Operator	Associativity
`on`	-
`:`	-
`*`	-
`->`	right

Clock variable

' *ident* denotes the clock variable *ident*.

Parenthesized clock

(*clock*) stands for *clock*.

Sub-clock

The clock *clock* `on` *expr* denotes the sub-clock of *clock* when *expr* is true.

Dependent and independent function clock

The clock *named-clock*$_1$ `->` *named-clock*$_2$ denotes the clock of a function, say f, mapping arguments on clock *named-clock*$_1$ to results on clock *named-clock*$_2$. The names in *named-clock*$_1$ denote the input parameters of f whereas the names in *named-clock*$_2$ denote the outputs of f. When no name is present in *named-clock*$_1$ nor *named-clock*$_2$, the function clock is said independent.

Tuple clock

The clock $(clock_1 * ... * clock_n)$ is the clock of expressions evaluating to $(v_1, ..., v_n)$ where v_i is on clock $clock_i$.

Named clock

A named clock is either a regular clock, a product of named clock or has a form *(ident:clock)* meaning that the parameter *ident* has clock *clock*.

2.6 Constants

$$\begin{array}{lcl} \textit{constant} & ::= & \textit{integer-literal} \\ & \mid & \textit{float-literal} \\ & \mid & \textit{char-literal} \\ & \mid & \textit{string-literal} \\ & \mid & \textit{boolean-literal} \end{array}$$

Constants are made of literals from the firth base types (integers, floating-point numbers, characters, character strings and booleans).

2.7 Patterns

$$\begin{array}{lcl} \textit{pattern} & ::= & \textit{ident} \\ & \mid & (\ \textit{pattern}\ ) \\ & \mid & \textit{pattern}\ ,\ \textit{pattern}\ \{,\ \textit{pattern}\} \\ & \mid & \texttt{()} \end{array}$$

Patterns allow selecting data structures of a given shape and binding identifiers to components of the data structure. In the current implementation, only tuple patterns are provided.

Variable pattern

The pattern made of an identifier matches any value.

Parenthesized pattern

The pattern *(pattern)* matches the same values as *pattern*.

Tuple patterns

The pattern $pattern_1$,...,$pattern_n$ matches the value $(v_1, ..., v_n)$ if the pattern $pattern_i$ matches the value v_i.

Void pattern

The pattern `()` is the empty pattern.

2.8 Expressions

expr	::=	*ident*
	|	*global-name*
	|	*constant*
	|	*(expr)*
	|	*expr , expr* {, *expr* }
	|	*expr expr* { *expr* }
	|	`fun` *multiple-matching*
	|	`function` *simple-matching*
	|	*expr* `where` [`rec`] *let-binding*
	|	`let` [`rec`] *let-binding* {`and` *let-binding*} `in` *expr*
	|	*prefix-op expr*
	|	*expr infix-op expr*
	|	*expr & expr*
	|	*expr* `or` *expr*
	|	`not` *expr*
	|	*expr* `when` *expr*
	|	`if` *expr* `then` *expr* `else` *expr*
	|	`merge` *expr expr expr*
	|	*expr* `fby` *expr*
	|	`pre` *expr*
	|	*expr* `->` *expr*
	|	`reset` *expr expr* { *expr*} `every` *expr*

simple-matching ::= *pattern* `->` *expr*

multiple-matching ::= *pattern list* `->` *expr*

pattern-list ::= *pattern* { *pattern* }

let-binding ::= *pattern* `=` *expr*
| *ident pattern-list* `=` *expr*

infix-op ::= *infix-symbol* | `*` | `=` | `or` | `&`

The following table gives the relative precedences and associativity of operators. Higher precedences come first.

Operator	*Associativity*
prefix-symbol	-
function application	*right*
`pre`	-
`fby`	*left*
- -.	-
*...	*right*
... /... %... *mod*	*left*
+... -...	*left*
	right
comparisons	*left*
not	-
& &&	*left*
`or`‖	*left*
,	-
`merge`, `if`	-
`let`, `fun`, ...	-
`->`	*right*

2.8.1 Simple expressions

Constants

Expressions consisting in a constant evaluate to an infinite stream made of this constant.

Variables

Expressions consisting in a variable evaluate to the value bound to this variable in the current evaluation environment.

Parenthetised expressions

The expression (expr) have the same value than expr.

Function abstraction

The way, function abstractions are defined and evaluated is common to OBJECTIVE CAML. A function abstraction has the form:

`fun` *pattern*$_1$ ... *pattern*$_n$ `->` *expr*

Functions with only one argument can be defined with the `function` keyword instead of `fun`.

`function` *pattern* `->` *expr*

This form is only kept in LUCID SYNCHRONE to provide a syntax similar to the one of OBJECTIVE CAML.

Function application

The expression *expr*$_1$ *expr*$_2$ is an application. The expression *expr*$_1$ must evaluate to a functional value which is applied to the value of *expr*$_2$.

The expression *expr*$_1$ *expr*$_2$ ... *expr*$_n$ stands for *(...(expr*$_1$ *expr*$_2$*) ... expr*$_n$*)*. No evaluation order is specified.

When *expr*$_1$ is a function imported from the host language OBJECTIVE CAML and *expr*$_2$ is a stream then *expr*$_1$ *expr*$_2$ stands for the point-wise application of *expr*$_1$ to every element of *expr*$_2$.

Local definitions

The `let` and `let rec` constructs bind variables locally. The expression

`let` *pattern*$_1$ = *expr*$_1$ `and` ... `and` *pattern*$_n$ = *expr*$_n$ `in` *expr*

has the meaning of:

(`fun` *(pattern*$_1$*,...,pattern*$_n$*)* `->` *expr) (exp*$_1$*,...,expr*$_n$*)*

An alternate syntax is provided to define functional values. The definition

ident = `fun` *pattern*$_1$ ... *pattern*$_n$ `->` *expr*

can be declared in the following way:

ident *pattern*$_1$ ... *pattern*$_n$ = *expr*

Both forms define *ident* to be a function with n arguments.

Recursive definitions of variables are introduced by `let rec`:

`let rec` *pattern*$_1$ = *expr*$_1$ `and` ... `and` *pattern*$_n$ = *expr*$_n$ `in` *expr*

This defines *patternt*$_1$*...pattern*$_n$ as mutually recursive values local to *expr*. The *expr*$_i$ are not necessarily functions.

Reverse local definition

The language provides an alternate form of local definitions written in a reverse order. In this way, functions may be defined is a way similar to LUSTRE. The expression:

expr `where` *[*`rec`*]* $pattern_1 = expr_1$ `and` ... `and` $pattern_n = expr_n$

has the meaning of:

`let` *[*`rec`*]* $pattern_1 = expr_1$ `and` ... `and` $pattern_n = expr_n$ `in` *expr*

2.8.2 Operators

The operators written `infix-op` in the grammar can appear in infix position (between two expressions). The operators written `prefix-op` in the grammar (section 2.8 can appear in prefix position (in front of an expression).

Classical operators provided by OBJECTIVE CAML (from the `Pervasives` module) are imported. As for general scalar value imported from the host language, they become stream operators which are applied point-wisely to streams.

Delays

The expression `pre` *expr* is the delayed stream. *expr* must be a stream. The clock of the result is the clock of *expr*. The n-th value of the result is the $n-1$-th value of *expr*. Its value at the first instant is undefined.

The binary operator `fby` is the initialized delay operator. The first value of $expr_1$ `fby` $expr_2$ is the first value of $expr_1$. Its n-th value is the $n-1$-th value of $expr_2$.

Initialization

$expr_1$ `->` $expr_2$ initializes a stream. The $expr_i$ must be streams of the same type and on the same clock. It returns a stream with the same type and clock. The first value of the result is the first value of $expr_1$. Then, the n-th value of the result is the n-th value of $expr_2$.

Point-wise conditional

The expression `if` $expr_1$ `then` $expr_2$ `else` $expr_3$ is the point-wise conditional. $expr_1$ must be a boolean stream, $expr_2$ and $expr_3$ two streams of the same type. The type of the result is the type of $expr_2$. The expressions $expr_i$ must be on the same clock. The clock of the result is the clock of $expr_1$. The conditional returns a stream such that its n-th value is the n-th value of $expr_2$ if the n-th value of $expr_1$ is true and the n-th value of $expr_3$ otherwise.

Under-sampling

The expression $expr_1$ `when` $expr_2$ is the under-sampling operator. $expr_1$ must be a stream and $expr_2$, a boolean stream. The type of the result is the type of $expr_1$. The expressions $expr_i$ must be on the same clock *cl*. The clock of the result is a sub-clock *cl* `on` $expr_2$. This expression returns the sub-stream of $expr_1$ defined for all instants where $expr_2$ is defined and is true.

Over-sampling

The expression `merge` $expr_1$ $expr_2$ $expr_3$ allow two complementary streams to be merged. $expr_1$ must be a boolean stream, $expr_2$ and $expr_3$ two streams of the same type. The type of the result is the type of $expr_2$. If $expr_1$ is on clock *cl*, $expr_2$ must be on clock *cl* `on` $expr_1$ ($expr_2$ must be present when $expr_1$ is present and true) and $expr_3$ must be on clock *cl* `on not` $expr_1$. This expression returns a stream such that its n-th value is the n-th value of $expr_2$ if the n-th value of $expr_1$ is true and the n-th value of $expr_3$ otherwise.

Reseted application

`reset`*expr* $expr_1$ ... $expr_n$ `every` $expr_{n+1}$ allows for resetting a function application. The $expr_i$ and the result of this construction must be streams, all on the same clock. This construction acts as a regular function application except that all the streams defined in expr_1 restart to their initial value every time the current value of $expr_{n+1}$ is true.

2.9 Type definition

Abstract types can be defined. Their syntax is inherited from OBJECTIVE CAML and is reminded here.

type-definition	::=	`type` *typedef* { `and` *typedef*}
typedef	::=	[*type-params*] *typeconstr-name*
type-params	::=	' *ident*
	|	('*ident*{, '*ident*})

2.10 Module implementation

implementation	::=	{*impl-phrase* [`;;`] }
impl-phrase	::=	*value-definition*
	|	`open` *module-name*
	|	`type` *type-definition*
value-definition	::=	`let` [`rec`] *let-binding* {`and` *let-binding*} [`;;`]

A module implementation consists in a sequence of implementation phrases. An implementation phrase either opens a module, is a type definition or is a sequence of definitions.

- The instruction `open` modifies the list of opened modules by adding the module name to the list of opened modules, in first position.
- The type definition defines the type for the implementation phrases following the definition.
- The value definition defines some global values.

2.11 Scalar interfaces

Scalar interfaces written in OBJECTIVE CAML can be imported by LUCID SYNCHRONEİn the current implementation, a restricted subset of OBJECTIVE CAML interfaces is considered. The syntax is the following:

scalar-interface	::=	{ *scalar-interface-phrase* [`;;`]}
scalar-interface-phrase	::=	*value-declaration*
	|	*type-definition*
value-declaration	::=	`val` *ident* `:` *type*

When a value is imported from the host language OBJECTIVE CAML the value is automatically lifted to the stream level in the following way.

- A scalar value with a basic or declared type becomes a infinite stream of that type.
- A scalar functional value becomes a stream functional value applied point-wisely to its argument.

2.12 Denotational semantics in Haskell

This section gives the denotational semantics of a LUCID SYNCHRONE kernel in a few lines of Haskell [6]. It is based on a minimal set of stream primitives: a constant generator `constant`, a point-wise application `extend`, an initialized delay `fby`, a sampling operator `when`, an over-sampling operator `merge` and a reset function `reset`. We take some liberty with the Haskell syntax to keep our presentation simpler.

Whereas this denotational semantics gives a simulator of LUCID SYNCHRONE programs for free, it does not give a compiler. LUCID SYNCHRONE programs are compiled into state machines, as it is the case in existing synchronous data-flow languages.

```
module Lucid_synchrone
where

-- infinite streams
data Stream a = Pre a (Stream a)

-- Constant generator
constant e = Pre e (constant e)

-- Point-wise extension
extend (Pre f fs) (Pre e es) = Pre (f e) (extend fs es)

-- Initialised delay
(Pre x xs) fby ys = Pre x ys

-- Sampling
(Pre e es) when (Pre True c)  = Pre e (es when c)
(Pre e es) when (Pre False c) = es when c

-- Oversampling
merge (Pre True xs1)  (Pre x2 xs2) xs3          = Pre x2 (merge xs1 xs2 xs3)
merge (Pre False xs1) xs2          (Pre x3 xs3) = Pre x3 (merge xs1 xs2 xs3)

-- Derived operator
ifthenelse c x y =
  extend (extend (extend (constant (\c x y -> if c then x else y)) c) x) y

True = extend (Prelude.True)

False = extend (Prelude.False)

x -> y = ifthenelse (True fby False) x y

not = extend (constant (Prelude.not))

& = extend (constant (Prelude.&))

pre x = Pre dummy x -- dummy is any value of the good type

- Reset
whilenot c = let rec c0 = False -> pre c0 & not c in c0

reset f x c =
  let c0 = whilenot c in
  merge c0 (f (x when c0))
           (reset f (x when not c0) (c when not c0))
```

Figure 2.1: Denotational semantics in Haskell

Chapter 3

The batch compilation

This part describes how to transform Lucid Synchrone programs into Objective Caml programs. This is achieved by the command `lucyc`. `lucyc` accepts two kinds of arguments:

- Arguments ending in `.ls` are considered to be Lucid Synchrone source files. A file `f.ls` is a sequence of node declarations. From a file `f.ls`, the `lucyc` compiler produces an Objective Camlfile `f.ml` and a compiled interface `f.lzi`.
- Arguments ending in `.mli` are considered to be Objective Caml interface files. From a file `f.mli`, the `lucyc` compiler produces a compiled interface `f.lzi`. Every value defined in `f.mli` is considered to be a scalar value.

3.1 Options

The following options are accepted by the `lucyc` command:

`-c`
: Compile only. Produces a file ending in `.ml` defining the corresponding transition functions of the nodes defined in the input file.

`-I lib-dir`
: Adds `lib-dir` to the path of directories searched for compiled interface files `.lzi`.

`-O module`
: First opens the module `module`.

`-s node`
: Produces a file `node.ml`, simulating the main node node.

`-sampling rate`
: Sets the sampling rate to `<rate>` seconds for the simulation node.

`-i`
: Print types and clocks on the standard output.

`-realtime`
: Real-time mode of the compiler. Recursively defined functions are forbidden.

`-v`

Prints the compiler version number.

Chapter 4

Common errors

Syntax errors, typing errors and errors coming from the module system are similar to the one of Objective Caml. They are not detailed here.

Three kind of errors specific to Lucid Synchrone may appear in a program: clocking errors, causality errors and initialization errors. When an error occurs, the compiler displays some lines of the source code where the errors occured and points the location of the error with an error message.

Clocking errors

Several clocking errors may appear.

`This expression has clock` *cl* `but is used with clock` *cl'* The actual clock of the pointed expression does not match its expected clock.

`This expression should match a pattern of the form` *pat* The structure of the actual parameters in a function application does not match the formal parameters of the function.

`This pattern should match a pattern of the form` *pat* The structure of the result of a function application does not match the structure of the pointed pattern.

`The clock of this expression, that is,` *cl* `depends on some variables which must be bounded by a let.` The clock of the result of an application depends on some output parameters which must be defined.

`This expression has clock` *cl* `which contains undefined variables.` The clock of the function depends on some variables which are not globals or parameters of the function. These variables must be given as outputs of the function.

Causality errors

`Causality loop.` The pointed variable `x` belongs to a causality loop: it depends instantaneously on itself. The problem may be fixed by inserting a delay (`pre` or `fby`) in the loop.

Initialization errors

`This expression must be initialized.` The stream expression does not have an initial value. The problem may be fixed by inserting an initialization (`->` or `fby`).

Bibliography

[1] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. Academic Press, 1985.

[2] E.A. Ashcroft and W.W. Wadge. Lucid, a non procedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[3] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[6] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1990.

[7] Xavier Leroy. The Objective Caml system release 2.01. Documentation and user's manual. Technical report, INRIA, 1999.