A clocked denotational semantics for LUCID-SYNCHRONE in COQ*

Sylvain Boulmé   Grégoire Hamon

November 29, 2001

Abstract

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 it formally proved, or more generally to prove program properties.

In this work we are interested in LUCID-SYNCHRONE [PCCH01] (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 here to propose a notion of recursive functions for LS, as a generalization of recursive streams.

The main originality of this work is to apply the *"clocks as types"* paradigm (see [Cas92]) in the design of a formal semantics for a synchronous language. This paradigm consists in expressing static synchronization constraints with a restricted form of dependent types. We show here that such a type system is a subsystem of COQ type system.

*This paper is an extended version of [BH01].

Contents

1 Introduction

Lucid-Synchrone (LS for short) is a high-level language dedicated to the design of reactive systems – systems which have to react continuously with their environment. Such systems are often critical ones (planes control systems, power plant regulators, etc), and require strong temporal properties on programs.

LS ([PCCH01][1]) is a *data-flow synchronous language*, combining the real-time features of Lustre [HCRP91] and the functionality of ML (functions as value, local definitions, etc). Hence, LS manages infinite lists – streams – as primitive values. The *synchronous property* guarantees that programs will only need a bounded memory and bounded time of response during execution (and these bounds can be known *statically*). In LS, this property is checked with a type system, called *clock calculus* [Cas92]. This system mixes implicit polymorphism *à la* ML and a restricted form of dependent types (in sampling operations). Hence, clock inference is decidable. As types in ML, clocks help the programmer in understanding his program as they express *simple static constraints* of synchronization and provide control on the execution of programs. For instance, a polymorphic stream can be computed at any frequency.

We propose here a *shallow* and natural embedding of LS into the Coq proof assistant [BBC+00]. This embedding is shallow in the sense that functional and typed features of LS are implicitely embedded as equivalent features of Coq (see figure 1).

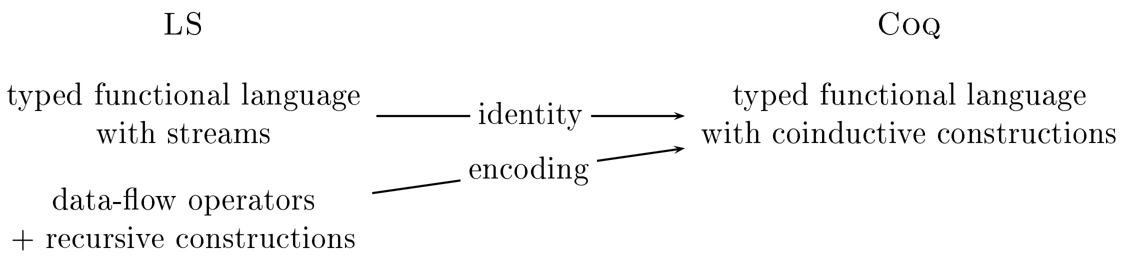

Figure 1: Shallow embedding of LS into Coq

We first define (section 2) a family of type called *sampled streams*. A *sampled stream* is a kind of Coq stream with *"absent elements"*. It is both parametrized by the type of its elements, and by a *clock*. This clock is a stream of booleans, indicating the presence of each of its elements. We then embed (section 3) each data-flow operator of LS as a Coq function over sampled streams. We also provide (section 4) a generic recursive operator for LS recursive constructions. In this embedding, clocks concepts (polymorphism, dependent types) of LS are represented by the equivalent types concepts in Coq. Even clock inference of LS can be emulate using inference features of Coq (see section 5). Hence, we have also embedded implicitely the static semantics of LS into Coq. This gives us a synchrony analyses for free: it is performed by Coq typechecker.

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

This embedding also provides a denotational semantics for LS, very close to the intuitive one. This semantic is *"denotational"* because the execution process of LS programs embedded in COQ is very far from their execution when compiled. It is due to our very naive representation of synchronous functions and especially to the derived encoding of recursive constructions of LS in COQ. But, this semantics is very simple, and we will actually use it to present LS operators to the reader. A simplified presentation of this paper can be found in [BH01].

This semantics seems suitable for proving properties on LS programs: following [Fil99, Par95] approach, it could be a starting point for designing a prover of LS programs into COQ. It could be also the basis to prove the LS compiler: indeed such a proof may consist in showing formally that implementations of LS constructions refine their descriptions given here.

We also use this semantics to propose a notion of recursive functions for LS, as a generalization of recursive streams (see section 4). Recursive functions have seldom been studied in the context of data-flow synchronous programming as they are generally not reactive. However, some of them are still reactive and have recently appeared as a way to express reconfigurable systems [HP00], this work may help in better understanding such functions.

Before presenting our encoding in COQ, we give a short presentation of LUCID-SYNCHRONE, we provide some general remarks about embedding synchronous languages into theorem provers and give a technical presentation of COQ, useful in order to understand the following of the paper.

1.1 Presentation of LUCID-SYNCHRONE

LUCID-SYNCHRONE is a language born to bring synchronous data-flow languages like LUSTRE closer to general functional languages, in particular ML. This was needed for several reasons. First, synchronous languages [Hal93], were in need of more mechanisms of abstraction. Second, to better understand the link between these two families of languages. Third, to study the possible use of this link in a certification or proof context.

1.1.1 A notion of discrete time

Synchrony is an hypothesis of instantaneous reaction of the system – computations must take no time. To handle this hypothesis, synchronous languages introduce a notion of logical and discrete time: time is a sequence of instants, shared among processes (i.e streams). This hypothesis allows a concurrent but deterministic description of systems.
Streams follow this logical time, all streams take their n^{th} value on the n^{th} logical instant.

1.1.2 Clocks to mix time scales

Time is logical and shared among all processes. However, it is often needed to define subsystems following different rates. This is handled by the clock mechanism.
A synchronous (or *sampled*) stream does not have to take a value on each logical instant: it can be slower than the logical time. The *clock* of such a stream is a boolean information

which indicates for each instant if the stream takes a value (it is *present*) or not (it is *absent*). We define:

- the *global* clock of the system as the fastest possible rate (the boolean information which is always true). This clock defines the logical time.
- the clock of a stream is also called its *local* clock. It must be a sub-clock of the global clock, we call "*local instants* of a stream" instants where its local clock is true.

To ensure synchrony (bounded memory and time execution), clocks have to verify some properties when combining streams (for example, when adding two streams they must have the same clock, otherwise a unbounded buffer may be needed at execution), this is done by a *clock calculus*. In LS clocks are inferred and the clock calculus is done by a type system [CP98].

1.1.3 data-flow operators

The language introduces special operators to handle streams and clocks:

- `fby` is an initialized delay. (x `fby` y) takes the value of x on its first instant, then the value of y delayed by one instant. The language also provides separately an initialization operator `->` and a delay operator `pre`.
- `when` is a filtering operator. x `when` y filters x on the clock y, this operator thus introduces the notion of clock.
- `merge` is an over-sampling operators. It takes two streams on complementary clocks and combines them in one stream.

We will present these operators in more details when embedding them into COQ. Classical operators ($+$, $-$, *not*, *if-then-else*) are lifted over streams and applied point-wisely.

1.1.4 Recursive constructions

Two very different kinds of recursions can be seen in the language:

- **recursive streams** – streams can be defined recursively, this recursion has to be not instantaneous (a delay is needed).
- **recursive functions** – not supported in a reactive context. They are traditionally not considered in the context of real time systems.

1.1.5 Static analysis performed by LS

The LS compiler performs various static analysis, we will just list them here. All this analysis will be found, in one form or another, in our COQ encoding of LS.

- *type analysis*: this is done by an ML type system,
- *clock calculus*: this is a kind of ML type system extended with some form of dependent types,
- *initialization analysis*: the `pre` operator introduces uninitialized streams, we have to check that they are in fact initialized,
- *causality analysis*: checks that recursion definitions are not instantaneous (computing the current value of a stream should not depend on itself) [CP01].

1.2 Embedding synchronous languages into theorem provers

As explained in the beginning of this introduction, Coq is well-suited to describe LS, because it is functional, it supports polymorphic and dependent types, streams and coinduction reasoning: an important part of LS can be embedded implicitly in Coq. We give in subsection 1.3 a technical presentation of Coq. We discuss here general choices of our description of LS in Coq, and compare our approach to other works in the area of application of formal methods to synchronous systems.

1.2.1 Embedding LS into Coq

Valid LS programs check some non-trivial static properties. Some of these properties are guaranteed by static analyses performed by the compiler: non-valid programs are rejected at compile-time. 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 consists in stating these properties as axioms or hypothesis in the description, relying on a compiler to check these properties. For instance, recursive streams may be introduced with an axiom stating their existence and the recursive equation they satisfy. 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, it seems still possible to get some help from the compiler. We may imagine special outputs of the static analysis of the compiler giving Coq proofs of some static properties of the program. Thus the work of Coq would be greatly simplified: Coq would only ensure the formal verification of these proofs, by typechecking. Let us note indeed, that the Coq typechecker is rather safe: a light version of this typechecker has been formally proven in Coq itself. [Bar99]

1.2.2 Related works

Synchronous languages have been introduced for software and hardware reliability. Thus, there is an intensive research work about application of formal methods to synchronous systems. Hence a lot of case study have been realized, using many formalisms [KNT00, BCDPV99, CGJ99]. There are also some axiomatizations of synchronous languages [BCDPV99, NBT98].

Our work 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. Indeed, this could be interesting to study the use of clocks in program proofs and specifications. Hence, LS is well-suited for this study, as its clock calculus is already expressed as a dependent type system whereas other synchronous data-flow languages (Lustre, Signal [BLJ91]) use specific analysis.

Actually, embedding clocks as types makes clearer and simpler the description of LS dynamic semantics. Indeed, previous experiments [DCC00, BCDPV99] have shown that it was unexpectedly difficult to describe synchronous language with a coinductive approach. Our experiment shows that such an approach is very simple using clocks as dependent types.

1.3 Technical presentation of COQ

The COQ language is based on the Calculus of Constructions [CH88], enriched with inductive[PM93] and co-inductive[Gim94] type definitions. Thus, it is a typed lambda-calculus, with *type* as first-class values (i.e. types are terms). Hence, roughly speaking (see [BBC+00] for more details), *terms* may be:

- *identifiers*, referring to defined constants or declared variables in the context.
- the *application* of a functional term `M` to a term `N`. It is denoted by "`(M N)`".
- the *abstraction* of a variable `x` of type `A` in a term `b` (i.e. the function whose arguments is `x` and whose result is `b`). It is 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`". If `x` does not occur in `B`, it is also denoted by "`A->B`".

As types are terms, they have themselves types called *sorts*. In the spirit of Curry-Howard isomorphism, types may represent programming datatypes or logical propositions. This is expressed using two special sorts, `Set` as type of datatypes, and `Prop` as type of propositions. Hence, let `B` of type `Prop`, if `A` is of type `Set`, then the functional type "`(x:A)B`" means the logical proposition "for all `x` in `A`, `B`". In the same way, if `A` is now of type `Prop`, then the functional type "`A->B`" means the proposition "`A` implies `B`". In both cases, a proof of such a proposition is a function building a proof `b` of `B` assuming a term `x` of `A`: a term `[x:A]b`.

A product "`(x:A)B`" may correspond to a *polymorphic* type (for instance, if `A` is `Set` and `B` is of type `Set`) or a *dependent* type (for instance, if `A` and `B` are both of type `Set`). Hence, the expressive power of COQ is due to the fact that it unifies many different concepts with a small set of constructions.

Induction & Co-Induction in COQ COQ language allows inductive and coinductive type definitions [PM93, Gim98]. They express different way of handling recursive datatypes *a la* ML in COQ language. Indeed, the first ones allow to handle only finite values with primitive recursion whereas the second ones allow to handle infinite values with co-recursion (a kind of lazy recursion). Primitive recursion corresponds to smallest fixpoints whereas co-recursion corresponds to biggest fixpoints. However, in both cases, computations are always finite: COQ is strongly normalizing.

For instance, `nat` can be defined using the inductive opposite definition. It introduces two constructors of `O` for zero and `S` for successor. Moreover, it defines `nat` as the smallest type generated with only these constructors.

```
Inductive nat : Set :=
  | O : nat
  | S : nat->nat.
```

Values of type `nat` can be destructed using `Cases` construction, a kind of pattern-matching. `Fixpoint` constructions allow to express primitive recursion. Hence, we can define below `funPower` the nth-power of a function, built using primitive recursion over naturals, i.e. argument `n` (see [BBC+00] for more explanations on the syntax).

```
Fixpoint funPower[A:Set; f: A->A; n:nat]: A -> A :=
  [x:A]Cases n of
   |  O => x
   | (S m) => (funPower A f m (f x))
  end.
```

These constructions express also elimination (or recurrence) scheme on naturals. This allows for instance to prove the following commutation lemma about funPower:

```
Lemma funPower_commut:
  (A:Set; f: A->A; n:nat; x:A)(f (funPower A f n x))=(funPower A f (S n) x).
```

Using a very similar syntax, we define opposite `Stream`, the type of infinite lists. It has only one constructor `Cons` for adding an element to a stream. For any set `A`, `(Stream A)` is the biggest type generated with only this constructor.

```
CoInductive Stream[A:Set]: Set :=
 | Cons : A->(Stream A)->(Stream A).
```

As for `nat`, values of type `(Stream A)` can be destructed using `Cases`. For instance, `hd` and `tl`, respectively for head and tail of streams can be defined by:

```
Definition hd := [A:Set; x:(Stream A)]Cases x of (Cons a _) => a end.

Definition tl := [A:Set; x:(Stream A)]Cases x of (Cons _ s) => s end.
```

However recursion over streams (i.e. using `Fixpoint`) is not allowed. But, we can use co-recursion for *building* streams. For instance, constant stream may be defined in this way:

```
CoFixpoint const: (A:Set)A->(Stream A) := [A:Set;a:A](Cons A a (const A a)).
```

`CoFixpoint` is a lazy recursion operator, because the unfolding of such a definition occurs only once inside a "`Cases ... of`". As a recursive call (in a cofixpoint definition) must be protected by at least one constructor, each unfolding of such a definition correspond to a destruction. Moreover recursive call cannot be inside arguments of other functions than constructors. This guarantees that computations always terminate, even on infinite values (see [Gim94]).

In the spirit of the Curry-Howard isomorphism, inductive and coinductive definitions may also be used for defining logical predicate. For instance, bisimulation over streams can be defined by the following coinductive definition:

```
CoInductive EqSt[A:Set]: (Stream A)->(Stream A)->Prop :=
  eqst : (s1,s2:(Stream A))(hd A s1)=(hd A s2)
           -> (EqSt A (tl A s1) (tl A s2))->(EqSt A s1 s2).
```

Thus, a proof that two streams are bisimulable is a couple of proofs: heads of both streams are equal, and their tails are bisimulable. Hence, usually, proof of bisimulation are built using cofixpoints.

Coq and type inference It is tedious to give explicitly all arguments of functions, as some of them can often be deduced from the context: for example, some `A` in the previous definition of `EqSt`. It is possible to ask to Coq typechecker to infer those terms, replacing them by "?" character. However, this inference may fail, because it is not decidable. For instance, the previous definition can also be written:

```
CoInductive EqSt[A:Set]: (Stream A)->(Stream A)->Prop :=
  eqst : (s1,s2:(Stream ?))(hd ? s1)=(hd ? s2)
           -> (EqSt A (tl ? s1) (tl ? s2))->(EqSt ? s1 s2).
```

As, it also tedious to write lots of "?", Coq provides a mode for defining functions with implicit arguments. In this mode, arguments, which other arguments depend on, becomes implicit: during application of functions, Coq tries to infer them as in ML. For example, if we define `hd`, `tl` and `EqSt` in implicit arguments mode, we can directly write:

```
CoInductive EqSt[A:Set]: (Stream A)->(Stream A)->Prop :=
  eqst : (s1,s2:(Stream A))(hd s1)=(hd s2)
           -> (EqSt (tl s1) (tl s2))->(EqSt s1 s2).
```

However, because of undecidability of higher order unification, this inference algorithm may fail. In this case, it is still possible to give explicitly to Coq instances of implicit arguments by prefixing them with "$n!$", where n is the number of the argument. In all the following, definitions are written by default in implicit arguments mode.

Let us remark that types may also be omitted during abstraction. For instance, in the definition of `hd` below, we omit to type `A` with `Set`. However, there is no mechanism of generalization in Coq such as the `let` mechanism of ML. Hence, polymorphic abstraction is explicit:

```
Definition hd := [A; x:(Stream A)]Cases x of (Cons a _) => a end.
```

Equality Equality is a very delicate notion in Coq (see discussion in subsection 5.4). Actually, there is no primitive notion of equality. However, the system uses an internal (and thus implicit for the user) equality during typechecking. This equality is interconvertibility: two terms are interconvertible if they have the same normal form. This equality is of course decidable (thanks to strong normalization). For instance, `(tl (const O))` and `(tl (Cons (S O) (const O)))` are interconvertible.

Moreover, it is possible to reflect this equality into the language, using inductive types. For instance, standard equality (corresponding to `=` infix operator) is defined by the following `eq` predicate:

```
Inductive eq[A:Set; x:A]: A->Prop :=  refl_equal : (eq x x).
```

On closed term, this equality corresponds to interconvertibility. However it is strictly more general. For instance, the following lemma is provable (by case analysis) whereas `x` and `(Cons (hd x) (tl x))` are not interconvertible.

```
Lemma stream_unfold:(A:Set; x:(Stream A))x=(Cons (hd x) (tl x)).
```

Hence a proof of `x=y` is roughly speaking a proof that any closed instances of `x` and `y` satisfying the current context are interconvertible.

Dependent equality The standard equality is not always powerful enough. Indeed, it only allows comparisons of terms that have interconvertible types. As in this development dependent types are used a lot, one need to have an equality over terms that may have types only "provably equal", but not interconvertible. We thus use `eq_dep`, the equality over dependent types, defined in the standard library. In the following, `==<T>` is our notation[2] for the dependent equality (`eq_dep`) over the indexed type `T`. Hence `x==<T>y` implies that `x` is of type `(T a)` for a given `a`, and that `y` is of type `(T b)` for a given `b`, such that `a=b`.

In this development, we use the axiom of the standard library allowing to prove that for `x` and `y` of type `(T a)`, `x==<T>y` implies `x=y`. Indeed, this very desirable property is not directly provable in Coq (nor its contrary) (cf. Streicher [Str93]).

2 Sampled Streams and their equality in Coq

First, given a type `A`, we define the type of *"sampled elements of* `A`*"*, for representing values of `A` on local instants. It is actually a kind of option type, that extends every type `A`, with two values `(None A)` on non-local instants and `(Fail A)` on local instants. The value `(Fail A)` will be used for having uninitialized values in delays and in recursions. The fact that we are in a local instant is given as a parameter of the type `samplElt`.

Hence, we have three possible forms of values for sampled elements of `A`:

- `(None A)` of type `(samplElt A false)`.
- `(Any a)` of type `(samplElt A true)` (assuming `a:A`).
- `(Fail A)` of type `(samplElt A true)`.

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

Then, we just define clocks as streams of booleans. The type of "sampled Streams" is a coinductive type parametrized by `A` and a clock `c`. Its single constructor `sp_cons` takes as arguments a sampled element and a sampled stream.

```
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 extensional equality (or bisimulation) over sampled streams is defined for streams that may not have *interconvertible* clocks, but only *extensionally equal* clocks. Actually, we

[2] Coq provides extensible grammars, allowing the user to modify parsing and pretty-printing, in order to introduce her own notations.

only generalize the definition of the extensional equality over standard streams (`EqSt`), using dependent equality over sampled elements of `A` (`eq_dep`).

```
CoInductive sp_eq: (c1,c2:clock)(samplStr A c1)->(samplStr A c2)->Prop :=
 sp_eq_proof :
   (c1,c2:clock; s1:(samplStr A c1); s2:(samplStr A c2))
        (sp_hd s1)==<(samplElt A)>(sp_hd s2)
             ->(sp_eq (sp_tl s1) (sp_tl s2))->(sp_eq s1 s2).
```

The fact that bisimulable sampled streams have bisimulable clocks is actually an obvious consequence of the above definition. It is also obvious that `sp_eq` is an equivalence relation.

The first interesting result on bisimulation is that we can define a notion of "clock coercion" compatible with bisimulation. Hence, for two bisimulable clocks `c1` and `c2`, we can coerce a stream `s` on `c1` into a stream on `c2` such that `s` and its coercion are bisimulable.

```
CoFixpoint clock_coerce:
  (c1,c2:clock)(EqSt c1 c2)->(samplStr A c1)->(samplStr A c2).

Theorem sp_eq_clock_coerce:
 (c1,c2:clock; H:(EqSt c1 c2); s:(samplStr A c1))(sp_eq s (clock_coerce H s)).
```

This may be used for coercing streams on a clock `c` into streams on `(not (not c))` or conversely. Of course, these two last coercions must be explicit (whereas they are built-in in LS, and thus implicit).

An expected properties of lucid programs is that they are well initialized. We express this property by defining a notion of *"wellformed"* sampled stream. A wellformed sampled stream, is a stream without `Fail` element. Hence, we first define the predicate `is_no_Fail` for sampled elements and then define the predicate `sp_wf` expressing the wellformedness of sampled streams.

```
Definition is_no_Fail: (b:bool)(samplElt A b)->Prop
 := [b;s] Cases s of
     | Fail => False
     | _ => True
    end.

CoInductive sp_wf: (c:clock)(samplStr A c) -> Prop :=
  sp_wf_proof: (c:clock; s:(samplStr A c))
      (is_no_Fail (sp_hd s))->(sp_wf (sp_tl s))->(sp_wf s).
```

3 Lucid-Synchrone data-flow operators in Coq

We now detail the description of LS operators. Point-wise operators and sampling operators are very easy to describe, they just iterate instantaneous operators over sampled elements on the global clock . Delay operators are more complicated because their global behavior depend on the local clock.

3.1 Point-wise operators

We first lift every object of the base language into constant sampled streams. As in LS, these constants are polymorphic on clocks. The `sp_const` operator only iterates the `elt_const` operator on every global instant. At every global instant, if we are on a local instant, `elt_const` encapsulates the constant into a sampled elements using `Any`, else it `elt_const` returns `None`.

```
Variable A: Set.
Variable a: A.

Definition elt_const : (b:bool)(samplElt A b)
  := [b]<(samplElt A)>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))).
```

Of course, it is obvious to show that `sp_const` builds wellformed sampled streams, no `Fail` can occur

We now define `sp_extend` the point-wise application for streams of functions: it 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 the application. `sp_extend` iterates `elt_extend` on every global instant.

At every global instant, if we are not in a local instant, `elt_extend` returns `(None B)`. At every local instant, if either of its arguments is `Fail`, `elt_extend` returns `Fail`. Otherwise, `elt_extend` applies the function element to the argument. Let us remark that in the code below, on the structural case analysis on `x`, the `None` case will never occur. It is actually needed for typechecking structural case analysis. This could be avoided using an *inversion* function, performing specialized case analysis over sampled elements on local instants. However, it is not clear that it would make the code more readable (and the unused case would only be hidden inside the inversion function).

```
Variables A,B: Set.

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

It is easy to show that if both arguments of sp_extend are wellformed, then the result is itself wellformed. It also easy to prove the following lemmas, simplifying expressions involving sp_const and sp_extend:

```
Lemma sp_extend_const: (A,B:Set; c:clock; f:A->B; x:A)
  (sp_eq (sp_extend (sp_const f c) (sp_const x c))
         (sp_const (f x) c)).

Lemma sp_extend_id:(A:Set; c:clock; f: A->A; x:(samplStr A c))
  ((z:A)(f z)=z) -> (sp_eq (sp_extend (sp_const f c) x) x).

Lemma sp_extend_comp:
 (A,B,C:Set; c:clock; f:A->B; g:B->C; h:A->C; x:(samplStr A c))
 ((z:A)(g (f z))=(h z)) ->
  (sp_eq (sp_extend (sp_const g c) (sp_extend (sp_const f c) x))
         (sp_extend (sp_const h c) x)).
```

For instance, if we define sp_not as the point-wise boolean negation (negb), we can use the previous lemmas to prove the following one:

```
Definition sp_not := [c:clock](sp_extend (sp_const negb c)).

Lemma sp_not2_id:(c:clock; lc:(samplStr bool c))
   (sp_eq (sp_not (sp_not lc)) lc).
```

Another interesting application of sp_extend is the definition of the point-wise if-then-else, sp_if:

```
Variable A: Set.

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).
```

3.2 Sampling operators

We first build the sp_on clock constructor. It builds a clock from a sampled boolean stream. This clock is a subclock of the local clock of the argument. This is the type operator introducing dependent types in the LS clock system. Hence, we will be able to sample streams over a dynamically computed clock.

The sp_on operator iterates elt_on on every global instant. On a non-local instant, elt_on return false. On a local instant, if the element is not fail, it is returned, otherwise, the result is true: the error will be diffused in sampling operations (see elt_merge and 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))).
```

We now define the under-sampling operator `sp_when`. It simply copies the input, forgetting elements not on local instant of the output.

```
Variable A: Set.

Definition elt_when:
 (b:bool;o:(samplElt bool b))(samplElt A b)->(samplElt A (elt_on o))
 := [b;o]
  <[b;o:(samplElt bool b)](samplElt A b)->(samplElt A (elt_on o))>Cases o of
    None => [x](None ?)
  | (Any b0) => [x:(samplElt A true)]<(samplElt A)>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 2!(sp_on lc) (elt_when (sp_hd lc) (sp_hd x))
                                  (sp_when (sp_tl lc) (sp_tl x))).
```

We now define the over-sampling operator `sp_merge`. It is a lazy conditional combining streams on complementary clocks. It is an over-sampling operator, since output is on faster clock than the inputs.

```
Variable A: Set.

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

CoFixpoint sp_merge :
 (c:clock; lc:(samplStr bool c))
    (samplStr A (sp_on lc))
      ->(samplStr A (sp_on (sp_not lc)))
        ->(samplStr A c)
 := [c;lc;x;y](sp_cons (elt_merge (sp_hd x) (sp_hd y))
                       (sp_merge (sp_tl x) (sp_tl y))).
```

We have the following relation between sp_if, sp_merge and sp_when:

```
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, and thus provide implicit "memory manipulations". We first consider the *"followed by"* operator (fby) which is an initialized delay. Hence, if x and y are two streams sampled on the same local clock, then x fby y is a stream also sampled on this local clock. Its value is the first element of x at the first local instant, and the previous element of y (for the local clock) at other local instants. Hence, x fby y is a delay of y on the local clock, initialized with the first element of x.

Let us note that the fact x and y are sampled on the same local 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.

3.3.1 Examples in Coq syntax

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

- lifted constants are given between back-quote as `‘(0)‘`. Hence, their clock 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.[3] This operator will be studied in detail 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. Hence, `c` is a polymorphic variable. This must be explicitly expressed to Coq. Here Coq infers the clocks of `‘(0)‘` constant and `‘S‘` constant (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: [4]

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

Finally, we can define functions on streams, whose result is a recursive stream. For instance, `first` which takes a stream `x` and return a constant stream, containing the first element of `x`. `first` is thus polymorphic both on type of element and on clocks. Coq infers for it the type `(A:Set; c:clock)(samplStr A c) -> (samplStr A c)`.

```
Definition first := [A; c; x:(samplStr A c)](rec0 [fst](x fby fst)).
```

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 type `(c:clock)(samplStr bool c) -> (samplStr nat c)`, differs only in their definition by the position of `when` with respect to `fby`. But 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 local instant the number of local instants since the last instant where `lc` was false. Indeed, in the former case, `when` is applied before `fby`: thus, the local clock of `fby` is `(sp_on lc)`. On the contrary, in the latter case, `fby` is on `c`.

[3]On the contrary to ML language, Coq does not provide a mechanism for implicitely generalizing polymorphic variables.

[4]In LS, this definition is compiled in a reactive way, using only a memory size of two integers.

```
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)‘)).
```

Actually, having only `fby` in the language would be a bit constraining, since it requires explicit initialization of delays. Such an initialization may be useless if the initial value is not used. For instance given a boolean stream `lc`, if we want to define `repeat_until` the stream `true` at the first local instant and until `lc` becomes false, and `false` after. We first define `arrow` a synchronization operator: at the first instant `arrow x y` is `x`, and `y` after. Hence, the first value of `y` is always unused.

```
Definition arrow := [A;c; x,y:(samplStr A c)](sp_if (‘true‘ fby ‘false‘) x y).

Definition repeat_until :=
  [c;lc](<c>rec0 [rpu](arrow ‘true‘ (ext (ext ‘andb‘ (sp_not lc))
                                         (‘false‘ fby rpu)))).
```

In the above definition of `repeat_until`, the initialization of the delay of `rpu` with `false` is useless as it is forgotten by the `arrow` operator.

So, we introduce now the uninitialized delay: the unary operator called `pre`. Below `-->` is the infix denotation of arrow.

```
Definition repeat_until :=
  [c;lc](<c>rec0 [rpu]‘true‘ --> (ext (ext ‘andb‘ (sp_not lc)) (pre rpu))).
```

In fact, in our encoding, we have introduced `arrow` and `pre` as primitive, and defined `fby` using `arrow` and `pre`. Of course, having `pre` in the language has a cost, since one must check that every stream is initialized.

3.3.2 Formal descriptions of delay operators

We now define `sp_arrow`, the definition corresponding to `arrow` actually implemented in our encoding.

```
CoFixpoint sp_arrow: (c:clock)(samplStr A c)->(samplStr A c) -> (samplStr A c)
 := [c;x;y](sp_cons (sp_hd x)
                (if (hd c)
                 then (sp_tl y)
                 else (sp_arrow (sp_tl x) (sp_tl y)))).
```

We now 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 use 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.

```
CoFixpoint sp_delay: (c:clock)(samplStr A c)->(samplElt A true)->(samplStr A c):=
 [c]<[c0](samplStr A c0)->(samplElt A true)->(samplStr A c0)>
    Cases c of (Cons hc tc) =>
      <[b](samplStr A (Cons b tc))->(samplElt A true)->(samplStr A (Cons b 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.
```

Now, in order to define `sp_pre` (corresponding to the `pre` operator presented before), we need only to apply `sp_delay` on `Fail`:

```
Definition sp_pre: (c:clock)(samplStr A c)->(samplStr A c) :=
  [c;s](sp_delay s (Fail A)).
```

Then, `sp_fby` (prefix operator corresponding to `fby`) is defined by:

```
Definition sp_fby := [c;x;y](sp_arrow 1!c x (sp_pre y)).
```

It is not hard to prove that the previous definition of `arrow` is equivalent to the new one:

```
Lemma arrow_equiv: (c:clock;x,y:(samplStr A c))(sp_wf x)->(sp_wf y)->
   (sp_eq (sp_arrow x y) (sp_if (`true` fby `false`) x y)).
```

3.3.3 Elimination of the `pre` operator

If the `pre` operator makes the language more flexible, it is a bit hard to handle in proofs, as it introduces `Fail`. In this section, we give a general characterization for eliminating `pre` operator. The idea is that if `pre` is on the right branch of an `arrow`, we may able to transform it into `sp_delay` using any value for initialization. The theorem we want to prove can now be expressed as something like:

> *"assuming* `x:(samplStr B c2)`, `y:(samplStr A c1)`, `a:A`,
> *and* `g: (samplStr A c1)` $\rightarrow$ `(samplStr B c2)`,
> *then* `(sp_arrow x (g (sp_pre y)))` *and*
> `(sp_arrow x (g (sp_delay y (Any a))))` *are bisimulable."*

Actually this property is not checked for any `g`. For instance, if `g` has a memory, the behavior of `g` may be changed by the first value of its argument. Hence, we assume that `g` is a memory less function (as point-wise or sampling operations). Moreover, if `c2` is strictly faster than `c1` then the first element of `(sp_pre y)` will be used by `g` after `sp_arrow` has started to return its right argument. Hence, we also assume that `c2` is a subclock of `c1`.

Now, we define memory less functions as a special type. Indeed, we define them as a kind of stream of function.

```
Variables A,B: Set.

CoInductive mem_less: clock->clock->Set :=
  ml_cons:
   (c1,c2:clock)
     ((samplElt A (hd c1))->(samplElt B (hd c2)))
        ->(mem_less (tl c1) (tl c2))
          ->(mem_less c1 c2).
```

Then, after defining ml_hd and ml_tl as destructors of mem_less, we define application for memory less functions:

```
CoFixpoint mem_less_apply:
 (c1,c2:clock)(mem_less c1 c2)->(samplStr A c1)->(samplStr B c2)
 := [c1,c2;m;x](sp_cons ((ml_hd m) (sp_hd x))
                        (mem_less_apply (ml_tl m) (sp_tl x))).
```

The subclock relation, called clock_inf, is defined coinductively, using leb the non-trivial boolean order such that (leb false true).

```
CoInductive clock_inf: clock -> clock -> Prop :=
  clock_inf_proof:
   (c1,c2:clock)
     (leb (hd c1) (hd c2))->
       (clock_inf (tl c1) (tl c2))->(clock_inf c1 c2).
```

We have of course the following property of sp_on with respect to clock_inf:

```
Lemma sp_on_clock_inf:
  (c:clock; lc:(samplStr bool c))(clock_inf (sp_on lc) c).
```

We are now able to formally express the theorem of elimination of pre:

```
Theorem sp_pre_elimination:
 (c1,c2:clock; g:(mem_less A B c1 c2); x:(samplStr B c2); y:(samplStr A c1);
  a:A)
   (clock_inf c2 c1)->
     (sp_eq
        (sp_arrow x (mem_less_apply g (sp_pre y)))
        (sp_arrow x (mem_less_apply g (sp_delay y (Any a))))).
```

In order to instantiate this theorem for a particular function f, it is enough to build a memory less g, such that f and g are extensionally equal with usual application for f and mem_less_apply for g. Thus, this theorem gives a very simple criteria to show that a function using pre is well-initialized. For example, g can be instantiated by when or extend, but not by merge or fby.

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 well-founded recursion with primitive recursion.

LS provides two notions of recursive constructions: recursive streams and recursive functions of streams. The firsts are always reactive, but not the seconds (we will see some examples below). Thus, we are especially interested in recursive streams. However, we have defined here a unique operator of recursion, from which both can be derived. As some preconditions of this recursion operator can be simplified in the particular case of recursive streams, we first present this case, then, we present the general recursion operation.

4.1 Recursive 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:

```
rec0: (A:Set; c:clock)((samplStr A c)->(samplStr A c))->(samplStr A c).
```

It is build as a smallest fixpoint: by successive approximations using `Fail`. However, applying `rec0` to any function does not always make sense: functional arguments of `rec0` must satisfy some preconditions. And for this class of functions, there is at most one fixpoint (see `rec0_uniq` below): the smallest is also the biggest.

This smallest fixpoint construction follows a diagonal process formally described in section 4.2.3. To help the reader to understand this process, we present it in figure 2 on `Nat` example. For a given clock `c`, let `F = [N:(samplStr nat c)]('(0)' fby (ext 'S' N))` . We have `(rec0 F) = (Nat c)`. On this figure, it appears that for any n, the n-th element of `(rec0 F)` is the n-th element of $(\mathtt{F}^n(\bot(c)))$, where $\bot(c)$ is the constant `Fail` stream, sampled on `c`.

c	true	false	false	true	true	true	false	...
$\bot(c)$	Fail	None	None	Fail	Fail	Fail	None	...
$\mathtt{F}^1(\bot(c))$	0	id						
$\mathtt{F}^2(\bot(c))$	0	None	id					
$\mathtt{F}^3(\bot(c))$	0	None	None	id				
$\mathtt{F}^4(\bot(c))$	0	None	None	1	id			
$\mathtt{F}^5(\bot(c))$	0	None	None	1	2	id		
$\mathtt{F}^6(\bot(c))$	0	None	None	1	2	3	id	
$\mathtt{F}^7(\bot(c))$	0	None	None	1	2	3	None	id

Figure 2: `rec0` diagonal process on `(Nat c)`

This diagonal process will give the expecting result if functional arguments of `rec0` satisfy two properties: length-fail-preservation on the global clock, and wellformedness-increase.

4.1.1 Preconditions of `rec0`

Length-fail-preserving functions (on the global clock) *Length-preservation* on the global clock is a general property of LS programs. Every operator of LS satisfies it. It means that for every LS function, at every global instant, the function takes its one sampled element (possibly `None`) on each its inputs and return one sampled element (possibly `None`) on its output. Thus, this property means two important things about LS. First, the result at a given global instant of a function can not depend on an input at a global instant in the future. Second, LS function evaluation is actually itself a synchronous system on the global clock (see [CP98]).

Actually *length-fail-preserving* also says also that functions are monotonous with respect to the flat order with `Fail` as minimum element. Hence, it also gives to `Fail` the semantics of an error.

Hence, we first define the flat order on sampled elements with `(Fail A)` as minimum:

```
Definition flat_ord :=[b1,b2:bool; x:(samplElt A b1); y:(samplElt A b2)]
  Cases x of
   | Fail => True
   | _ => x==<(samplElt A)>y
  end.
```

Then, using `sp_nth` that return the $\mathtt{n}+1$-th element of sampled stream on the global clock, we define `sp_inf`. This relation extend `flat_order` for the $\mathtt{n}+1$-first elements of two streams:

```
Definition sp_inf:=
 [n:nat; c1,c2:clock; s1:(samplStr A c1); s2:(samplStr A c2)]
   (m:nat)(le m n)->(flat_ord (sp_nth m s1) (sp_nth m s2)).
```

Now we say informally that a function is length preserving if for every `n`, it is monotonous for the order `(sp_inf n)`. For instance, the lemma showing that `sp_merge` is length preserving is the following statement:

```
Lemma sp_merge_lfp:
  (n:nat; c1,c2:clock; lc1:(samplStr bool c1); lc2: (samplStr bool c2);
   s1:(samplStr A (sp_on lc1)); s2:(samplStr A (sp_on lc2));
    t1:(samplStr A (sp_on (sp_not lc1))); t2:(samplStr A (sp_on (sp_not lc2))))
   (sp_inf n lc1 lc2) ->
     (sp_inf n s1 s2)->
       (sp_inf n t1 t2)->
         (sp_inf n (sp_merge s1 t1) (sp_merge s2 t2)).
```

Wellformedness-increase A function F of type `(samplStr A c) -> (samplStr A c)` is *wellformedness-increasing* if: if for every n, its input has no fail element until the n-th instant on the global clock, then its output has no fail element until the $n+1$-th instant on the global clock. Thus, by iterating infinitely on F, we build a wellformed stream.

Wellformedness-increase is related to causality. *Causality* guarantees that during construction of a recursive stream, an element of the stream does not depend on itself: it prevents instantaneous recursion. Thus, a function is *causal* if each element of its output only depend on strictly anterior elements of the inputs.

Actually, wellformedness-increase is stronger than causality. Indeed the recursive definition below is not wellformedness-increasing, because not well-initialized, (see discussion about elimination of pre page 18), whereas it is causal. But, both notions coincide for well-initialized functions.

```
Definition Nat_on_badinit :=
  [c; lc](<c>rec0 [N](`(0)` --> (merge lc (pre (N when lc)) `(0)`))).
```

Thus, we introduce a notion of *"partial wellformedness" on the n-th first instant of the global clock*, indicating if a stream has no fail element until the *nt*-th instant of the global clock. Being a property on the global clock, it is easy to handle in proofs.

```
Fixpoint glob_nfstwf [n:nat]:(c:clock)(samplStr A c)->Prop :=
  [c; s]
    Cases n of
    | O => True
    | (S m) => (is_no_Fail (sp_hd s)) ∧ (glob_nfstwf m (sp_tl s))
    end.
```

Then, we can show that every LS operators (except fby and pre) preserves partial wellformedness. For instance, for sp_merge, we have the following lemma:

```
Lemma sp_merge_glob_nfstwf:
  (n:nat; c:clock;  lc:(samplStr bool c); s:(samplStr A (sp_on lc));
    t:(samplStr A (sp_on (sp_not lc))))
     (glob_nfstwf n lc)->
       (glob_nfstwf n s)->
        (glob_nfstwf n t)->
         (glob_nfstwf n (sp_merge s t)).
```

Of course, sp_fby is a special operator that increases partial wellformedness for its second argument.

```
Lemma sp_fby_glob_nfstwf:
  (n:nat; c:clock; x,y:(samplStr A c))
    (glob_nfstwf (S n) x)->
      (glob_nfstwf n y)->
       (glob_nfstwf (S n) (sp_fby x y)).
```

Correctness of `rec0` We are now able to set formally the precondition of rec0:

```
Variable A: Set.
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)).
```

Under these preconditions, `(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).
```

4.1.2 Using `rec0` operators

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 wellformedness.

Mutually recursive streams We now show how to use `rec0` for expressing mutual recursive streams of LS: we put each mutual recursive occurrence as a parameter of a recursive stream, linking mutual recursion at hand. Hence, expressing mutual occurrence need some syntactic treatment, but it seems automatable.

For instance, suppose we want to build a function `counter` parametrized by a constant stream of integer `n` and returning a stream of boolean equal to true every `n` instants and `false` at other instants. To implement this, we use a stream `acc` that counts the executed instants, and a stream of boolean `count` (which is the result of `counter`) that tests equality of `acc` with `n`. Actually, `acc` also depends on `count` because `acc` is reseted each time `count` is true.

We can encode this in the following section of Coq. We declare a clock `c` and a stream of natural `n` on `c`. We need the hypothesis that `n` is wellformed in the proof that `counter` satisfies the preconditions of `rec0`. Then, one first defines `acc_aux` which is the recursive definition of `acc` explained before, parametrized by `count`. Then, the result of `counter` is defined as a recursive stream `count` using `(acc_aux count)` instead of the mutual recursive occurrence to `acc`. Below, `beq_nat` is the scalar equality over natural (returning a boolean).

```
Variable c: clock.

Variable n: (samplStr nat c).

Hypothesis n_wf:(sp_wf n).

Definition acc_aux:=[count]
 (<c>rec0 [acc] `(1)` fby (sp_if count `(1)` (ext `S` acc))).

Definition counter:=
 (rec0 [count](ext (ext `beq_nat` n) (acc_aux count))).
```

Here we are not interested in defining `acc`. But we could define it by applying `acc_aux` to `counter`.

Proving that `counter` satisfies the preconditions of `rec0` actually requires to show that `rec0` has itself a good behavior with respect to length-fail-preservation and partial well-formedness. Here are the properties we have proved about `rec0`:

```
rec0_lfp
     : (A:Set; c:clock; F1,F2:((samplStr A c)->(samplStr A c)); n:nat)
        ((s1,s2:(samplStr A c))
          (sp_inf n s1 s2)->(sp_inf n (F1 s1) (F2 s2)))
        ->(sp_inf n (rec0 F1) (rec0 F2))

rec0_glob_nfstwf
     : (A:Set; c:clock; F0:((samplStr A c)->(samplStr A c)); n:nat)
        ((m:nat; s:(samplStr A c))
          (lt m n)->(glob_nfstwf m s)->(glob_nfstwf (S m) (F0 s)))
        ->((s1,s2:(samplStr A c); m:nat)
            (sp_inf m s1 s2)->(sp_inf m (F0 s1) (F0 s2)))
        ->(glob_nfstwf n (rec0 F0))
```

This has allowed us to prove in Coq that the recursive definition of `counter` is correct.

4.2 A generic recursive operator

We introduce now the generic operator `rec1` from which `rec0` is derived. 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. Particular instantiation of `B` can be:

1. unit type. Actually `rec0` is build with this particular instance of `rec1`.
2. tuple of streams. It allows then to build monorphic (on clocks) recursive function of streams.
3. a dependent pair clock/stream. It allows then to build polymorphic (on clocks) recursive function of streams.
4. scalar types. In a future extension of LS, there may be a notion of static constant (ie scalar), allowing for instance to define a notion of buffer parametrized by the size of the buffer. Then `rec1` could be used to define recursive function on a static parameter.

Actually, all the previous properties of `rec0` are specialized properties of `rec1`. Before presenting them, we advocate a bit more `rec1` on some examples, as suggested above.

4.2.1 Examples of recursive functions

As first example, `rec1` allows building some kind of mutual recursion. For instance the above `counter` can be written in `counter1`, using the following way. The stream `acc` is now a parameter of a function `counter1_aux`. And `counter` is only an application of `counter1_aux` initializing `acc` with `‘(1)‘`. Hence, we are very close to usual functional programming. The function `counter1_aux` use a recursive function `count`, which is a particular instantiation of `rec1` using `(samplStr nat c)` as a special instantiation of `B`, and the constant function `[_]c` for `C`.

```
Definition counter1_aux :=
 [c:clock][n]
   (rec1 3![_]c
        [count: (samplStr ? c)->(samplStr ? c); acc]
          (sp_if (ext (ext ‘beq_nat‘ acc) n)
                 (‘true‘ fby (count ‘(1)‘))
                 (‘false‘ fby (count (ext ‘S‘ acc))))).

Definition counter1 :(c:clock)(samplStr nat c)->(samplStr bool c)
 := [c; n](counter1_aux n ‘(1)‘).
```

A more interesting example (maybe THE example) is `reset`. The `reset` operation is a function application, resetting the function being applied every time a given condition lc is true. More precisely, it resets all (bound and free) variables of a function f with their initial value, during the application of f. But, all f cannot be reset this way. f may contain free variables used by other part of the program, it could be very dangerous to reset these variables. Thus, one must exhibit a (simple) condition guaranteeing that resetting variables of f will not affect other part of the program. Good candidates for such variables are actually bound variables and free variables polymorphic on clocks. Thus, a simple static condition on f is to require it to be polymorphic on clocks. Thus `reset` has the following type:

```
reset: (A,B:Set)
        ((c:clock)(samplStr A c)->(samplStr B c))
         ->(c:clock)(samplStr A c)->(samplStr bool c)->(samplStr B c)
```

Actually as f is polymorphic, resetting it is just applying it to a different clock. For instance let recall `counter`, previously defined, of type:

```
counter: (c:clock)(samplStr nat c)->(samplStr bool c)
```

Each application of `counter` to a different clock, resets recursive streams `acc` and `count`. Hence `reset` is quite simple to write using polymorphic recursion and the previous constant `repeat_until`[5]. Indeed, under the following assumptions,

```
Variable A,B: Set.

Variable f: (c:clock)(samplStr A c)->(samplStr B c).

Hypothesis f_wf: (c:clock; x:(samplStr A c))(sp_wf x)->(sp_wf (f x)).
```

[5]We recall that `repeat_until` defined page 17 takes a stream `lc` of booleans, and returns a stream of booleans, true at the first instant and until `lc` becomes true.

we have proven that `reset` satisfies the following lemma (using `rec1_inv` a generalization of `rec0_inv`):

```
reset_inv:
 (c:clock; x:(samplStr A c); lc:(samplStr bool c))
   (sp_wf x)
    ->(sp_wf lc)
      -> let c0=(repeat_until lc)
         in
           (sp_eq (reset f x lc)
                  (merge c0
                         (f (x when c0))
                         (reset f (x when (sp_not c0)) (lc when (sp_not c0))))).
```

This lemma says that `reset` makes a new application of `f` to `x` every times `lc` is true.

In order to build such a `reset`, we instantiate `B` using the type `{ c:clock & ((samplStr A c)*(samplStr bool c)) }` where `{ & }` is the type operator of dependent pairs (or existential type) and `*` the type operator of non-dependent pairs. Then `C` is instantiated by `ProjS1` the first projection for dependent pair. In the following code `ProjS2` is the second projection for dependent pairs, and `Dpair` is the constructor of dependent pairs. `Dpair` takes as arguments the predicate on which the dependent pair is built, and the second projection of the pair: the first projection is here inferred from the second one.

```
Definition reset := [c; x: (samplStr ? c);lc]
   (rec1 3!ProjS1
       [reset; p:{ c:clock & ((samplStr A c)*(samplStr bool c)) }]
       let (x,lc)=(ProjS2 p)
       in
         let c0 = (repeat_until lc)
         in
           (merge c0 (f (x when c0))
                    (reset (Dpair [c:clock](samplStr A c)*(samplStr bool c)
                                 ((x when (sp_not c0)),(lc when (sp_not c0))))))
    (Dpair [c:clock](samplStr A c)*(samplStr bool c) (x,lc))).
```

Another interesting example is Eratosthenes sieve for computing the sequence of prime number. The kernel of the algorithm can be coded as a function `sieve` given below of type `(c:clock)(samplStr nat c) -> (samplStr bool c)` such that if `x` is a subsequence (or sub-stream) of natural number containing only natural numbers not multiple of any number (except 0 and 1) strictly inferior to the first of `x`, then `(sieve n)` is the sequence of boolean indicating which element of `x` are prime number. Let us remark that with such a precondition on `x`, the first element of `x` must be itself a prime number. Thus, using `sieve`, the sequence of prime number can be given by:

```
Definition primes := [c]
  let nat_from_2 = (ext `S` (ext `S` (Nat c)))
  in (nat_from_2 when (sieve nat_from_2)).
```

Now, `sieve` can be defined in a way close to `reset`, in order to satisfy the following recursive equation `sieve_inv`. First, we define `filter` a function taking a stream of natural `x` and returning a stream of boolean indicating which element of `x` are multiple (using the Euclidean remainder function, called `mod`) of the first element of `x`. Hence, this stream of boolean is true at the first instant. Then `sieve` satisfies an equation very similar to the one of `reset`.

```
Definition filter:=[c; x:(samplStr nat c)]
 (ext (ext ‘beq_nat‘
           (ext (ext ‘mod‘ x (first x))))
      ‘(0)‘).

Lemma sieve_inv: (c:clock;  x:(samplStr nat c))
   (sp_wf x)->
      (sp_eq (sieve x)
             (merge (filter x)
                    (‘true‘ fby ‘false‘)
                    (sieve (x when (sp_not (filter x)))))).
```

Actually, `reset` can be compiled in a reactive way because when the recursive call is made, no further computation will ever be needed in the caller (this comes from the fact that when `(repeat_until lc)` becomes false for the first time, it will never become true again). This can be related to tail-recursion in classical languages, and as an analogy we call such functions *tail-recursives.*

On the contrary, `sieve` is not tail recursive and can not actually be compiled into a reactive program, because it needs an unbounded memory size. The recursive call here creates a new process which is concurrently executed, whereas the reset is a very particular case where the new process can indeed replace the caller.

This difference of behavior between the two functions, comes from a very particular semantic property of `repeat_until`. Thus, it seems difficult to distinguish between `sieve` or `reset` syntactically.

4.2.2 Preconditions

We now generalize preconditions of `rec0` to `rec1`, such that previous examples satisfie these preconditions. We recall that the argument of `rec1` is a function `F1` transforming a function of type `(x:B)(samplStr A (C x))` into a function of the same type.

Generalizing length-fail-preservation is quite direct: we require that `F1` preserves length-fail-preserving functions. Generalizing wellformedness-increase is less obvious: in `reset` or in `sieve`, we have polymorphic recursion on clock, and thus recursive call may not be on the same local clock than the current call. Thus, our definition of wellformedness-increase as “increasing partial wellformedness on the global clock” is too much restrictive, because recursive call may happen “after” the current call on the global clock. A notion of “increasing partial wellformedness on local clocks” is better suited.

We now define *“partial wellformedness on local clock”.* This means that we define a predicate `loc_nfstwf: (n:nat; c:clock) (samplStr A c) -> Prop` such that a term of type `(loc_nfstwf n s)` is a proof that sampled stream `s` has no fail element until the `n`-th local instant of `s`. And as we want to express properties like “`(loc_nfstwf n s)` implies (loc_nfstwf (S n) (f s))", we need to be able to have formal reasoning about local instants (whatever their "real" existence). Thus, we take the convention, that if s has only a finite number of local instant, and if s has no fail element on all its local instant, then (loc_nfstwf n s) is true for every n, even if n is bigger than the number of local instant. Actually, it means that we don't care whether s has a finite number of local instant: if s is wellformed, it has no fail element until every n-th local instant. Hence, loc_nfstwf is defined by the following coinductive type:

```
CoInductive loc_nfstwf: (n:nat; c:clock)(samplStr A c)->Prop :=
 | loc_nfstwf_O:
    (c:clock; s:(samplStr A c))(loc_nfstwf O s)
 | loc_nfstwf_false:
    (n:nat; c:clock; s:(samplStr A c))
      (hd c)=false -> (loc_nfstwf n (sp_tl s))->(loc_nfstwf n s)
 | loc_nfstwf_true:
    (n:nat; c:clock; s:(samplStr A c))
      (hd c)=true ->
         (is_no_Fail (sp_hd s))
           ->(loc_nfstwf n (sp_tl s))->(loc_nfstwf (S n) s).
```

Hence a proof of (loc_nfstwf n s) is finite when s has an infinite number of local instant, but may be infinite (ie built using a cofixpoint), waiting for local instants that will never occur.

The idea is now to express wellformedness-increase using loc_nfstwf instead of glob_nfstwf. For rec0, both definitions of wellformedness-increase are equivalent. Indeed we have proven the following equivalence:

```
Variable A,B:Set.

Theorem rec0_nfstwf_inc_glob2loc:
 (c: clock; f: (samplStr A c) -> (samplStr B c))
  ((n:nat; s:(samplStr A c))
    (glob_nfstwf n s)->(glob_nfstwf (S n) (f s)))
  ->(n:nat; s:(samplStr A c))(loc_nfstwf n s)->(loc_nfstwf (S n) (f s)).

Theorem rec0_nfstwf_inc_loc2glob:
 (c: clock; f: (samplStr A c) -> (samplStr B c))
  ((n:nat; s:(samplStr A c))
    (loc_nfstwf n s)->(loc_nfstwf (S n) (f s)))
  ->(n:nat; s:(samplStr A c))(glob_nfstwf n s)->(glob_nfstwf (S n) (f s)).
```

However, loc_nfstwf is quite harder to handle. For instance, expressing preservation of partial wellformedness on local clock is harder for sampling operators, because we need now to express synchronization constraint between several kind of local instants. For instance, for sp_merge, this property is the following :

```
Lemma sp_merge_loc_nfstwf:
  (n:nat; c:clock;  lc:(samplStr bool c); s:(samplStr A (sp_on lc));
    t:(samplStr A (sp_on (sp_not lc))))
     (loc_nfstwf n lc)->
       ((m:nat)(locle m n (sp_on lc) c)->(loc_nfstwf m s))
       ->((m:nat)(locle m n (sp_on (sp_not lc)) c) -> (loc_nfstwf m t))
         -> (loc_nfstwf n (sp_merge s t)).
```

In the above statement, `locle` is a coinductive predicate of type `nat->nat->clock->clock->Prop` such that `(locle n1 n2 c1 c2)` implies that the `n1`-th local instant of `c1` will happen on the global clock before the `n2`-th local instant of `c2`.

We are now able to express preconditions for `rec1`.

```
Variable B: Set.

Variable C: B -> clock.

Variable F1: ((x:B)(samplStr A (C x)))  -> (x:B)(samplStr A (C x)).

Variable P: B -> Prop.

Hypothesis F1_lfp:
  (s1,s2:(x:B)(samplStr A (C x)); n:nat)
    ((x:B)(P x)->(sp_inf n (s1 x) (s2 x)))
     ->(x:B)(P x)->(sp_inf n (F1 s1 x) (F1 s2 x)).

Hypothesis F1_nfstwf_inc:
  (n:nat; s:(x:B)(samplStr A (C x)))
    ((x:B)(P x)->(loc_nfstwf n (s x)))
      ->(x:B)(P x)->(loc_nfstwf (S n) (F1 s x)).
```

Under these assumptions `(rec1 F1)` is of type `(x:B)(samplStr A (C x))`. `P` is precondition of `(rec1 F1)` allowing its domain of application to element of `B` satisfying `P`. Hence, `(rec1 F1)` satisfies :

```
rec1_wf: (x:B)(P x)->(sp_wf (rec1 F1 x)).

rec1_inv: (x:B)(P x)->(sp_eq (rec1 F1 x) (F1 (rec1 F1) x)).

rec1_uniq: (r1:((x:B)(samplStr A (C x))))
           ((x:B)(P x)->(sp_eq (r1 x) (F1 r1 x)))
           ->(x:B)(P x)->(sp_eq (rec1 F1 x) (r1 x)).
```

It is thus now easy to see how to specialize `rec1` into `rec0`.

Let us remark, that the proof that `reset` (respectively `sieve`) satisfies the preconditions of `rec1` is not very hard. But, it uses fundamentally the fact that the first element of `repeat_until` (respectively `filter`) is always true. Indeed, without this information, we would not be able to prove the assumption of wellformedness-increase: for instance, if the first element of `repeat_until` was always false, the recursive definition would lead to an infinite loop at the first local instant.

4.2.3 Formal encoding

We describe now how `rec1` is encoded, and how its fundamental properties are proved. Following a traditional approach to fixpoints, we consider the family of `n`-th power of `F1`, denoted `(funPower F1 n)`, applied to the constant function `[z:B](undef A (C z))`, which produces a constant stream of `Fail` on the local clock `(C z)` for any `z`. It is very easy to prove the following properties, using assumptions on `F1`:

```
Lemma powerF1_loc_nfstwf:
 (n:nat;x:B)(P x)->
  (loc_nfstwf n (funPower F1 n [z:B](undef A (C z)) x)).

Lemma powerF1_lfp:
  (n,m:nat; s1,s2:(x:B)(samplStr A (C x)))
    ((x:B)(P x)->(sp_inf m (s1 x) (s2 x)))->
      (x:B)(P x)->(sp_inf m (funPower F1 n s1 x) (funPower F1 n s2 x)).

Lemma powerF1_increases:
  (n,m:nat; x:B)(P x)->
    (sp_inf m (funPower F1 n [z:B](undef A (C z)) x)
              (funPower F1 (S n) [z:B](undef A (C z)) x)).
```

From `powerF1_increases` and `powerF1_loc_nfstwf`, it is easy to deduce relations between `n` + 1-th element of `m`-th power of `F1`.

```
Lemma powerF1_nth: (n,m:nat; x:B)(le n m)->(P x)->
  (sp_nth n (funPower F1 (S m) [z:B](undef A (C z)) x))
   =(sp_nth n (funPower F1 (S (S m)) [z:B](undef A (C z)) x)).
```

Then, we reason by analysis. Assuming we have a fixpoint `r1` of `F1`, we describe it extensionally.

```
Variable r1: (x:B)(samplStr A (C x)).
Hypothesis r1_inv: (x:B)(P x)->(sp_eq (r1 x) (F1 r1 x)).

Theorem r1_ext: (n:nat; x:B)(P x)->
 (sp_nth n (r1 x))=(sp_nth n (funPower F1 (S n) [z:B](undef A (C z)) x)).
```

Hence, this theorem says that a fixpoint for `F1` is unique (if it exists), and it gives us a good candidate to implement `rec1`: it formalizes what has been explained in figure 2 page 20.

Before defining `rec1`, we first define `rec1_tmp` to compute the `n`-th tail of `rec1`. Below, `clock_nth_tl` is the `n`-th tail power function.

```
CoFixpoint rec1_tmp:
 (n:nat)((x:B)(samplStr A (C x)))->(x:B)(samplStr A (clock_nth_tl n (C x)))
 := [n; s]
     let s'=(F1 s) in [x](sp_cons (sp_nth n (s' x)) (rec1_tmp (S n) s' x)).

Definition rec1:(x:B)(samplStr A (C x))
  :=[y:B](rec1_tmp O [x:B](undef A (C x)) y).
```

Then, we prove that `rec1` is a fixpoint. Unicity is deduced from `r1_ext`.

5 Concluding remarks about this experiment

This experiment aims to use the "clocks as types" paradigm in the design of a formal semantics for synchronous languages. From our point of view, it is a success: such an approach is promising. We give now some remarks that may help in reusing this work to build a more practical tool. Subsection 5.4 is dedicated to a more meta-theorical aspect: we explain why type interconvertibility of LS corresponds to type interconvertibility in Coq.

In subsection 5.1, we discuss the possibility to use Coq for infering clocks of LS programs. In subsection 5.2, we discuss ways for automating proofs of length-fail-preserving and wellformedness-increasing properties. In subsection 5.3, we show that our semantics is not quite complete with respect to LS and show how to fix this. It is the opportunity to discuss if the introduction of `Fail` was a good idea.

5.1 Clock inference

In a practical tool, we could imagine having a dedicated tactic infering Coq types of LS programs, as does the LS compiler. But it may be still interesting to keep implicit syntax in Coq for LS programs, as it may help handling this programs in Coq.

As shown on many examples in this article, a big part of LS clock inference can be performed by Coq, using implicit arguments mode and other syntactic facilities. Actually, Coq user have to manage three tasks for making inference work:

- *polymorphic generalization*: "let-in" generalization of ML.
- *instantiate polymorphic constant*. This instantiation can "?" in most cases. However, this instantiation is sometimes a way to bound polymorphic variable for generalization as in `Nat_on_half` and `over` below.
- *make explicit some implicit arguments*: those of type "`(samplStr bool ?)`". Indeed, sampled elements of booleans are always explicit in LS, whereas they can be implicit in Coq. We detail this point below, on `hold` example.

To illustrate the points above, we introduce some new examples. First, we define `half` the polymorphic constant alternating `true` with `false` (or the half frequency sampler). Then we define a new polymorphic constant `Nat_on_half`, undersampling `Nat` on `half`. As we have to bind the variable clock `c`, we have the choice between binding it to `Nat` or to `half` as shown by `Nat_on_half` or `Nat_on_half'`

```
Definition half := [c](<c>rec0 [half] ‘true‘ fby (sp_not half)).

Definition Nat_on_half := [c]((Nat ?) when (half c)).
Definition Nat_on_half’ := [c]((Nat c) when (half ?)).
```

We now introduce a generic function for oversampling a stream `x`, inserting the last value of `x` when necessary or `ydef` if `x` has not yet started. As expected, Coq infers it the type below.

```
Definition hold :=
  [A; c; ydef:(samplStr A c); lc; x]
    (rec0 [y](merge lc x ((ydef fby y) when (sp_not lc)))).

hold : (A:Set; c:clock)
        (samplStr A c)
         ->(lc:(samplStr bool c))(samplStr A (sp_on lc))->(samplStr A c)
```

However, this function does not behave exactly as its equivalent in LS, because implicit mode of Coq makes here argument `lc` implicit. Hence, if we want to define `over` below, as we would do in LS, we have to make this argument explicit. Hence, in the definition of `over`, we need to use "n!" notation in order to explicit the arguments. An alternative is to define a macro `hold'` that makes this argument explicit, and uses it as in the definition of `over'` below. Actually `merge` is defined from `sp_merge` using such a macro.

```
Definition over := [c;x](hold '(1)' 4!(half c) x).

Syntactic Definition hold':=[ydef;lc;x](hold ydef 4!lc x).
Definition over':=[c;x](hold' '(1)' (half c) x).
```

Of course, as expected, `over` is of type:

```
(c:clock)(samplStr nat (sp_on (half c)))->(samplStr nat c)
```

5.2 Proof automation

Currently proof automation for length-fail-preserving and wellformedness-increasing properties is not very powerful: we only use *à la prolog* resolution on a good basis of lemmas. A beautiful way to make things more automatic is to describe formally the abstract syntax of LS in Coq, and to reflect causality and initialization analysis in Coq: it would both prove these analyses in Coq, and allow their use in proving precondition of `rec0`. Let us remark that a property such as length-fail-preserving that is (in principle) satisfied for every LS function, would not need to be proved anymore by the user: there would just be an internal proof stating that every LS function is length-fail-preserving. Checking that a Coq term is an LS function would just be Coq typechecking.

5.3 Was `Fail` a good idea ?

Adding a `Fail` value in the semantics allows to deal both with initialization and causality in a very simple way. Hence, there is a single (and quite simple) property to guarantee both of them in recursive constructions.

However, our semantics is not quite complete: some valid (but strange ?) LS programs are not representable in our semantics. This incompleteness is due to the fact that our length-fail-preserving lemma for `sp_fby` is too weak:

```
Lemma sp_fby_lfp:
  (n:nat; c:clock; x1,x2,y1,y2:(samplStr A c))
    (sp_inf n x1 x2)->(sp_inf n y1 y2)->(sp_inf n (sp_fby x1 y1) (sp_fby x2 y2)).
```

Actually, it should be:

```
  (n:nat; c1,c2:clock; x1,y1:(samplStr A c1); y1,y2:(samplStr A c2))
    (sp_inf n x1 x2)->(sp_inf n y1 y2)->(sp_inf n (sp_fby x1 y1) (sp_fby x2 y2)).
```

But, we have notice this mistake too late for fixing it. Indeed, this last version of the lemma is not provable, because our notion of "length-fail-preserving" is itself too weak. Hence, here is an example of valid LS program that does not meet the length-fail-preserving precondition of `rec0`.

```
Definition expb :=
  [c](<c>rec0 [expb](`true` fby (merge expb (`false` fby `true`)
                                            (`true` fby `false`)))).
```

Indeed, under the assumption `(sp_inf n expb1 expb2)`, the following statement is false:

```
(sp_inf n
  (sp_fby (sp_const false (sp_on expb1)) (sp_const true (sp_on expb1)))
  (sp_fby (sp_const false (sp_on expb2)) (sp_const true (sp_on expb2))))
```

If at the first instant `expb1` is `Fail` and `expb2` is `(Any false)`, first `fby` is `(Any false)` and second `fby` is `None`. Then at the next instant, if both `expb1` and `expb2` are `(Any true)`, then first `fby` is `(Any true)`, and second is `(Any false)`.

Nevertheless, it seems possible to modify our notion of length-fail-preserving, in order to avoid these problems. These modifications would consist of the two following changes in the previous definition:

- changing `flat_order` such that `None` becomes a maximum element (keeping `Fail` as a minimum element). Hence, `sp_on` becomes monotonous :
  `(flat_order` lc_1 lc_2`)` $\rightarrow$ `(leb (sp_on` lc_2`) (sp_on` lc_1`))`
- requiring `(sp_inf` n s_1 s_2`)` implies `(glob_nfstwf` n s_1`)` (ie that only the $n+1$-th element of s_1 can be `Fail`).

With this new notion of length-fail-preserving, recursive operators may be still correct, and `fby` becomes fully length-fail-preserving. But, this has to be formally checked.

Actually, a simpler approach for defining recursive operators is to consider a *fail sampled stream* (the $\perp(c)$ stream of figure 2 page 20) instead of a *fail sampled element*. Let us remark indeed, that `Fail` is not necessary to define `pre`: we can always initialize the delay of `x` with the first element of `x`, as in the following definition.

```
CoFixpoint sp_init_pre : (c:clock)(samplStr A c) -> (samplStr A c)
 := [c]<[c0](samplStr A c0)->(samplStr A c0)>
    Cases c of (Cons hc tc) =>
      <[b](samplStr A (Cons b tc))->(samplStr A (Cons b tc))>
         if hc
         then
            [x](sp_delay x (sp_hd x))
         else
            [x](sp_cons (None A) (sp_init_pre (sp_tl x)))
    end.
```

However, we loose here the initialization analyses.

5.4 Clock interconvertibility

Interconvertibility of dependent types is delicate. Indeed, dependent types contain expressions of the language, and thus checking the interconvertibility of types requires to evaluate these expressions during typechecking. Static typechecking requires thus the termination of expression evaluation. We explain here how this applies to LS programs, and we show that type interconvertibility of LS is coherent with our description in COQ. But before, we detail a bit the notion of interconvertibility of dependent types.

In a programming language, we may like to express length constraints on arrays in the type system. This is done for instance in ADA, where type of arrays are parametrized by the length of the array. Let us here suppose a type `list_n` as lists of fixed length. And let us suppose an operation of concatenation `append_n` on these lists. In a COQ-like syntax, we would have:

```
list_n: nat -> Set.
append_n:(n1,n2:nat)(list_n n1)->(list_n n2)->(list_n (plus n1 n2)).
```

Hence, in order to have generic operations on `list_n` such `append_n`, we may need to apply, inside a type expression, `list_n` to any expression of type `nat`. The next question is what is equality of types during typechecking. For instance, it could be very useful to have:

1. “`(list_n (plus (2) (2)))`” equals to “`(list_n (plus (4)))`”
2. or for any variable `n`, “`(list_n (plus n (0)))`” equals to “`(list_n n)`”.

In ADA, the problem is solved with dynamic typechecking. At compile time, every expressions in types are equals (hence `(list_n (5))` equals to `(list_n (4))`). And, real constraints verification is performed at execution-time: an error is raised at execution in case of type inconsistency.

On the contrary, in dependent type systems, we want the typechecking to be completely static. To check interconvertibility of types, the system evaluates expressions and checks whether they have the same normal form. Hence, evaluation of expressions must terminate.

Hence, in COQ, the first equality above is true, whereas the second depends on the implementation of `plus`. Some implementations of `plus` may use a recursion on the second argument, and thus will satifisfies this particular interconvertibility rule. But in general, such equalities (with free variables) do not correspond to interconvertibility rules. Indeed, this kind of equalities is not decidable and requires a proof from the user by induction. Such proofs provide an explicit coercion function between the corresponding types.

In our embedding, LS expressions are COQ terms built using LS operators. Because all these operators are cofixpoints, and because of lazy evaluation, there is no evaluation on “pure” LS expressions in COQ. Evaluation on LS expressions is done using destructors of sampled streams (for instance `sp_hd` or `sp_tl`) which are not part of LS expressions. Hence, if we restrict COQ type system to LS expression, there is no evaluation. And thus, type interconvertibility is syntactic equality. This is exactly the notion of interconvertibility of LS.

6 Conclusion

We have shown here a simple denotational semantics of Lucid-Synchrone, such that:

- streams of LS are represented by a kind of streams (sampled streams) in Coq.
- functions of LS are represented by functions in Coq.
- clocks concepts (polymorphism, dependent types) of LS are represented by the equivalent concepts in Coq. Even clock inference of LS can be emulate using inference features of Coq.

Thus, we have established formally an embedding from LS into Coq preserving semantics of strongly typed functional languages with streams. This result strengthens both LS and Coq. Indeed, it shows that LS can be derived naturally from a much more general one. It shows also 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 many examples along the paper.

A following to this work could be to build tools for proving LS programs into Coq. It could be interesting then to think about a notion of *"synchronous property"*. Indeed, we may imagine a logic following Curry-Howard isomorphism such proofs in this logic are synchronous functions, and then find 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.

References

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7, 1999.

[BBC+00] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual, version 6.3.1.* Coq Project, INRIA-Rocquencourt, May 2000.

[BCDPV99] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7.* IEEE Computer Society, January 1999.

[BH01] S. Boulmé and G. Hamon. Certifying synchrony for free. In *Logic for Programming and Reasonning, LPAR'01*, volume 2250 of *Lecture Notes in Artificial Intelligence*, 2001.

[BLJ91] 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.

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

[CGJ99] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in Coq. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'99)*, number 1690 in Lecture Notes in Computer Science, 1999.

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

[CP98] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, 28-29 March 1998. Extended version available as a VERIMAG tech. report no. 97–07)) at `www-spi.lip6.fr/~pouzet`.

[CP01] Pascal Cuoq and Marc Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[DCC00] C. Dumas-Canovas and P. Caspi. A PVS proof obligation generator for Lustre programs. In *Logic for Programming and Reasonning, LPAR2000*, number 1955 in Lecture Notes in Artificial Intelligence, 2000.

[Fil99] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, July 1999.

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

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

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

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HP00] Grégoire Hamon and Marc Pouzet. Modular resetting of synchronous dataflow programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[KNT00] M. Kerboeuf, D. Nowak, and J.-P. Talpin. The steam-boiler problem in Signal-Coq. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'2000)*, number 1869 in Lecture Notes in Computer Science, 2000.

[NBT98] D. Nowak, J.-R. Beauvais, and J.-P. Talpin. Co-inductive axiomatization of a synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98)*, number 1479 in Lecture Notes in Computer Science, 1998.

[Par95] Catherine Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Mathematics for Programs Constructions'95*, number 947 in Lecture Notes in Computer Science, 1995.

[PCCH01] Marc Pouzet, Paul Caspi, Pascal Cuoq, and Grégoire Hamon. *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] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993.

[Str93] Thomas Streicher. *Investigations into intensional type theory.* Habilitationsschrift, Ludwig Maximilian Universität, München, 1993.

Index