Certifying synchrony for free

Sylvain Boulmé and Grégoire Hamon

Laboratoire d'Informatique de Paris 6 (LIP6),
8, rue du Capitaine Scott, 75015 Paris, France.
`Sylvain.Boulme,Gregoire.Hamon@lip6.fr`

Abstract. We express reactive programs in Coq using data-flow synchronous operators. Following Lucid-Synchrone approach, synchronous static constraints are here expressed using dependent types. Hence, our analysis of synchrony is here directly performed by Coq typechecker.

1 Introduction

Synchronous languages [Hal93] have been designed to help in the conception of reactive systems, especially critical reactive systems (planes, power plants control...). *Synchrony* is a program property which ensures bounded reaction-time and memory at execution. Synchronous languages statically check this property, however, in a critical context, it may be needed to have a formal proof of this property, or more generally to prove program properties.

In this work we are interested in Lucid-Synchrone [CP00] (LS for short), a *data-flow synchronous language*. We present here a natural and *shallow* embedding of LS into the Coq proof assistant. This embedding concerns both the dynamic and the static semantics of the language, such that synchrony analysis is obtained for free. Moreover, it gives us a denotational semantics of LS in Coq and is thus a good starting point for designing a prover for LS programs in Coq, following [Fil99,Par95] approach. This semantics can also be used to experiment with the language: we have used it to propose a notion of recursive functions for LS, as a generalization of recursive streams (see [BH01]).

We are now going to quickly introduce LS and Coq. These introductions are just aiming at making this paper self-explanatory, for more details on these tools, you should consult their tutorials and reference manuals ([PCCH01][1], [BBC$^+$00]).

1.1 Lucid-Synchrone

Lucid-Synchrone is a language born to bring synchronous data-flow languages like Lustre [HCRP91] closer to general functional languages, in particular ML. This was needed for several reasons. First, synchronous languages were in need of more mechanisms of abstraction. Second, to better understand the link between these two families of languages, third to use of this link in a certification or proof

[1] The LS compiler is available at: `http://www-spi.lip6.fr/lucid-synchrone/`

context. LS brings good answers to the first of these needs, it has inspired the next generation of the industrial tool SCADE. We are here interested in the third of these needs, and in formally certifying or proving programs.

LS is an ML like language over streams (or *data-flow*): all values are infinite sequences. This streams can be on each instant either absent or present. They are equipped with a *clock*, a boolean information indicating on each instant whether the stream is present or absent. The clock which is always `true` is called the *basic clock* of the system. To ensure synchrony, clocks have to verify some constraints, this is done by a static analysis, the *clock calculus* [Cas92]. Previous works on LS [CP98] have shown that it could be defined as a ML type system extended with dependent types. This has allowed the LS compiler to provide clocks polymorphism and inference.

The compiler also performs other static analysis. Especially, it checks *causality* in recursive definitions. It means that instantaneous recursion is forbidden: the current value of a stream should not depend on itself. This property is also represented in our encoding.

1.2 Coq

Coq is a proof assistant based on the Calculus of Constructions [CH88], enriched with inductive [PM93] and co-inductive [Gim94] type definitions. Calculus of Constructions is a typed lambda-calculus, with *types* as first-class values (i.e. types are terms). Hence, *terms* may be:

- *identifiers*, defined constants or declared variables in the context.
- the *application* of a functional term `M` to a term `N`, denoted by "`(M N)`".
- the *abstraction* of a variable `x` of type `A` in a term `b`, denoted by "`[x:A]b`".
- the *product* of a type family `B` indexed by a variable `x` of type `A` (i.e. the type of functions "`[x:A]b`" where `b` is of type `B`, assuming `x` of type `A`). It is generally denoted by "`(x:A)B`" or simply "`A->B`" if `x` does not occur in `B`.

In the spirit of Curry-Howard isomorphism, types may represent programming datatypes or logical propositions. This is expressed in Coq using two special types of types, `Set` as type of datatypes, and `Prop` as type of propositions. Hence, "`(x:A)B`" may also be interpreted as "for all `x` in `A`, `B`" and "`->`" as the logical implication.

Via coinductive type definitions [Gim98], Coq provides infinite values and co-recursion[2]. We have used them to define synchronous streams in Coq.

Coq provides syntactic features, such as implicit terms (denoted by a "`?`") or defined constants with implicit arguments, that force the system to try some type inference *à la* ML[3]. We have heavily used these features to provide a kind of clock inference.

[2] Let us note however that Coq is strongly normalizing: hence, all computations terminate, even on infinite values.

[3] with the following exceptions: there is no mechanism for polymorphic generalization and inference may fail due to undecidability of high-order unification.

1.3 Related works

Valid LS programs check some non-trivial static properties. Some of these properties are guaranteed by static analyses performed by the compiler, rejecting non-valid programs. Others are guaranteed by construction, i.e. by the semantics of the language. A first difficulty was to understand how to integrate these properties in our description. If we are only interested in proving properties about LS programs, then a simple approach consist in stating these properties as axioms or hypothesis in the description, relying on a compiler to check these properties. This approach is for instance suggested by [BCDPV99]. Here, we are more ambitious. We want to prove the existence of LS programs, and their static fundamental properties, independently of any external system. This is also the approach of [DCC00]. In this approach, a compiler could still be used to provide some Coq proofs from its static analysis, thus easing the work on Coq.

Synchronous languages have been introduced for software and hardware reliability. Thus, formalizing synchronous systems is an active research area. A lot of case studies have be done, using many formalisms [KNT00,BCDPV99,CGJ99]. There are also some axiomatizations of synchronous languages [BCDPV99,NBT98]. Our work thus lies in this latter category. Our main originality here, is to embed the clock calculus of LS as a special instance of Coq type system. LS is well-suited for this study, as its clock calculus is expressed as a dependent type system whereas other synchronous data-flow languages (Lustre, Signal [BLJ91]) use specific analysis.

2 Sampled Streams and their equality in Coq

We first define the type of *"sampled elements of* `A`*"*, to represent instantaneous values of type `A` in synchronous streams. On a given instant, a sampled element can be either present or absent. We also introduce a present error element, which is raised in case of instantaneous recursion.

Then, we define clocks as streams of booleans. The type of *"sampled streams"* can then be defined as a coinductive type parametrized by a type `A` and a clock `c`. Its single constructor `sp_cons` takes as arguments a sampled element and a sampled stream.

```
Inductive samplElt[A:Set]: bool -> Set:=
 | None: (samplElt A false)
 | Any: A -> (samplElt A true)
 | Fail: (samplElt A true).

Definition clock:=(Stream bool).

CoInductive samplStr[A:Set]: clock -> Set:=
 sp_cons: (c:clock)
            (samplElt A (hd c))->(samplStr A (tl c))->(samplStr A c).
```

We define two destructors `sp_hd` and `sp_tl` for accessing respectively the head and the tail of sampled streams. We have of course the following property:

```
Lemma unfold_samplStr:
 (c:clock; s:(samplStr A c))s=(sp_cons (sp_hd s) (sp_tl s)).
```

The equality we consider over sampled streams, noted `sp_eq`, is bisimulation or extensional equality. It is defined for streams that may not have *interconvertible* clocks, but only *bisimulable* clocks. We have also defined a notion of "clock coercion" compatible with bisimulation (see [BH01]).

We are actually only interested in programs producing *wellformed* sampled streams, i.e. sampled streams without `Fail` element. Hence, we have defined a logical predicate `sp_wf` indicating wellformness of sampled streams (see [BH01]).

3 Lucid-Synchrone data-flow operators in Coq

We now detail the encoding of LS data-flow operators. The language provides three kinds of operators, *point-wise* operators are classical operators like $+$, $-$, ... lifted to streams, sampling operators are streams operators allowing under or over-sampling and *delay* operators allow delaying a stream.

3.1 Point-wise operators

First, every object of the base language is lifted into a constant sampled stream. As in LS, these constants have polymorphic clocks. Operator `elt_const` returns a sampled element: either `Any` of its argument or `None` depending on the clock. The `sp_const` operator only iterates it on every instant. To make the following Coq code more readable, we have replaced by "$\cdots$" some type information needed by typechecking. By convention, `A` and `B` are variables of type `Set`.

```
Variable a: A.

Definition elt_const : (b:bool)(samplElt A b)
  := [b]<···>if b then (Any a) else (None A).

CoFixpoint sp_const : (c:clock)(samplStr A c)
  := [c](sp_cons (elt_const (hd c)) (sp_const (tl c))).
```

Lifted functions will then be applied point-wisely on streams. `sp_extend` takes a stream of functions and a stream of arguments, and returns the stream of their point-wise application. Functions and arguments are required to be on the same clock, thus, no memory is required by this function. As before, `elt_extend` is the instantaneous application on sampled elements, and `sp_extend` iterates it on every instant.

An interesting application of `sp_extend` is the definition of `sp_if`, the point-wise if-then-else (or multiplexer in circuits):

```
Definition If:= [b:bool; x,y:A]if b then x else y.

Definition sp_if:=
 [c:clock; lc:(samplStr bool c); x,y:(samplStr A c)]
  (sp_extend (sp_extend (sp_extend (sp_const If ?) lc) x) y).
```

```
Definition elt_extend:
 (b:bool)(samplElt A->B b)->(samplElt A b)->(samplElt B b)
 := [b; f]<···>Cases f of
          | None => [_](None B)
          | (Any vf) =>
             [x]Cases x of
                | (Any vx) => (Any (vf vx))
                | Fail => (Fail ?)
                | ···
                end
          | Fail => [_](Fail ?)
          end.

CoFixpoint sp_extend:
 (c:clock)(samplStr A->B c)->(samplStr A c)->(samplStr B c)
 := [c;f;x](sp_cons (elt_extend (sp_hd f) (sp_hd x))
                    (sp_extend (sp_tl f) (sp_tl x))).
```

Fig. 1. Definition of `extend` operator

3.2 Sampling operators

Sampling operators are used either to filter a stream (`when`) or to combine several complementary streams (`merge`). The `when` operator filters a streams on a boolean condition: the result is the sub-stream of the entry only present when the boolean condition is true. Thus the clock of the result differs from the one of the entry: it is on a subclock of it, defined by the condition.

We first need an operator to dynamically build such a subclock. The clock constructor `sp_on` coerces a sampled boolean stream into a clock. Hence, this is the type operator introducing dependent types in the LS clock system.

As usual, `elt_on` defines the instantaneous behavior of the constructor, and `sp_on` only iterates it on every instant. With `Fail` as argument, `elt_on` returns true: the error will be diffused by the sampling operations (see `elt_when` below).

```
Definition elt_on: (b:bool)(samplElt bool b)->bool
 := [b;o]Cases o of
         | None => false
         | (Any x) => x
         | Fail => true
         end.

CoFixpoint sp_on: (c:clock)(samplStr bool c) -> clock
 := [c;lc](Cons (elt_on (sp_hd lc)) (sp_on (sp_tl lc))).
```

Now we can define `sp_when`, the sampling operator. It simply copies its input, forgetting elements when clock of the output is false.

```
Definition elt_when:
 (b:bool;o:(samplElt bool b))(samplElt A b)->(samplElt A (elt_on o))
 := [b;o]<···>Cases o of
         | None => [x](None ?)
         | (Any b0) =>
              [x:(samplElt A true)]<···>if b0 then x else (None ?)
         | Fail => [x](Fail ?)
         end.

CoFixpoint sp_when:
  (c:clock; lc:(samplStr bool c))(samplStr A c)->(samplStr A (sp_on lc))
 := [c;lc;x](sp_cons ··· (elt_when (sp_hd lc) (sp_hd x))
                          (sp_when (sp_tl lc) (sp_tl x))).
```

The sp_merge operator is defined in the same way. It is a kind of if-then-else, whose branches have exclusive clocks[4]. It allows a kind of over-sampling, since output is on faster clock than branches of the sp_merge. Its type is:

```
CoFixpoint sp_merge :
 (A:Set; c:clock; lc:(samplStr bool c))
     (samplStr A (sp_on lc))
      ->(samplStr A (sp_on (sp_not lc)))->(samplStr A c).
```

We have the following relation between sp_if, sp_merge and sp_when (where sp_eq is bisimulation over sampled streams):

```
Lemma if_equiv:
 (A:Set; c:clock; lc:(samplStr bool c); x,y:(samplStr A c))
  (sp_wf x)->  (sp_wf y) ->
    (sp_eq (sp_if lc x y)
           (sp_merge (sp_when lc x) (sp_when (sp_not lc) y))).
```

3.3 Delay operators

Delay operators allow referring to past values, thus providing implicit memory manipulations. We consider here the *"followed by"* operator (fby) which is an initialized delay (LS also provides uninitialized delay, and initializer, their full description in Coq can be found in [BH01]).

If x and y are two streams sampled on the same clock, then x fby y is a stream sampled on this clock, whose present elements are, at first, the first present element of x, and then forever the previous present element of y. Hence, x fby y is a delay of y, initialized with the first element of x.

The fact that x and y have the same clock is fundamental here: it guarantees that only one element of y will have to be memorized before x yields its first value, i.e fby is a one place buffer. Hence, clocks provide here a simple way to express static synchronization constraints.

[4] In the type of sp_merge, sp_not is the point-wise negation defined using sp_extend.

Examples in Coq syntax Before defining formally `fby` in Coq, we study some examples to illustrate the expressive power of delays within the rest of the language. It is also the opportunity to show inference mechanisms in Coq, very close to the inference mechanisms of LS. We use here some grammar extensions of Coq, to provide a more friendly syntax:

- lifted constants are given between back-quote as `` `(0)` ``, their clocks are inferred.
- `sp_extend` is abbreviated into `ext`.
- `when` is the infix notation corresponding to the operator `sp_when`, with permuted arguments. (Hence the condition is given on the right).
- `merge` is an abbreviation for `sp_merge`, forcing the condition to be explicit (the condition is implicit in `sp_merge`, which makes the type inference fail on most practical examples).
- `rec0` is the fixpoint operator for streams. It takes a function as argument, which introduces the bounded name for referring to recursive occurrences of the stream being built. `rec0` may be prefixed by "`<c>`" where `c` is the clock of the result. This syntactic feature is useful for binding clock variables which need to be generalized by hand. This operator will be studied in details in section 4.

The sequence of natural numbers can now be written as `Nat` below. Coq infers the type `(c:clock)(samplStr nat c)` for this definition. This stream is polymorphic on clock. This must be explicitly expressed to Coq. Here, Coq infers clocks for `the constants` `` `(0)` `` and `` `S` `` (natural successor function), using `fby` and `ext` clock constraints.

```
Definition Nat := [c](<c>rec0 [N](`(0)` fby (ext `S` N))).
```

We can also define the Fibonacci sequence: [5]

```
Definition Fib :=
  [c](<c>rec0 [fib] (`(1)` fby (ext (ext `plus` fib) (`(0)` fby fib)))).
```

A delicate feature of delays is the fact that they do not permute with sampling operators. For instance, `Nat_on` and `Nat_mod`, both of the same type, differs only in their definition by the position of `when` with respect to `fby`. Their semantics are very different. The former enumerates naturals on `(sp_on lc)`, and returns zero on `(sp_on (sp_not lc))`. The latter indicates at each instant the number of passed instants since `lc` was false. Indeed, in the former case, `when` is applied before `fby`: thus, the clock of `fby` is `(sp_on lc)`. On the contrary, in the latter case, `fby` is on `c`.

```
Definition Nat_on :=
  [c; lc](<c>rec0 [N](merge lc (`(0)` fby ((ext `S` N) when lc)) `(0)`)).

Definition Nat_mod :=
  [c; lc](<c>rec0 [N](merge lc ((`(0)` fby (ext `S` N)) when lc) `(0)`)).
```

[5] In LS, this definition is compiled using only a memory size of two integers.

Formal descriptions of `fby` Below, we first define an internal operator (ie not a LS operator) for delay. It takes a sampled stream and a sampled element on `true` (ie `Fail` or a value) and uses it in order to initialize the delay. It is defined by cofixpoint, and actually this last parameter is used for memorizing the last element of the stream. Then, we define `sp_fby x y`: we wait for the first present element of `x` and use it to delay `y`.

```
CoFixpoint sp_delay:
 (c:clock)(samplStr A c)->(samplElt A true)->(samplStr A c)
 := [c]<···>Cases c of (Cons hc tc) =>
        <···>if hc
             then
               [x;a](sp_cons a (sp_delay (sp_tl x) (sp_hd x)))
             else
               [x;a](sp_cons (None A) (sp_delay (sp_tl x) a))
       end.

CoFixpoint sp_fby:
 (c:clock)(samplStr A c)->(samplStr A c)->(samplStr A c)
 := [c]<···>Cases c of (Cons hc tc) =>
        <···>if hc
             then
               [x;y](sp_delay y (sp_hd x))
             else
               [x;y](sp_cons (None A) (sp_fby (sp_tl x) (sp_tl y)))
       end.
```

4 Recursive constructions

Recursive constructions in LS can not be directly (or syntactically) translated into cofixpoints of Coq: guarded conditions of Coq are too restrictive for this purpose. However, we can build in Coq a generic recursion operator (using cofixpoint of Coq) which emulates recursive constructions of LS. This kind of trick is already used in the standard library of Coq for emulating generic wellfounded recursion with primitive recursion. However, here the computational behavior of our recursive constructions is very far from the one when compiled in LS.

LS provides two notions of recursive constructions: recursive streams and recursive functions of streams. The first are always reactive, but not the second (see [BH01]). Thus, we are especially interested in the first construction. However, we define here a unique operator of recursion, from which both constructions can be derived. As some preconditions of this recursion operator can be simplified in the particular case of recursive streams, we only present this case. We do not present here the formal encoding of this operator (see [BH01]), it is built classically as a smallest fixpoint: by successive approximations using `Fail`. Indeed, for the considered functions, there is at most one fixpoint (see `rec0_uniq` below): the smallest is also the biggest.

4.1 Basic recursion on streams

Recursive streams are built by iterating infinitely a function on itself, using the `rec0` operator used in previous examples. This operator is of type:

```
(A:Set; c:clock)((samplStr A c)->(samplStr A c)) -> (samplStr A c)
```

However, applying `rec0` only makes sense for functions satisfying some preconditions: length-fail-preservation (`F0_lfp`) on the basic clock, and wellformness-increase (`F0_nfstwf_inc`).

```
Variable c: clock.
Variable F0: (samplStr A c) -> (samplStr A c).

Hypothesis F0_lfp:
  (s1,s2:(samplStr A c); n:nat)
    (sp_inf n s1 s2)->(sp_inf n (F0 s1) (F0 s2)).

Hypothesis F0_nfstwf_inc:
  (n:nat; s:(samplStr A c))(glob_nfstwf n s)->(glob_nfstwf (S n) (F0 s)).
```

We now detail the meaning of these preconditions.

Length-fail-preserving functions (on the basic clock) *Length-fail-preservation* on the basic clock is an invariant property satisfied by every LS programs. It deals both with synchrony (length-preservation) and with the "error semantics" of `Fail` (fail-preservation).

Length-preservation on the basic clock means that for every LS function, at every instant, the function takes one sampled element (possibly `None`) on each input and returns one sampled element (possibly `None`). *Fail-preservation* means that functions are monotonous with respect to the flat order with `Fail` as minimum element.

Hence, we have defined `(sp_inf n)` as an order over sampled streams such that `(sp_inf n s1 s2)` means that the `n` first elements of `s1` are inferior for the flat order to their respective element in `s2` (see [BH01]). Now we informally say that a function is length-fail-preserving if for every `n`, it preserves the order `(sp_inf n)`. Length-fail-preservation has been proved for all LS operators.

Wellformness-increase Wellformness-increase is related to causality: length-fail-preserving and causal functions are indeed wellformness-increasing. We first have defined `glob_nfstwf`: a notion of *"partial wellformness" on the n-th first instants of the basic clock*, indicating if a stream has no fail element until the n-th instant of the basic clock (see [BH01]). Being a property on the basic clock, it is easy to handle in proofs.

A function F of type `(samplStr A c) -> (samplStr A c)` is *wellformness-increasing*, if its input is partially wellformed until the n-th instant implies that its output is partially wellformed until the $n+1$-th instant. Thus, by iterating infinitely on F, we build a wellformed stream.

Then, we can prove wellformness-increasing of `sp_fby` on its second argument, and partial wellformness preservation by other operators.

Correctness of `rec0` Under the preconditions given above, `(rec0 F0)` is of type `(samplStr A c)`, and satisfies the three following properties (proven in Coq), stating that `(rec0 F0)` is wellformed, and is the unique fixpoint of `F0`.

```
Lemma rec0_wf: (sp_wf (rec0 F0)).

Theorem rec0_inv: (sp_eq (rec0 F0) (F0 (rec0 F0))).

Theorem rec0_uniq:
  (r0:(samplStr A c))(sp_eq r0 (F0 r0))->(sp_eq (rec0 F0) r0).
```

We have already seen some examples of single recursive streams using `rec0`. The satisfaction of recursive preconditions for each of these examples has been proved in Coq. Most often, these proofs have been completely discharged by the Coq prover using *à la prolog* resolution on a basis of lemmas containing properties of each LS operator with respect to length-fail-preservation and partial wellformness.

4.2 Other recursive constructions

The `rec0` operator can also be used to express *mutually recursive streams* of LS, by putting each mutually recursive occurrence as a parameter of a simple recursive stream and linking mutual recursion by hand. Hence, expressing such mutual definitions of LS in Coq need some syntactic treatment, but it seems automatable. In order to prove that such "mutually recursive definitions" satisfy the preconditions of `rec0`, we have proven that `rec0` has itself a good behavior with respect to length-fail-preservation and partial wellformness (see [BH01]).

Actually, all the previous properties of `rec0` are specialized properties of a more general operator `rec1`. This operator has type:

```
rec1: (A,B:Set; C:(B->clock))
       (((x:B)(samplStr A (C x)))->(x:B)(samplStr A (C x)))
         ->(x:B)(samplStr A (C x))
```

Thus, it allows building recursive functions of type `(x:B)(samplStr A (C x))` where `B` is any type. Motivations and formal encoding of `rec1` are given in [BH01].

5 Conclusion

We have here established formally a kind of morphism from LS into Coq preserving the semantics of strongly typed functional languages with polymorphism, dependent types, and streams:

- streams of LS are represented by sampled streams in Coq.
- functions of LS are represented by functions of Coq.
- clock concepts (polymorphism, dependent types) of LS are represented by the equivalent type concepts in Coq.

This result strengthens both LS and Coq. Indeed, it shows that LS can be derived naturally from a more general language. It also shows that Coq coinductive part is powerful enough to represent programs from the real world.

As a consequence, LS programs can be almost syntactically embedded into Coq. Actually, embedding LS programs into Coq requires expliciting some clock and types parameters to Coq (especially because there is no automatic generalization in Coq over polymorphic variables), but it would be easy to automate this translation. Indeed, most of the clock inference of Lucid-Synchrone is already done by the type inference of Coq. We have illustrated this point on some examples along the paper[6].

Thus, in the future, we may base us on this encoding in Coq to prove automatically the validity of LS programs. Currently automation of these proofs (length-fail-preserving and partial wellformness increasing) is not very powerful: we only use *à la prolog* resolution on a good basis of lemmas. But, we probably could use the compiler analyses to produce these proofs more efficiently. Coq typechecker would then just have to check the correction of these proofs.

A following to this work could be to build tools for proving general properties on LS programs into Coq. It could then be interesting to think about a notion of *"synchronous property"*. Indeed, we may find a logic following Curry-Howard isomorphism such that proofs in this logic are synchronous functions, and then have a theorem of realizability for this logic. It would provide a good framework for synthesizing proofs from programs following Parent approach [Par95].

Another application to this work could be to prove the correction of a LS compiler. It may indeed consist in proving that the compiler satisfies the denotational semantics presented here.

6 Acknowledgements

We would like to thank Marc Pouzet for all the time spend discussing this work and for many useful suggestions.

References

[BBC+00] Barras B, et al. *The Coq Proof Assistant Reference Manual, version 6.3.1*. Coq Project, INRIA-Rocquencourt, May 2000.

[BCDPV99] Bensalem S, et al. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7*. IEEE Computer Society, January 1999.

[BH01] Boulmé S and Hamon G. A clocked denotational semantics for Lucid-Synchrone in Coq. Technical report, LIP6, 2001. Available at `http://www-spi.lip6.fr/lucid-synchrone/lucid_in_coq_report.ps`.

[BLJ91] Benveniste A, et al. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[6] More examples and the whole Coq code of the encoding is available at `http://www-spi.lip6.fr/lucid-synchrone/lucid_in_coq/`

[Cas92] Caspi P. Clocks in dataflow languages. *Theorical Computer Science*, 94:125–140, 1992.

[CGJ99] Coupet-Grimal S and Jakubiec L. Hardware verification using co-induction in Coq. In *TPHOLs'99*, number 1690 in LNCS. 1999.

[CH88] Coquand T and Huet G. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CP98] Caspi P and Pouzet M. A co-iterative characterization of synchronous stream functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, ENTCS. 28-29 March 1998.

[CP00] Caspi P and Pouzet M. Lucid Synchrone, an ML extension to Lustre. In *Submited for publication*. 2000.

[DCC00] Dumas-Canovas C and Caspi P. A PVS proof obligation generator for Lustre programs. In *LPAR'2000*, number 1955 in LNAI. 2000.

[Fil99] Filliâtre JC. *Preuve de programmes impératifs en théorie des types*. Ph.D. thesis, Université Paris-Sud, July 1999.

[Gim94] Gimenez E. Codifying guarded definitions with recursive schemes. Technical Report 95-07, LIP ENS-Lyon, December 1994.

[Gim98] Gimenez E. A tutorial on recursive types in Coq. Technical Report 0221, INRIA, May 1998.

[Hal93] Halbwachs N. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[HCRP91] Halbwachs N, et al. The Synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[KNT00] Kerboeuf M, et al. The steam-boiler problem in Signal-Coq. In *TPHOLs'2000*, number 1869 in LNCS. 2000.

[NBT98] Nowak D, et al. Co-inductive axiomatization of a synchronous language. In *TPHOLs'98*, number 1479 in LNCS. 1998.

[Par95] Parent C. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics for Programs Constructions'95*, number 947 in LNCS. 1995.

[PCCH01] Pouzet M, et al. *Lucid Synchrone v2.0 - Tutorial and Reference manual*. LIP6, April 2001. Available at http://www-spi.lip6.fr/lucid-synchrone/lucid_synchrone_2.0_manual.ps.

[PM93] Paulin-Mohring C. Inductive Definitions in the System Coq - Rules and Properties. In *TLCA*, number 664 in LNCS. 1993.