A constructive denotational semantics for Kahn networks in Coq

Christine Paulin-Mohring

A constructive denotational semantics for Kahn networks in Coq*

Christine Paulin-Mohring
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

November 13, 2009

Abstract

Semantics of programming languages and interactive environments for the development of proofs and programs are two important aspects of Gilles Kahn's scientific contributions. In his paper "The semantics of a simple language for parallel programming" [11], he proposed an interpretation of (deterministic) parallel programs (now called Kahn networks) as stream transformers based on the theory of complete partial orders (cpos). A restriction of this language to synchronous programs is the basis of the data-flow Lustre language which is used for the development of critical embedded systems [14, 10].

We present a formalization of this seminal paper in the Coq proof assistant [4, 15]. For that purpose, we developed a general library for cpos. Our cpos are defined with an explicit function computing the least upper bound (lub) of an increasing sequence of elements. This is different from what Kahn developed for the standard Coq library where only the existence of lubs (for arbitrary directed sets) is required, giving no way to explicitly compute a fixpoint. We define a cpo structure for the type of possibly infinite streams. It is then possible to define formally what is a Kahn network and what is its semantics, achieving the goal of having the concept closed under composition and recursion. The library is illustrated with an example taken from the original paper as well as the Sieve of Eratosthenes, an example of a dynamic network.

1 Introduction

Semantics of programming languages and interactive environments for the development of proofs and programs are two important aspects of Gilles Kahn's scientific contributions. In his paper "The semantics of a simple language for

*appeared in From Semantics to Computer Science, Essays in Honour of Gilles Kahn, edited by Y. Bertot, G. Huet, J.-J. Lvy and G. Plotkin, Cambridge University Press 2009, ISBN-13: 9780521518253

parallel programming" [11], he proposed an interpretation of (deterministic) parallel programs (now called Kahn networks) as stream transformers based on the theory of complete partial orders (cpos). A restriction of this language to synchronous programs is the basis of the data-flow Lustre language [14, 10] which is used now for the development of critical embedded systems. Because of the elegance and generality of the model, Kahn networks are also a source of inspiration for extensions of the data-flow synchronous paradigm to higher-order constructions [7] or to more permissive models of synchrony [8].

We present a formalization of this seminal paper in the Coq proof assistant [4, 15]. For that purpose, we developed a general library for cpos. Our cpos are defined with an explicit function computing the least upper bound (lub) of a monotonic sequence of elements. This is different from what Kahn developed for the standard Coq libraries where only the existence of lubs is required, giving no way to explicitly compute a fixpoint. However, Kahn's library was intended as the background for a computer formalisation of the paper "Concrete Domains" by Kahn and Plotkin [13] and it covers general cpos with the existence of a lub for arbitrary directed sets while our work only considers ω-cpos with lubs on monotonic sequences which is a sufficent framework for modeling Kahn networks.

We define a cpo structure for the type of possibly infinite streams. This is done using a coinductive type in Coq with two constructors, one for adding an element in front of a stream, the second constructor add a silent step `Eps`. From the structural point of view, our streams are infinite objects; this is consistent with the fact that these streams are models for communication links which are continuously open even if there is no traffic on the line. However, we identify the empty stream with an infinite stream of `Eps` constructors, so our data type models both finite and infinite streams. We define the prefix order on this data type and the corresponding equality. We also develop useful basic functions: the functions for head, tail and append used by Kahn [11], but also a filtering and a map function.

It is then possible to define formally what is a Kahn network and what is its semantics, achieving the goal of having the concept closed under composition and recursion. A Kahn network will be defined by a concrete set of edges corresponding to links in the network, each one associated with the type of the objects which are transmitted on that link. With each noninput edge is associated a node which is a continuous function producing a stream of outputs from streams given as inputs. The type of Kahn networks has a natural cpo structure. The semantics of a Kahn network is obtained in the following way: we provide streams for the input edges of the system, then the system is interpreted as an equation on the streams corresponding to the traffic on all the edges, seen as a continuous function. The semantics of the network is the fixpoint of this continuous function. We prove that this solution is a continuous function both of the network and of the input streams. By selecting the appropriate outputs, a system can be interpreted as a new node to be used in another system. Also the continuity with respect to the system itself gives the possibility of recursively defining a system.

Our library is illustrated with an example taken from the original paper as well as the Sieve of Eratosthenes, an example of a dynamic network, recursively defined.

Outline The remaining part of the introduction gives the main notations used in this paper. Section 2 recalls Kahn's approach in [11] which introduces cpos as a natural structure for the semantics of a simple parallel language. Section 3 introduces our definition of Cpo structures in Coq. It is based on a structure of ordered types. We define the cpos of monotonic and continuous functions as well as several constructions for product of cpos. Section 4 introduces the type of possibly infinite streams and defines a cpo structure on it (in particular a lub function). We start with the simpler case of the flat cpo. We define a general function on the cpo of streams computing the value depending on the head and tail of the stream (and giving the empty stream in case the input is empty). We derive from this operator the constructions for head, tail and append as well as functionals for mapping a function on all elements of the stream or filtering the elements of a stream with respect to a boolean condition. We derive a cpo structure for natural numbers as a particular case of streams. Section 5 defines a type for Kahn networks and the associated semantics. Section 6 illustrates our library on two examples, one taken from the original paper [11], the other is the Sieve of Eratosthenes.

Notation In this paper, we use mathematical notations close to the Coq notations.

The expression $A \to B$ represents both the type of functions from type A to type B and the proposition: "A implies B". The arrow associates to the right: $A \to B \to C$ represents $A \to (B \to C)$.

The expression $\forall x, P$ represents the proposition "for all x, P" or the type of dependent functions which associate with each x an object of type P. We can annotate a variable with its type and put several binders like in $\forall (x\, y : A)(z : B), P$ which represents the property: "for all x and y of type A and z of type B, P holds".

The function which maps a variable x of type A to a term t is written $\mathsf{fun}\, x \Rightarrow t$ or $\mathsf{fun}\, x : A \Rightarrow t$. We can introduce several binders at the same time.

We write $c\, x_1 \ldots x_n \stackrel{\text{def}}{=} t$ to introduce c as an abbreviation for the term $\mathsf{fun}\, x_1 \ldots x_n \Rightarrow t$. We write $x = y$ for (polymorphic) definitional equality in Coq (i.e. terms that are structurally equal). We use the notation $x == y$ for a specific equivalence relation associated with the type of x and y, defined as a setoid equality[1].

[1] In Type theory, there is a natural polymorphic equality called Leibniz equality which corresponds to convertibility and which allows rewriting in an arbitrary context. This equality is sometimes too strong. It is also possible to associate with a type a specific equivalence relation. The type together with the relation is called a setoid. The relation can be used pretty-much like an equality for rewriting, but only under a context built using operators which are proved to preserve the setoid relation. Coq offers facilities to manipulate the setoids.

We shall also use the Coq notation $\{x : A|P\}$ for the type of pairs (a, p) with a of type A and p a proof of $P[a/x]$ and $\{x : A\&\{y : B|P\}\}$ for the type of triples (a, b, p) with a of type A, b of type B and p a proof of $P[a/x, b/y]$.

2 From a simple parallel languages to cpos

In [11], Kahn proposes an algol like language to describe parallel system. Each process is defined as a procedure with input and output parameters which are interpreted as channels. In the body of the procedure, there is a possibility to wait for a value on an input parameter or to send a value on an output parameter. Global channels can be declared, the processus can be instatiated and executed in parallel. The idea is that each channel correponds to a fifo in which values can be stored or read. There is no bound on the size of the fifo and a processus can be blocked waiting a value on an empty channel.

More precisely a Kahn network is built from autonomous computing stations linked together. The stations exchange information through communication lines. The assumptions are that a computing station receives data from input lines, computes using its own memory and produces result on some of the output lines. The communication lines are the only medium of communication between stations and the transmission of data on these lines is done in a finite amount of time. With each communication line, we associate the type of data which transit on that line. Each node can use the history of the inputs to produce an output, so it can be seen as a function from the streams of inputs to the streams of outputs. This function is continuous, which means that an output can be delivered without waiting for an infinite amount of information on the inputs lines.

A Kahn network is represented as an oriented graph. The nodes are the computing stations and the edges are the communication lines. We distinguish the input lines which have no source node. The graphical representation of the example in Kahn's paper is given in figure 1.

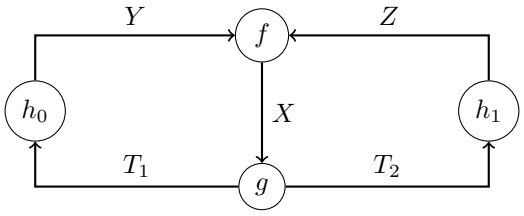

Figure 1: A simple example of Kahn network

In Kahn semantics, we look at the sequence of values that will be sent on each channel. This is a possibly infinite sequence (a stream). Locally each node is interpreted as a function taking as input a stream for each input edge and computing a stream for each output edge.

The system itself will behave as the solution of a set of equations defining the stream of values on the channels (one equation for each node). In our example, the system of equations will be:

$$X = f(Y, Z) \quad Y = h_0(T_1) \quad Z = h_1(T_2) \quad T_1 = g_1(X) \quad T_2 = g_2(X)$$

This is a recursive definition. In order to ensure the existence of a solution, we use a cpo structure on the set of streams (with the prefix order) and we prove that each node corresponds to a monotonic and continuous function.

Now if we have a system and we distinguish input and output edges, the solution is itself a continuous function from inputs to outputs so behaves like a node. This is an essential property for a modular design of systems. It gives also the possibility to recursively defined a system, like the Sieve of Eratosthenes which we describe in 6.2 and which corresponds to the network in figure 2.

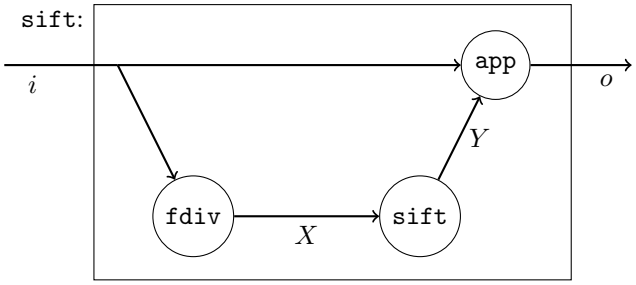

Figure 2: A Kahn network for the Sieve of Eratosthenes

The precise definition of nodes in the previous examples will be given in section 6.

3 Formalizing cpos constructively

The basic structure used for the interpretation of Kahn networks is the ω-complete partial order. We developed a Coq library of general results on ω-cpos.

3.1 Definition

An ω-cpo is a type D with a partial order $\leq$, a least element (usually written $\bot$) and a least-upper bound (written $\mathtt{lub}\, h$) for any monotonic sequence h of elements in D ($h : nat \to D$ such that $\forall nm, n \leq m \to h\, n \to h\, m$).

An ω-cpo is actually a weaker structure than ordinary cpos where lubs exist for any directed set of elements. However, ω-cpos are sufficient for the construction of fixpoints. In the following we refer to ω-cpos simply as cpos.

3.1.1 Ordered structure

We define the type `ord` of ordered structures. An object in the type `ord` is a dependent record with a type A, a binary relation $\leq$ on A, and a proof that this relation is reflexive and transitive.

An example of an ordered structure is the type `nat` of natural numbers with the usual order. In the following, we shall abusively write `nat` for the object of type `ord` corresponding to this structure.

Notations When O is an ordered structure, we write $x : O$ to mean that x is an element of the carrier of O. The coercion mechanism in Coq allows us to support this abuse of notation: whenever a type is expected and an ordered structure O is provided, Coq automatically coerces the term O to its carrier.

When O is an ordered structure and $x, y : O$, the Coq notation mechanism allows to write $x \leq y$ to express that x and y are in the relation associated with the ordered structure O. We shall write $x \leq_O y$ when we want to make the ordered structure explicit.

Equality We define an equality on an ordered structure by:
$x == y \stackrel{\text{def}}{=} x \leq y \wedge y \leq x$;
this is obviously an equivalence relation.

Order on functions Given an ordered structure O and a type A, there is a natural order on the type $A \to O$ of functions from A to O, called the pointwise order, which is defined by $f \leq_{A \to O} g \stackrel{\text{def}}{=} \forall x, f\,x \leq_O g\,x$. We write $A \xrightarrow{o} O$ for the corresponding ordered structure.

Monotonic functions Given two ordered structures O_1 and O_2, we introduce the type of monotonic functions from O_1 to O_2. Elements of this type are records with a function f of type $O_1 \to O_2$ (formally from the carrier of O_1 to the carrier of O_2) and a proof that this function is monotonic. With the pointwise order on functions, this type is an ordered structure written $O_1 \xrightarrow{m} O_2$.

If an object f has type $O_1 \xrightarrow{m} O_2$, it is formally a pair with a function and a proof of monotonicity. In Coq, we introduce a coercion from f to a function from O_1 to O_2, such that we can write $(f\,x)$ in a way consistent with mathematical practice. We consequently have the following property:
$\forall (f : O_1 \xrightarrow{m} O_2)(x\,y : O_1), x \leq y \to f\,x \leq f\,y$.
We also proved that any monotonic function preserves equality.

The composition of two monotonic functions is monotonic: when $f : O_1 \xrightarrow{m} O_2$ and $g : O_2 \xrightarrow{m} O_3$, we define $g@f$ of type $O_1 \xrightarrow{m} O_3$ the monotonic function such that $(g@f)\,x = g\,(f\,x)$

Notation fun $x \stackrel{m}{\Rightarrow} t$**.** If t is an expression of type O_2 depending on x of type O_1, we write $\mathsf{fun}\,x \stackrel{m}{\Rightarrow} t$ for the object f in Coq of type $O_1 \xrightarrow{m} O_2$, which is a monotonic function such that $f\,x = t$. In Coq, the object f is formally a pair

built from the function $\mathsf{fun}\, x \Rightarrow t$ and a proof of monotonicity. The (informal) notation $\mathsf{fun}\, x \overset{m}{\Rightarrow} t$ hides the proof of monotonicity and helps us to insist on the functional behavior of f. In our Coq development we try to systematically define objects in $O_1 \overset{m}{\to} O_2$ using combinators like the composition of monotonic functions in order to get proof of monotonicity for free relying on the type system. After the definition of such an object f, we systematically prove a (trivial) lemma $f\, x = t$ which captures the functional behavior of the monotonic function. This lemma is useful for rewriting the expressions involving f.

3.1.2 Cpo structure

A cpo structure is defined as a record which contains:

- an ordered structure O;
- a least element $\bot$ of type O;
- a least upper-bound function `lub` for monotonic sequences; the constant `lub` has type: $(\mathtt{nat} \overset{m}{\to} O) \to O$;
- proofs of the following properties:
  - $\forall x : O, \bot \leq x$
  - $\forall (f : \mathtt{nat} \overset{m}{\to} O)(n : \mathtt{nat}), f n \leq \mathtt{lub}\, f$
  - $\forall f : (\mathtt{nat} \overset{m}{\to} O)(x : O), (\forall n, f n \leq x) \to \mathtt{lub}\, f \leq x$

A cpo structure is implicitly identified with the underlying ordered structure. In particular, if D_1 and D_2 are two cpo structures, we can write $D_1 \overset{m}{\to} D_2$ for the ordered structure of monotonic functions from D_1 to D_2.

Continuity It is easy to show from the properties of `lub` that given D_1 and D_2 two cpo structures, $F : D_1 \overset{m}{\to} D_2$ a monotonic function from D_1 to D_2 and f a monotonic sequence on D_1, we have

$$\mathtt{lub}\,(F @ f) \leq F\,(\mathtt{lub}\, f)$$

We say that F is *continuous* whenever the other direction is true, namely:

$$\forall f : \mathtt{nat} \overset{m}{\to} D_1, F\,(\mathtt{lub}\, f) \leq \mathtt{lub}\,(F @ f).$$

We write $D_1 \overset{c}{\to} D_2$ for the ordered structure of continuous functions. When g has type $D_2 \overset{c}{\to} D_3$ and f has type $D_1 \overset{c}{\to} D_2$, we write $g @\_ f$ the element of $D_1 \overset{c}{\to} D_3$ which corresponds to the composition of f and g, ie such that $(g @\_ f)\, x = g\,(f\, x)$.

3.2 Cpo constructions

The structure of cpos is preserved by the usual constructions of functions and products. In this part we show constructions for the cpos of functions, monotonic functions and continuous functions as well as the product of two cpos, of an arbitrary family of cpos and the k-times product D^k of a cpo D.

3.2.1 Functional constructions

Given a cpo structure D and a type A, we can define a cpo structure on the set of functions from A to D using a pointwise construction for $\bot$ and $\texttt{lub}$:

$$\bot_{A\to D} \stackrel{\text{def}}{=} \mathsf{fun}\, x \to \bot_D \qquad \texttt{lub}_{A\to D}\, h \stackrel{\text{def}}{=} \mathsf{fun}\, x \Rightarrow \texttt{lub}_D(\mathsf{fun}\, n \stackrel{m}{\Rightarrow} h\, n\, x)$$

We write $A \stackrel{O}{\to} D$ for the cpo of simple functions from A to D.

Given an ordered type O, it is easy to show that $\bot_{O\to D}$ is a monotonic function and that $\texttt{lub}$ preserves monotonicity. So we have a cpo structure written $O \stackrel{M}{\to} D$ on the type of monotonic functions.

If D_1 and D_2 are two cpo structures, then because $\bot_{D_1 \stackrel{m}{\to} D_2}$ is a continuous function and $\texttt{lub}$ preserves continuity, we also have a cpo structure on continuous functions from D_1 to D_2. We write $D_1 \stackrel{C}{\to} D_2$ for this cpo structure.

3.2.2 Product constructions

We formalized other constructions on cpos corresponding to products.

Binary product The binary product $D_1 \times D_2$ of two cpo structures has a cpo structure written $D_1 \otimes D_2$.

$$\bot_{D_1\otimes D_2} \stackrel{\text{def}}{=} (\bot_{D_1}, \bot_{D_2})$$

$$\texttt{lub}_{D_1\otimes D_2}\, h \stackrel{\text{def}}{=} (\texttt{lub}_{D_1}(\mathsf{fun}\, n \stackrel{m}{\Rightarrow} \texttt{fst}\,(h\, n)), \texttt{lub}_{D_2}(\mathsf{fun}\, n \stackrel{m}{\Rightarrow} \texttt{snd}\,(h\, n)))$$

The projection and pairing functions are continuous, we have defined

- $\texttt{FST} : \forall D_1 D_2, D_1 \otimes D_2 \stackrel{C}{\to} D_1 \qquad \texttt{SND} : \forall D_1 D_2, D_1 \otimes D_2 \stackrel{C}{\to} D_2$;
- $\texttt{PAIR} : \forall D_1 D_2, D_1 \stackrel{C}{\to} D_2 \stackrel{C}{\to} D_1 \otimes D_2$.

We also defined functions for currying and uncurrying

- $\texttt{CURRY} : \forall D_1 D_2 D_3, ((D_1 \otimes D_2) \stackrel{C}{\to} D_3) \stackrel{C}{\to} D_1 \stackrel{C}{\to} D_2 \stackrel{C}{\to} D_3$,
- $\texttt{UNCURRY} : \forall D_1 D_2 D_3, (D_1 \stackrel{C}{\to} D_2 \stackrel{C}{\to} D_3) \stackrel{C}{\to} (D_1 \otimes D_2) \stackrel{C}{\to} D_3$.

Indexed product $\prod D$ For modeling Kahn networks, it is useful to have a generalized product over an arbitrary number of cpos.

We take a set I of indexes and a family D of cpos indexed by I, that is, $D : I \to \texttt{cpo}$. The cpo structure for the product written $\prod D$ is just a dependent generalization of the function type $I \stackrel{o}{\to} D$ in which the domain D may depend on the index. When D is a type expression depending on a free variable i, we write $\prod_i D$ for $\prod(\mathsf{fun}\, i \Rightarrow D)$

- Carrier: $\forall i : I, D\, i$

- Order: $x \leq_{\Pi D} y \stackrel{\text{def}}{=} \forall i, x\,i \leq_{D\,i} y\,i$
- Least element: $\bot_{\Pi D} \stackrel{\text{def}}{=} \mathsf{fun}\,i \Rightarrow \bot_{D\,i}$
- Least upper bound: $\mathtt{lub}_{\Pi D}\,h \stackrel{\text{def}}{=} \mathsf{fun}\,i \Rightarrow \mathtt{lub}_{D\,i}(\mathsf{fun}\,n \stackrel{m}{\Rightarrow} h\,n\,i)$

The interesting constructions on that structure are

- a projection function `PROJ` of type: $\forall i : I, (\prod D) \stackrel{C}{\to} D\,i$
- given two indexed families D and D' over the set I, the mapping `MAPi` of a continuous function on the elements of an indexed product has type $(\forall i, Di \stackrel{C}{\to} D'i) \to \prod D \stackrel{C}{\to} \prod D'$ and is defined such that $\mathtt{MAPi}\,f\,p\,i = f\,i\,(p\,i)$.
- An operation to lift the indexes. Assume that we have two sets of indexes I and J, a family D of cpos indexed by I, and a function $f : J \to I$. We define a continuous function `LIFTi` of type $\prod D \stackrel{C}{\to} \prod_j D\,(f\,j)$ such that $\mathtt{LIFTi}\,p\,j = p\,(f\,j)$. It allows to select, reorganize or duplicate the elements in the product.

Finite product D^k It is also useful to have a finite product on the same cpo D. Given $k : \mathtt{nat}$, one possibility is to take the function space $\{i | i < k\} \stackrel{O}{\to} D$, but in that case we will have to deal with the subset type in Coq which is not always convenient. Instead we take the type $\mathtt{nat} \to D$ but instead of the pointwise order for functions, we introduce an order up-to k: $f \leq g \stackrel{\text{def}}{=} \forall n, n < k \to f\,n \leq g\,n$. We write $k \to D$ for the cpo structure with this order. The least element is defined pointwise. For the lub, there is a small difficulty. The natural definition would be:

$$\mathtt{lub}_{k \to D}\,h\,n = \mathtt{lub}_D\ (\mathsf{fun}\,p \stackrel{m}{\Rightarrow} h\,p\,n)$$

But $(\mathsf{fun}\,p \Rightarrow h\,p\,n)$ is monotonic only when $n < k$. However, the value of $\mathtt{lub}_{k \to D}\,h\,n$ for $k \leq n$ is meaningless so we can choose an arbitrary one. Consequently we introduce h' such that $h'\,p\,n = h\,p\,n$ when $n < k$ and $\bot$ otherwise. Then taking:

$$\mathtt{lub}_{k \to D}\,h\,n = \mathtt{lub}_D\ (\mathsf{fun}\,p \stackrel{m}{\Rightarrow} h'\,p\,n)$$

gives us the expected properties.

3.3 Fixpoints

Given a cpo structure D and a monotonic function F of type $D \stackrel{m}{\to} D$, we can define a monotonic function $(\mathtt{iter}\,F)$ of type $\mathtt{nat} \stackrel{m}{\to} D$ such that $\mathtt{iter}\,F\,0 = \bot$ and $\mathtt{iter}\,F\,(n+1) = F\,(\mathtt{iter}\,F\,n)$.

We define the fixpoint of F as the least-upper bound of this sequence: $\mathtt{fixp}\,F \stackrel{\text{def}}{=} \mathtt{lub}\,(\mathtt{iter}\,F)$.

The constant `fixp` has type $(D \xrightarrow{m} D) \to D$. It is itself a monotonic function.

It is easy to show that $\texttt{fixp}\, F \leq F\,(\texttt{fixp}\, F)$. The equality $\texttt{fixp}\, F == F\,(\texttt{fixp}\, F)$ is provable under the assumption that F is a continuous function.

We can also show that `fixp` is a continuous function from the cpo $(D \xrightarrow{C} D)$ of continuous functions on D to D. Consequently, we are able to define `FIXP` of type $(D \xrightarrow{C} D) \xrightarrow{C} D$, such that for all F of type $D \xrightarrow{C} D$:

$$\texttt{FIXP}\, F = \texttt{fixp}\,(\texttt{fun}\, x \xRightarrow{m} F\, x) \qquad \texttt{FIXP}\, F == F\,(\texttt{FIXP}\, F).$$

Scott's induction principle We proved Scott's induction principle. A predicate is said to be *admissible* if it is true for the `lub` of a monotonic sequence when it is true for all the elements of the sequence:
$\texttt{admissible}P \stackrel{\text{def}}{=} \forall f : \texttt{nat} \xrightarrow{m} D, (\forall n, P\,(f\, n)) \to P(\texttt{lub}\, f)$.

Scott's induction principle states that when a predicate is admissible, if it is true for $\bot$ and preserved by a monotonic function $F : D \xrightarrow{m} D$, then it is true for the fixpoint of F:

$$\forall P, \texttt{admissible}\, P \to P \bot \to (\forall x, P\, x \to P\,(F\, x)) \to P\,(\texttt{fixp}\, F)$$

Minimality It is easy to prove that the fixpoint of a monotonic function is the minimal solution for the equation. Namely:
$\forall (F : D \xrightarrow{m} D)(x : D), F\, x \leq x \to \texttt{fixp}\, F \leq x$.

4 The cpo of streams

We now want to define a cpo structure for concrete data types. Before developing the construction for streams, we show the simpler case of a flat cpo, which illustrates the main ideas.

4.1 The flat cpo

The simplest nontrivial (i.e., not reduced to $\bot$) cpo is the flat domain. Given a type A, we add an extra element $\bot$ and we have $x \leq b$ if and only if $x = \bot$ or $x = b$.

A natural solution could be to take as the carrier for this cpo the option type on A with values either `None` or $\texttt{Some}\, a$ with $a : A$.

```
Inductive option (A:Type) : Type :=
  None :  option A | Some : A → option A
```

The constant `None` will be the least element. However we cannot define constructively a least upper bound. Indeed, given an increasing sequence of elements in our domain, we would have to decide whether all the elements are $\bot$ in which case the lub is $\bot$ or if there exists an element in the sequence which is of the form $\texttt{Some}\, a$ in which case the lub is this element. Because we follow a constructive

approach in Coq where functions correspond to algorithms, we cannot define a function which takes such a decision.

The computation of lubs is possibly an infinite process, a solution to represent infinite computations in Coq is to use coinductive types. This is the approach taken by V. Capretta [6] for dealing with general recursive functions in Coq. The solution is to introduce:

```
CoInductive flat (A:Type) : Type :=
  Eps : flat A → flat A | Val : A → flat A
```

A value in type `flat` is either finite of the form $\overbrace{\texttt{Eps}\,(\ldots(\texttt{Eps}}^{n}\,(\texttt{Val}\,a))\ldots)$ (written $\texttt{Eps}^n\,(\texttt{Val}\,a)$) in which case it represents the value a (with extra `Eps` steps corresponding to silent computations) or an infinite object $\texttt{Eps}^\infty$ coinductively defined by $\texttt{Eps}^\infty = \texttt{Eps}\,\texttt{Eps}^\infty$ corresponding to a diverging computation.

$\texttt{Eps}^\infty$ will be our least element and we need to identify all the representations of the value a ignoring the `Eps` constructors.

In order to achieve that, we define co-inductively the order on the flat domain with these three rules:

$$\frac{x \leq y}{\texttt{Eps}\,x \leq \texttt{Eps}\,y} \qquad \frac{x \leq \texttt{Val}\,a}{\texttt{Eps}\,x \leq \texttt{Val}\,a} \qquad \frac{y = \texttt{Eps}^n(\texttt{Val}\,a)}{\texttt{Val}\,a \leq y}$$

From this definition we proved reflexivity, transitivity and properties like $\texttt{Eps}^\infty \leq x$ or $\texttt{Val}\,a \leq x \rightarrow x == \texttt{Val}\,a$.

We can now look at the construction of lubs. We have a monotonic sequence h of elements in $\texttt{flat}\,A$ and we want (constructively) to build the least upper bound which can be either $\perp$ or a value.

If x is an element of $\texttt{flat}\,A$, we write $[x]_n$ for the same element but removing the n-th first `Eps` constructors (or less if we find a value before). We have $x == [x]_n$. Now in order to build the least upper bound, we look at $h\,0$. If we find a value then we have our bound; if not, we produce an `Eps` and we continue by looking at $[h\,0]_1; [h\,1]_1$ if we find a value then we are done, if the elements start with an `Eps` then we produce an `Eps` in the least upper bound and we continue. At the n-th step we look at the sequence $[h\,0]_n; [h\,1]_n; \ldots; [h\,n]_n$, we try to find a direct value, otherwise we produce an `Eps` step and continue. This mechanism is illustrated in the figure 3; the Coq formalisation will be given in the more involved case of streams. If one of the elements $(h\,k)$ in the sequence is a value, then there exists n such that $[h\,k]_n = \texttt{Val}\,a$ so we will eventually find this value before the p-th step with $k \leq p$ and $n \leq p$.

4.2 Streams

We now look at the case of streams which is developed following the same kind of reasoning.

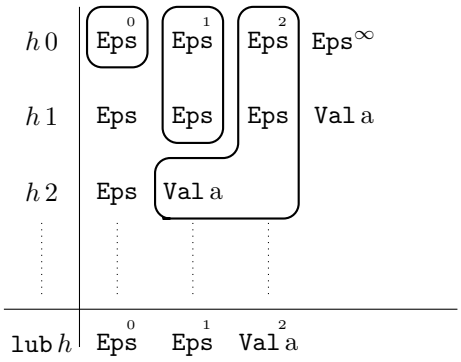

Figure 3: Computation of lubs in a flat domain

4.2.1 Definition

The type of streams is co-inductively defined as:

```
CoInductive Str (A:Type):Type :=
  Eps : Str A → Str A | cons : A → Str A → Str A
```

As before $\texttt{Eps}^\infty$ can be coinductively defined by $\texttt{Eps}^\infty = \texttt{Eps}\,\texttt{Eps}^\infty$. It represents the empty stream and is also our $\bot$ element. We define a function $[\_]_n$ removing `Eps` in front of the stream by induction on n and case analysis on the stream.

$$[s]_0 = s \qquad [\texttt{Eps}\,x]_{n+1} = [x]_n \qquad [\texttt{cons}\,a\,x]_{n+1} = \texttt{cons}\,a\,x$$

4.2.2 Order

The order we consider on streams is the prefix order, we must also ignore the `Eps` steps which correspond to silent computations.

$$\frac{x \leq y}{\texttt{Eps}\,x \leq y} \qquad \frac{[y]_n = \texttt{cons}\,a\,z \qquad x \leq z}{\texttt{cons}\,a\,x \leq y}$$

The idea is that in order to show that $x \leq y$, there are two cases: if x is $\texttt{Eps}\,x'$ then we try to show $x' \leq y$, if x is $(\texttt{cons}\,a\,x')$ then after a finite number of `Eps`, y should be of the form $(\texttt{cons}\,a\,y')$ and we need to show that x' is less than y'. We do not know how many `Eps` steps we should remove so we cannot decide whether $x \leq y$ or not, and similarly we cannot decide whether a stream is finite or not. This corresponds well to the vision of the streams as a model of asynchronous communication links: it is not possible to know if more information will arrive and when. If we want to transmit a finite number of elements, we have to decide on a special character to indicate the end of the data.

Decidability of the empty stream is not required for the development we want to perform and it is the price to pay for having an explicit computation of lubs and fixpoints.

We can prove the expected properties of the order besides reflexivity and transitivity :

- $\texttt{Eps}^\infty \leq x$
- $\neg(\texttt{cons}\, a\, x \leq \texttt{Eps}^\infty)$
- $\texttt{cons}\, a\, x \leq \texttt{cons}\, b\, y \leftrightarrow a = b \wedge x \leq y$

Equality As in other ordered structures, equality on streams $x == y$ is defined as $x \leq y \wedge y \leq x$. It is important to distinguish this equality from intensional equality in Coq: $x = y$ means that x and y are structurally equal. For instance we have $x == \texttt{Eps}\, x$ for all x while $x = \texttt{Eps}\, x$ is only true for the $\texttt{Eps}^\infty$ stream.

Simulation properties Coinductive definitions in Coq correspond to greatest fixpoints. The primitive way to build a proof of $x \leq y$ is to use fixpoint definitions in Coq, which should be guarded. This guarded condition is very syntactic and not always convenient to use in tactics. An alternative way is to define a co-induction principle which in this case corresponds to a simulation principle.

We introduce a principle which does not rely on the intensional representation of streams. We have to find a relation R on streams that is compatible with equality on streams and such that when $R\, x\, y$ holds and x is equal to a stream $\texttt{cons}\, a\, x'$ then y is also equal to $\texttt{cons}\, a\, y'$ and $R\, x'\, y'$ also holds. If such an R exists then it is included in the relation $\leq$ on streams. This principle can be written as follows:

$$\frac{\begin{array}{c}\forall x\, y\, z\, t, x == z \to y == t \to R\, x\, y \to R\, z\, t \\ \forall a\, x\, y, R\, (\texttt{cons}\, a\, x)\, y \to \exists z, y == \texttt{cons}\, a\, z \wedge R\, x\, z\end{array}}{\forall x\, y, R\, x\, y \to x \leq y}$$

From this we can derive a principle which says that in order to prove $x \leq y$, it is enough to prove it in the particular case where $x == \texttt{cons}\, a\, x'$. This principle is in practice sufficient in order to avoid reasoning on whether x is the empty stream or not.

4.2.3 Least upper bounds

Destructuring streams We introduce a predicate `is_cons` on streams. It is defined as an inductive predicate with two constructors $\texttt{is\_cons}\, x \to \texttt{is\_cons}\, (\texttt{Eps}\, x)$ and $\texttt{is\_cons}\, (\texttt{cons}\, a\, x)$. We can prove that $\texttt{is\_cons}\, x$ is equivalent to $\exists a\, s, x == \texttt{cons}\, a\, s$.

In Coq `is_cons` is a *noninformative* proposition: we know the existence of a and s but we cannot use a and s to compute a value. However if we know that a stream contains a `cons` constructor, the algorithm that removes `Eps`

constructors will eventually stop on a `cons` constructor. In Coq, we defined a function `uncons` of type:

$$\forall x, \texttt{is\_cons}\, x \to \{a : A \& \{s : \texttt{Str}\, A | x == \texttt{cons}\, a\, s\}\}.$$

From the computational point of view, this function takes a (nonempty) stream as input and returns a pair (a, s) of its head and tail plus the proof that $x == \texttt{cons}\, a\, s$. Technically, the function `uncons` is defined in Coq by a fixpoint doing a structural induction on the proof of $(\texttt{is\_cons}\, x)$ and a case analysis on the structure of x.

Building lubs The construction of lubs for monotonic sequences of streams is a generalization of the idea used in the case of flat domains.

Consider a monotonic sequence h of streams, and a natural number n. We can look at the first constructor of $h\,0 \ldots h\,(n-1)$. Either we only have `Eps` steps or there exists $m < n$ such that $h\,m = \texttt{cons}\, a\, s$. In this last case, the lub will start by $\texttt{cons}\, a$ and we know that for all $p \geq m$, the stream $h\,p$ is equal to $\texttt{cons}\, a\, s'$ (because $h\,m \leq h\,p$) such that we can extract a subsequence of h corresponding to the tails of $h\,p$ and recursively find the lub of this sequence. Following this idea, we built a function `fCon` which takes as argument a sequence of stream h and a natural number n and either answers that all $h\,m$ starts with `Eps` for all $m < n$ or find an element a and another sequence h' such that there exists $m < n$ that all $h\,(k+m)$ is equal to $\texttt{cons}\, a\, (h'\, k)$. Formally the Coq type of `fCon` is:

$$\begin{array}{l}\forall (h : \texttt{nat} \xrightarrow{m} \texttt{Str}\, A)(n : \texttt{nat}),\\ \quad \{a : A \,\&\, \{h' : \texttt{nat} \xrightarrow{m} \texttt{Str}\, A |\\ \qquad\qquad \exists m, m < n \wedge \forall k, h\,(k+m) == \texttt{cons}\, a\, (h'\, k)\}\}\\ \quad + \{\forall m, m < n \to h\, m = \texttt{Eps}(\_)\}\end{array}$$

This function is defiend by structural recursion on n.

We write $[h] \stackrel{\text{def}}{=} \texttt{fun}\, n \stackrel{m}{\Rightarrow} [h\, n]_1$ for the sequence h where we have removed the first `Eps` step of each stream, our lub function is defined with a cofixpoint:

```
CoFixpoint lubn (h:nat⟶m Str A) (n:nat) : Str A :=
match fCon h n with
      (a,h',_) ⇒ cons a (lubn h' 1)
    | _        ⇒ Eps (lubn [h] (n+1))
end
```

This recursive function is accepted by Coq because it is a *guarded fixpoint*: any recursive call in the body appears under a constructor of the type of streams.

The `lub` of a stream sequence h is just defined as $(\texttt{lubn}\, h\, 1)$. We proved that this is the lub of the sequence of streams.

4.2.4 Useful functions on streams

In his paper, Gilles Kahn introduced several functions to manipulate streams: taking the first element, the stream without the first element or the concatenation of the first element of a stream to another.

All these functions are proved to be continuous. In our development, we derive them from the more general scheme of a function from streams to streams defined by case analysis on the structure of the input stream. If the input is equal to $\texttt{cons}\,a\,x$ then the result is computed from a and x by a given function F, if it is $\texttt{Eps}^\infty$ then the result is $\texttt{Eps}^\infty$.

Let a parameter function F have type $A \to \texttt{Str}A \xrightarrow{m} \texttt{Str}B$. The function that we named `case` is coinductively defined by:

$$\texttt{case}\,F\,(\texttt{Eps}\,x) = \texttt{Eps}\,(\texttt{case}\,F\,x) \qquad \texttt{case}\,F\,(\texttt{cons}\,a\,x) = F\,a\,x$$

It is easy to check that $\texttt{case}\,F\,\bot == \bot$ and that $x == \texttt{cons}\,a\,y \to \texttt{case}\,F\,x == F\,a\,y$.

When F is a continuous function of type $A \to \texttt{Str}A \xrightarrow{C} \texttt{Str}B$, then $\texttt{case}\,(\mathsf{fun}\,a\,s \xRightarrow{c} F\,a\,s)$ is also a continuous function of type $\texttt{Str}A \xrightarrow{C} \texttt{Str}B$. The `case` construction is also continuous with respect to the argument F, such that we can build `CASE` of type $(A \xrightarrow{O} \texttt{Str}A \xrightarrow{C} \texttt{Str}B) \xrightarrow{C} \texttt{Str}A \xrightarrow{C} \texttt{Str}B$.

From this scheme, we derive the following functions:

- `first` has type $\texttt{Str}\,A \to \texttt{Str}\,A$ and is defined by:
  $\texttt{first} \stackrel{\text{def}}{=} \texttt{case}\,(\mathsf{fun}\,a\,s \xRightarrow{m} \texttt{cons}\,a\,\bot)$
- `rem` has type $\texttt{Str}\,A \to \texttt{Str}\,A$ and is defined by:
  $\texttt{rem} \stackrel{\text{def}}{=} \texttt{case}\,(\mathsf{fun}\,a\,s \xRightarrow{m} s)$
- `app` has type $\texttt{Str}\,A \to \texttt{Str}\,A \to \texttt{Str}\,A$ and is defined by:
  $\texttt{app}\,x\,y \stackrel{\text{def}}{=} \texttt{case}(\mathsf{fun}\,a\,s \xRightarrow{m} \texttt{cons}\,a\,y)\,x$.
  We remark that $\texttt{app}\,x\,y$ only takes the first element of x, and adds it in front of y. It corresponds to the "followed by" operation in synchronous data-flow languages and not to the usual append function on lists which cannot be defined in that framework.

We also build their continuous versions: `FIRST` and `REM` of type $\texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A$ and `APP` of type $\texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A$

We proved the properties which are given by Kahn [11]:

$$\texttt{first}\,\bot == \texttt{rem}\,\bot = \texttt{app}\,\bot\,x == \bot$$

$$\texttt{first}\,x == \texttt{app}\,x\,\bot \qquad x == \texttt{app}\,(\texttt{first}\,x)\,(\texttt{rem}\,x)$$

Instead of $x = \bot \vee \texttt{rem}\,(\texttt{app}\,x\,y) == y$ in Kahn [11], we proved that $\texttt{is\_cons}\,x \to \texttt{rem}\,(\texttt{app}\,x\,y) == y$.

We also proved that $\texttt{app}\,(\texttt{first}\,x)\,y == \texttt{app}\,x\,y$.

Bisimulation revisited Using the `rem` function, it is possible to express a bisimulation principle. In order to prove that two streams are equal, it is sufficient to find a relation R which is stable by equality, which implies equality on first elements and is preserved by remainders for nonempty streams. Such a relation R is included in equality:

$$\begin{array}{l}\forall D\,(R : \mathtt{Str}\,A \to \mathtt{Str}\,A \to \mathit{Prop}),\\ (\forall x_1\,x_2\,y_1\,y_2, R\,x_1\,y_1 \to x_1 == x_2 \to y_1 == y_2 \to R\,x_2\,y_2)\\ \to (\forall x\,y, (\mathtt{is\_cons}\,x \vee \mathtt{is\_cons}\,y) \to R\,x\,y \to \mathtt{first}\,x == \mathtt{first}\,y)\\ \to (\forall x\,y, (\mathtt{is\_cons}\,x \vee \mathtt{is\_cons}\,y) \to R\,x\,y \to R\,(\mathtt{rem}\,x)\,(\mathtt{rem}\,y))\\ \to \forall x\,y, R\,x\,y \to x == y.\end{array}$$

4.2.5 Mapping and filtering

Mapping A useful functional on streams is to apply a function $F : A \to B$ to any element of a stream of A in order to obtain a stream of B.

We easily build this function using our fixpoint construction and case analysis.

We first build a function `Mapf` of type $(\mathtt{Str}\,A \xrightarrow{C} \mathtt{Str}\,B) \xrightarrow{C} A \xrightarrow{O} \mathtt{Str}\,A \xrightarrow{C} \mathtt{Str}\,B$ such that $\mathtt{Mapf}\,f\,a\,s = \mathtt{cons}\,(F\,a)\,(f\,s)$.

Then we introduce $\mathtt{MAP} \stackrel{\text{def}}{=} \mathtt{FIXP}\,(\mathtt{CASE}\,@\_\mathtt{Mapf})$ of type $\mathtt{Str}\,A \xrightarrow{C} \mathtt{Str}\,B$ and `map` the underlying function of type $\mathtt{Str}\,A \to \mathtt{Str}\,B$. From the properties of `FIXP`, `CASE` and `Mapf`, we obtain easily the expected equalities:

$$\mathtt{map}\,\bot == \bot \qquad \mathtt{map}\,(\mathtt{cons}\,a\,s) == \mathtt{cons}\,(F\,a)\,(\mathtt{map}\,s).$$

Of course, we could have defined `map` directly in Coq using a guarded fixpoint (a fixpoint where recursive calls are directly under a constructor) on the co-inductive type $\mathtt{Str}\,A$ which satisfies the following equations:

$$\mathtt{map}\,(\mathtt{Eps}\,x) = \mathtt{Eps}\,(\mathtt{map}\,x) \qquad \mathtt{map}\,(\mathtt{cons}\,a\,x) = \mathtt{cons}\,(F\,a)\,(\mathtt{map}\,x)$$

Proving monotonicity and continuity of this function requires specific co-recursive proofs. Our definition using `FIXP` and `CASE` gives us these results directly without extra work.

Our technique applies to recursive definitions of functions on streams which do not directly correspond to guarded fixpoints like the `filter` function.

Filtering. Filtering is an operation that selects elements of a stream that satisfy a given (decidable) property P. This operator has been widely studied because it is the typical example of a nonguarded definition on co-inductively defined infinite streams.

Using p of type $A \to \mathtt{bool}$ to decide the property P, a definition in an Haskell-like language would be:

$$\mathtt{filter}\,p\,(\mathtt{cons}\,a\,s) = \mathsf{if}\ p\,a\ \mathsf{then}\ \mathtt{cons}\,a\,(\mathtt{filter}\,p\,s)\ \mathsf{else}\ \mathtt{filter}\,p\ s$$

The problem is that if P only holds on a finite number of elements of the stream then the output is finite and there is no way to decide that.

Y. Bertot [3] proposes a solution where there is an inductive proof that P holds infinitely many times in the input and this is used to produce an infinite stream as output. An alternative solution is to produce as output an infinite stream of values which are either a real value or a dummy constant.

With our representation of streams, we can simply define the stream in a similar way as for the map function using case analysis and fixpoint. We introduce $\texttt{Filterf}\,p$ of type $(\texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A) \xrightarrow{C} A \xrightarrow{O} \texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A$ such that $\texttt{Filterf}\,p\,f\,a\,s = \mathsf{if}\ p\,a\ \mathsf{then}\ \texttt{cons}\,a\,(f\,s)\ \mathsf{else}\ f\,s$.

Then we introduce $\texttt{FILTER}\,p \stackrel{\text{def}}{=} \texttt{FIXP}\,(\texttt{CASE}\,@\_\,(\texttt{Filterf}\,p))$ of type $\texttt{Str}\,A \xrightarrow{C} \texttt{Str}\,A$ and $(\texttt{filter}\,p)$ the corresponding function of type $\texttt{Str}\,A \to \texttt{Str}\,A$. We easily check the expected property:

$$\texttt{filter}\,p\,(\texttt{cons}\,a\,s) == \mathsf{if}\ p\,a\ \mathsf{then}\ \texttt{cons}\,a\,(\texttt{filter}\,p\,s)\ \mathsf{else}\ \texttt{filter}\,p\ s$$

4.2.6 Finiteness

We can define what it means for a stream to be finite or infinite. As usual, infinity is defined co-inductively and finiteness is an inductive predicate. We defined them the following way:

```
Inductive finite (s:Str A) : Prop :=
  fin_bot : s ≤ ⊥ → finite s
| fin_cons: finite (rem s) → finite s.

CoInductive infinite (s:Str A) : Prop :=
   inf_intro : is_cons s → infinite (rem s) → infinite s.
```

We were able to prove

- $s \le t \to \texttt{infinite}\,s \to \texttt{infinite}\,t$
- $s \le t \to \texttt{finite}\,t \to \texttt{finite}\,s$.
  This property is not provable if we take a different version of `finite` with an extra hypothesis $(\texttt{is\_cons}\,s)$ in the constructor `fin_cons`. With such a definition of `finite`, a proof of $\texttt{finite}\,t$ is isomorphic to the number of `cons` in t. Assuming $s \le t$, a proof of $\texttt{finite}\,s$ should give us the exact number of elements in s, but there is no way to explicitely compute this number. With our definition of `finite`, a proof of $\texttt{finite}\,s$ just gives us an upper bound of the number of `cons` in the stream and is consequentely compatible with the order on the streams.
- $\texttt{finite}\,s \to \neg\texttt{infinite}\,s$

4.3 The particular case of natural numbers

We put an ordered structure on the type `nat` of natural numbers but this is not a cpo because there is no lub for the sequence $h\,n = n$. If we want a cpo structure, we need to add an infinite element. One way to define a cpo for natural numbers reusing our previous library is to take the type of streams on the trivial type `unit` with only one element `tt` : `unit`. The 0 element will be $\texttt{Eps}^\infty$ as before, the successor function will be $\texttt{S}\,x \stackrel{\text{def}}{=} \texttt{cons}\,\texttt{tt}\,x$. We can define the top element S^∞ with the cofixpoint $S^\infty = \texttt{S}\,S^\infty$ and prove $\forall x, x \leq S^\infty$.

We write `Nat` for this cpo. There is an obvious monotonic function from `nat` to `Nat`.

This domain is used in order to define the `length` function from $\texttt{Str}\,A$ to `Nat`. It is just an application of the `map` construction with the functional $\texttt{fun}a \Rightarrow \texttt{tt}$. We were able to show the following properties:

- $\forall s : \texttt{Str}\,A, \texttt{infinite}\,(\texttt{length}\,s) \leftrightarrow \texttt{infinite}\,s$
- $\forall n : \texttt{Nat}, \texttt{S}\,n \leq n \rightarrow \texttt{infinite}\,n$

In the case of streams, we defined the append of streams x and y, just taking the first element of x (if it exists) and putting it in front of y. There is no way to define the usual append function on lists such that the concatenation of the empty stream and y is y, because we never know if x is empty by just looking at a finite prefix.

The situation is a bit different for the cpo of natural numbers where the concatenation corresponds to addition. When trying to add x with y we might look alternatively at the head part of x and y. Whenever we find a successor, we can produce a successor on the output. If one of x or y is 0 then we will always find `Eps` step on this input and the output will be equal to the other argument with just extra `Eps` steps inserted.

Following this idea, we have been able to define the addition as a continuous function on `Nat` and prove that it is commutative and that $\texttt{add}\,n\,0 == n$.

5 Kahn networks

We shall now explain our representation of Kahn networks in Coq.

5.1 Representing nodes

.

We define a shallow embedding of nodes. A *node* with inputs of type $A_1, \ldots, A_n$ and outputs in types $B_1, \ldots, B_p$ is a continuous function of type $\prod_i \texttt{Str}\ A_i \stackrel{C}{\rightarrow} \prod_j \texttt{Str}\ B_j$.

In general we allow arbitrary sets of indexes I for inputs and J for outputs. We associate with each index a type family $A : I \rightarrow \mathsf{Type}$ and $B : J \rightarrow \mathsf{Type}$, a *node of signature* $A\ B$ is an element of $\prod_i \texttt{Str}\ (A\,i) \stackrel{C}{\rightarrow} \prod_j \texttt{Str}\ (B\,j)$.

We distinguish the particular case of a simple node with only one output. Given a set of indexes I for inputs, a type family $A : I \to \mathsf{Type}$ and a type B, we define a *simple node of signature* $A\ B$ to be an element of $\prod_i \mathtt{Str}\ (A\,i) \xrightarrow{C} \mathtt{Str}\ B$.

A node with several outputs can just be seen as a set of simple nodes, each one corresponding to the projection on the corresponding output.

5.2 Representing systems

We start from a concrete set of edges and we distinguish input edges (given by a type I) from the other ones (given by a type J). We consider all the noninput edges to be output edges, it is not relevant for the system which outputs we want to observe.

We associate with each edge a type, so we have a type family $A : I + J \to \mathsf{Type}$. The type $I + J$ is the disjoint union of I and J. We write $(l\,i)$ (resp. $(r\,j)$) for the element of $I + J$ associated with $i : I$ (resp. $j : J$).

We define $SA \stackrel{\text{def}}{=} \mathsf{fun}\,i \Rightarrow \mathtt{Str}\,(A\,i)$ the type family indexed by $I + J$ of streams of elements of type $A\,i$ and $SA_I \stackrel{\text{def}}{=} \mathsf{fun}\,i \Rightarrow SA\,(l\,i)$ the type family indexed by I associated with the inputs.

Now each edge which is not an input edge has a source which is a node. Actually each edge is associated with one particular output of the source node. We already mentioned that a general node with n outputs is equivalent to n simple nodes.

In our model, each noninpu edge is associate to one simple node (its source).

A simple node in general is a function f of type $\prod_{k\in K} \mathtt{Str}\,A'_k \xrightarrow{C} \mathtt{Str}\,B$. We have to link the inputs of the node (indexed by K) with the edges of the system. This corresponds to producing a function $\sigma : K \to (I + J)$ which is compatible with the type system (i.e., A'_k is convertible with $A\,(\sigma\,k)$). Given f and σ, we could use the `DLIFTi` operation introduced in paragraph 3.2.2 in order to build a function f' of type $\prod SA \xrightarrow{C} \mathtt{Str}\,B$, which can also be seen as a simple node but taking all the edges of the system as input.
Instead of introducing for each node the extra level of indirection with the set K and the map $\sigma : K \to (I + J)$, we directly consider that an output edge is associated with a (simple) node of the system which is a continuous map taking all streams associated with an edge as input, i.e., an element of $\prod SA \xrightarrow{C} \mathtt{Str}\,B$. This gives us a very simple and uniform definition of the type of systems that we describe now.

A *system* with input edges I and output edges J of type $A : I + J \to \mathsf{Type}$ is an object of type:

$$\mathtt{system}\,A \stackrel{\text{def}}{=} \forall j : J, (\prod SA) \xrightarrow{C} \mathtt{Str}\,(A\,(r\,j))$$

The set of systems has a cpo structure corresponding to an indexed product $\prod_j (\prod SA \xrightarrow{C} \mathtt{Str}\,(A\,(r\,j)))$.

Equation associated with a system A system defines an equation on the streams associated with the edges, provided we give values for the input edges.

Formally if we have a system s on a type family $A : I + J \to \mathsf{Type}$ as before and an input *inp* which is a product of streams on I such that *inp* has type $\prod SA_I$, then we can define the set of equations as a continuous function ($\texttt{EQN\_of\_system}\, s\, inp$) of type $\prod SA \xrightarrow{C} \prod SA$ such that

$$\texttt{EQN\_of\_system}\, s\, inp\, X\, (l\, i) = inp\, i \qquad \texttt{EQN\_of\_system}\, s\, inp\, X\, (r\, j) = s\, j\, X$$

Taking the fixpoint of this function gives us the output observed on all the edges.

This function `EQN_of_system` is monotonic and continuous with respect to both the system and the input arguments. It has type:
$\texttt{system}\, A \xrightarrow{C} \prod SA_I \xrightarrow{C} (\prod SA \xrightarrow{C} \prod SA)$.

The solution of this equation for a system s and an input *inp* is obtained by taking the fixpoint of the functional ($\texttt{EQN\_of\_system}\, s\, inp$). It is still a continuous function both of the system of the inputs, so we obtain for each system s, a new node $\texttt{SOL\_of\_system}\, s$ of type $(\prod SA_I) \xrightarrow{C} (\prod SA)$ such that:

$$\texttt{SOL\_of\_system}\, s\, inp == \texttt{EQN\_of\_system}\, s\, inp\, (\texttt{SOL\_of\_system}\, s\, inp).$$

Now if we are only interested by a subset O of the output nodes, we use a mapping $\pi : O \to J$ and we use again the lift function in order to restrict our solution to a node indexed by I for inputs and O for outputs of type $(\prod SA) \xrightarrow{C} (\prod_{o:O} SA\, (r\, (\pi\, o)))$.

In the examples, we shall only be interested by one output $o : J$ which is simply obtained by applying $\texttt{SOL\_of\_system}\, s\, inp$ to $(r\, o)$.

5.3 Remarks

There are a few differences between our formalization and Kahn's original definition [11].

- Kahn defined a node as a continuous function, associated with edges for its inputs and outputs. In our formalism, we have the association between the edges and the output of a node but nothing on the link between the input of the node and the edges. The nodes are implicitly related to all edges. In practice, as we shall see in the examples, we shall start with a node defined as a continuous function with the appropriate number of arguments corresponding to the number of input edges in the node. Then, when defining the system, we simply project the relevant edges of the system on the corresponding inputs of the node.

- A noninput edge in our systems has one source node but may have several target nodes. This avoids the explicit use of duplication nodes which is discussed as a possible harmless extension in Kahn [11].

6 Examples

6.1 A simple example

We illustrate our approach with the same running example as in Kahn [11].

6.1.1 Definition

We first draw in figure 4 the graphical scheme corresponding to the network.

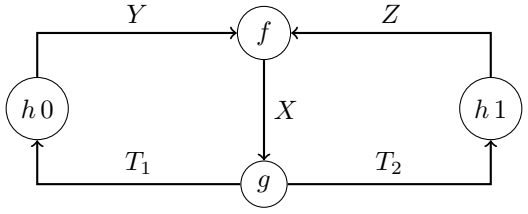

Figure 4: A simple example of Kahn network

The edges are X, Y, Z, T_1, and T_2. There is no input edge so the set I is defined as the empty type. The set J is an enumerated set with elements X, Y, Z, T_1, and T_2. All the links contain values in type `nat` such that the type family $A : I + J \to \mathsf{Type}$ is defined by $A\,k = \mathtt{nat}$ and we have $SA\,k = \mathtt{Str\,nat}$.

The functions corresponding to the nodes are f, g and h. The node g has two outputs corresponding to the functions g_1 and g_2. They satisfy the equations:

- $f\,U\,V = \mathtt{app}\,U\,(\mathtt{app}\,V\,(f\,(\mathtt{rem}\,U)\,(\mathtt{rem}\,V)))$
- $g_1\,U = \mathtt{app}\,U\,(g_1\,(\mathtt{rem}\,(\mathtt{rem}\,U)))$
- $g_2\,U = \mathtt{app}\,(\mathtt{rem}\,U)\,(g_2\,(\mathtt{rem}\,(\mathtt{rem}\,U)))$
- $h\,n\,U = \mathtt{cons}\,n\,U$

In Kahn [11], the equations involve an extra `first` application in the first argument of `app`, but because we proved: $\mathtt{app}\,(\mathtt{first}\,x)\,y == \mathtt{app}\,x\,y$, we can eliminate it.

In order to define these nodes, we use the fixpoint construction and the composition of the (continuous) functions `app`, `rem` and `cons` on streams.

The system itself (called `sys`) is translated from the scheme in the figure 4 which can also be seen as the set of equations:

$$X = f\,Y\,Z \quad Y = h\,0\,T_1 \quad Z = h\,1\,T_2 \quad T_1 = g_1\,X \quad T_2 = g_2\,X$$

In `Coq`, `sys` is a function from J to the continuous functions from $\prod SA \xrightarrow{C} \mathtt{Str\,nat}$. Given p of type $\prod SA$, we have:

- $\mathtt{sys}\,X\,p = f\,(p\,(r\,Y))\,(p\,(r\,Z))$

- $\texttt{sys}\, Y\, p = h\, 0\, (p\, (r\, T_1))$
- $\texttt{sys}\, Z\, p = h\, 1\, (p\, (r\, T_2))$
- $\texttt{sys}\, T_1\, p = g_1\, (p\, (r\, X))$
- $\texttt{sys}\, T_2\, p = g_2\, (p\, (r\, X))$

Now the resulting stream (called `result`) of type ($\texttt{Str}\,\text{nat}$) is obtained the following way:

1. The solution `sol` of the system `sys` has type $\prod SA_I \xrightarrow{C} \prod SA$.
2. Because I is empty, the type SA_I containts a trivial element `inp`, we apply `sol` to `inp` and get an object of type $\prod SA$.
3. The object $\texttt{sol}\,\texttt{inp}\,X$ that is the projection of the previous system on the link X is the expected result.

6.1.2 Properties

Kahn's paper proves that the result is an infinite stream containing alternatively 0 and 1. For that he proves: $\texttt{result} == \texttt{cons}\, 0\, (\texttt{cons}\, 1\, \texttt{result})$.

This is done in two steps, first he proves that `result` satisfies the following fixpoint equation:

$$\texttt{result} == \texttt{cons}\, 0\, (\texttt{cons}\, 1\, (f\, (g_1\, \texttt{result})\, (g_2\, \texttt{result})))$$

then proves that $f\, (g_1\, s)\, (g_2\, s) == s$.

The first equation is a consequence of two general properties of fixpoints:

1. A fixpoint on a continuous function is stable by composition: $\texttt{FIXP}\, f == \texttt{FIXP}\, (f@\_f) == \texttt{FIXP}\, f^{n+1}$.
   This is a consequence of a general lemma about fixpoint of composition of continuous functions:

   $$\forall (f\, g : D \xrightarrow{C} D),$$
   $$g@\_f \leq f@\_g \rightarrow f\, (\texttt{FIXP}\, g) \leq \texttt{FIXP} g \rightarrow \texttt{FIXP}(f@\_g) == \texttt{FIXP}\, g$$

2. Fixpoint on products can be simplified when the output on an index i depends only on the input on the same index i:

   $$\forall (I : \mathsf{Type})(D : I \rightarrow \texttt{cpo})(F : \textstyle\prod D \xrightarrow{C} \prod D)(i : I)(F_i : D\, i \xrightarrow{C} D\, i),$$
   $$(\forall p : \textstyle\prod D, F\, p\, i == F_i\, (p\, i)) \rightarrow \texttt{FIXP}\, F\, i == \texttt{FIXP}\, F_i.$$

We take the equation `EQN_sys` of type $\prod SA \xrightarrow{C} \prod SA$ associated with the system `sys` together with the empty input stream and we compose it three times. We obtain the equation:

$$\texttt{EQN\_sys}^3\, p\, (r\, X) = f\, (h\, 0\, (g_1\, (p\, (r\, X))))\, (h\, 1\, (g_2\, (p\, (r\, X)))).$$

Consequently the stream `result` which is the fixpoint of `EQN_sys` on the output X is also the fixpoint of `EQN_sys`^{3} on the output X and is also the fixpoint of F_X with $F_X\, s = f\,(h\,0\,(g_1\, s))\,(h\,1\,(g_2\, s))$. Using the definition of f and h, it is easy to see that: $F_X\, s == \texttt{cons}\,0\,(\texttt{cons}\,1\,(f\,(g_1\, s)\,(g_2\, s))$.

What remains to be proved is that $\forall s, f\,(g_1\, s)\,(g_2\, s) == s$. Kahn's paper uses a structural induction which is not appropriate because the stream s is possibly infinite. Instead we use a bisimulation technique.

We use a variation of the bisimulation principle given in section 4.2.4 which is:

$$
\begin{array}{l}
\forall D\,(R : \texttt{Str}\,A \to \texttt{Str}\,A \to \mathsf{Prop}),\\
(\forall x_1\, x_2\, y_1\, y_2, R\, x_1\, y_1 \to x_1 == x_2 \to y_1 == y_2 \to R\, x_2\, y_2)\\
\to (\forall x\, y, (\texttt{is\_cons}\, x \vee \texttt{is\_cons}\, y) \to R\, x\, y \to \texttt{first}\, x == \texttt{first}\, y)\\
\to (\forall x\, y, (\texttt{is\_cons}\,(\texttt{rem}\, x) \vee \texttt{is\_cons}\,(\texttt{rem}\, y)) \to\\
\qquad\qquad R\, x\, y \to \texttt{first}\,(\texttt{rem}\, x) == \texttt{first}\,(\texttt{rem}\, y))\\
\to (\forall x\, y, (\texttt{is\_cons}\,(\texttt{rem}\, x) \vee \texttt{is\_cons}\,(\texttt{rem}\, y)) \to\\
\qquad\qquad R\, x\, y \to R\,(\texttt{rem}\,(\texttt{rem}\, x))\,(\texttt{rem}\,(\texttt{rem}\, y)))\\
\to \forall x\, y, R\, x\, y \to x == y.
\end{array}
$$

We instantiate this principle by the relation $R\, s\, t \stackrel{\text{def}}{=} t == f\,(g_1\, s)\,(g_2\, s)$. The proof is based on algebraic properties of f, g_1, g_2, `rem`, and `first`.

We end up with the expected property $\texttt{result} = \texttt{cons}\,0\,(\texttt{cons}\,1\,\texttt{result})$ from which we deduce that `result` is an infinite stream because its length is infinite.

6.2 Sieve of Eratosthenes

The scheme corresponding to the sieve of Eratosthenes was given in figure 2 in section 2. What is interesting is that it is a recursive scheme. The scheme defines a node `sift` which is used as an internal node in the scheme itself. This is done using a fixpoint construction which is possible because the interpretation of a scheme is a continuous function of the nodes themselves.

The node `fdiv` is easily built using `case` and `filter` such that:

$$\texttt{fdiv}\,(\texttt{cons}\, a\, s) = \texttt{filter}\,(\texttt{div}\, a)\, s$$

We introduce the input index type I which is just a type with one element i and the output index type J which contains 3 elements X, Y and o. All the links have type `nat`, so using the same notation as before we introduce $A : I + J \to \mathsf{Type}$, $SA : I + J \to \texttt{cpo}$, $SA_I : I \to \texttt{cpo}$ such that $A\, k = \texttt{nat}$, $SA\, k = \texttt{Str}\,\texttt{nat}$ and $SA_I\, i = \texttt{Str}\,\texttt{nat}$.

We define the functional `Fsift` associated with the recursive system which has type $(\texttt{Str}\,\texttt{nat} \xrightarrow{C} \texttt{Str}\,\texttt{nat}) \xrightarrow{C} \texttt{system}\, A$ and is defined by case on the output link with p of type $\prod SA$:

$$
\begin{array}{l}
\texttt{Fsift}\, f\, X\, p = \texttt{fdiv}\,(p\,(l\, i))\\
\texttt{Fsift}\, f\, Y\, p = f\,(p\,(r\, X))\\
\texttt{Fsift}\, f\, o\, p = \texttt{app}\,(p\,(l\, i))\,(p\,(r\, Y))
\end{array}
$$

The construction SOL_of_system introduced in section 5.2 gives us a continuous function from systems to functions of type $\prod SA_I \xrightarrow{C} \prod SA$ from the streams corresponding to input edges to the streams corresponding to all edges.

The composition of SOL_of_system with Fsift gives us a continuous function from ($\mathtt{Str\,nat} \xrightarrow{C} \mathtt{Str\,nat}$) to $\prod SA_I \xrightarrow{C} \prod SA$.

Now the recursive graphical construction of the system says that sift is the functional corresponding to the input i and the output o.

Using $\mathtt{pair}_1$ (the continuous trivial function from D to $\prod_{i \in I} D$ when I has only one element), it is easy to build a continuous function focus of type $(\prod SA_I \xrightarrow{C} \prod SA) \xrightarrow{C} \mathtt{Str\,nat} \xrightarrow{C} \mathtt{Str\,nat}$ such that $\mathtt{focus}\,h\,s = h\,(\mathtt{pair}_1 s)\,o$.

The composition of focus, SOL_of_system and Fsift is now a continuous function from ($\mathtt{Str\,nat} \xrightarrow{C} \mathtt{Str\,nat}$) to ($\mathtt{Str\,nat} \xrightarrow{C} \mathtt{Str\,nat}$). We introduce sift as the fixpoint of this operator.

We can prove, using the fixpoint equation for sift, the following equality:

$$\mathtt{sift} == \mathtt{focus}\,(\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})).$$

Using the fixpoint equation for: $\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})$, it is easy to derive successively the following equalities for all stream s:

- $\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})\,(\mathtt{pair}_1\,s)\,(l\,i) == s$.
- $\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})\,(\mathtt{pair}_1\,s)\,(r\,X) == \mathtt{fdiv}\,s$.
- $\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})\,(\mathtt{pair}_1\,s)\,(r\,Y) == \mathtt{sift}\,(\mathtt{fdiv}\,s)$.
- $\mathtt{SOL\_of\_system}\,A\,(\mathtt{Fsift\,sift})\,(\mathtt{pair}_1\,s)\,(r\,o) == \mathtt{app}\,s\,(\mathtt{sift}\,(\mathtt{fdiv}\,s))$.

From these equalities, the property of fdiv, and the fixpoint equality for sift we easily deduce the expected property of sift:

$$\mathtt{sift}\,(\mathtt{cons}\,a\,s) == \mathtt{cons}\,a\,(\mathtt{sift}\,(\mathtt{filter}\,(\mathtt{div}\,a)\,s)) \qquad \mathtt{sift}\,\bot == \bot$$

7 Conclusion

7.1 Contributions

This paper describes three contributions.

The first one is a general Coq library for ω-cpos. This is a constructive version of cpos where there is an explicit function to build the least-upper bound of any monotonic sequence. It contains useful constructions such as generalized products, cpos of monotonic and continuous functions and combinators to manipulate them. It also introduces a fixpoint combinator. This library has also been used in a Coq library for modeling randomized programs as distributions [1, 2].

The second contribution is the definition of the cpo of streams of elements in a type A. This type, defined co-inductively, covers both the case of finite and infinite streams but without any explicit way to decide if the stream is finite or not. The type itself is not very original, it corresponds to an infinite stream with values which can be present or absent. What is interesting is the order defined on that data structure and the derived equality. Modulo this equality, we are able to reason on this data structure by only considering the interesting cases where the streams are equal to ($\texttt{cons}\, a\, s$). The most difficult construction on the type of streams was the least-upper bound. The fixpoint given by the cpo structure makes it possible to define in a natural way a function like `filter` that selects the elements of a stream satisfying a predicate P. This function is problematic with most representations of streams in Coq because the output can be finite or infinite depending on the number of elements in the input which satisfies P.

The last contribution is the modeling of Kahn networks as they are described in the paper [11]. We chose a shallow embedding where a system is represented using a set of links and for each noninput link a continuous function corresponding to the node. Each node can possibly take as input all the links of the system. This leads to a very simple and uniform definition of systems which itself can be seen as a cpo.

A system together with streams for inputs defines a continuous function on the streams associated with the links of the system (the history of the system). The fixpoint of this function defines the behavior of the system as a continuous function both from the inputs and the system.

Using this interpretation, we were able to formalize both the main example given by Kahn [11] and the sieve of Eratosthenes, an example of a recursive scheme which was presented by Kahn and MacQueen [12].

7.2 Remarks

Coq development The Coq development runs with Coq version 8.1 [15] currently under development. It makes an intensive use of the new setoid rewriting tactic. It is available from the author's web page. It contains approximately 1700 lines of definitions and statements and 3000 lines of proofs. The Coq notation mechanism as well as implicit arguments makes it possible to keep notations in Coq quite similar to the ones used in this paper.

What is mainly missing is a syntactic facility in order to automatically build complex continuous functions by composition of simpler functions and functionals.

Synchronous case. As we mentioned before, one important application of Kahn networks is their restriction to synchronous languages where no buffer is needed to store the values on the links. The nodes receive the inputs at a regular time given by a clock and instantaneously produce an output. A denotational

semantics of this calculus in Coq was given by Boulmé and Hamon [5]. Their approach is to make the type of the stream dependent on the clock (which is an infinite stream of boolean values), so there is a control on which data is available or not. They do that by considering ordinary infinite streams (only the `cons` constructor) with values which can either be an ordinary value in the type A of elements, or an absent value (when the clock is off) or a failure (no value when the clock is on).

We could adapt our development to this particular case by extending the definition of streams to make them dependent on the clock.

```
CoInductive Str A :clock → Type :=
  Eps : ∀ c, Str A c → Str A (false::c)
| cons : ∀ c, A → Str A c → Str A (true::c)
```

Then in order to define the bottom element of a stream on a clock c, it is convenient to have a bottom element in the type A. So the natural framework is to consider a cpo structure on A. Then the order on the streams can be defined the following way (the clock argument of constructors `Eps` and `cons` can be derived from the type of the stream so it is left implicit in the Coq notation).

$$\frac{x \leq y}{\texttt{Eps}\, x \leq \texttt{Eps}\, y} \qquad \frac{a \leq b \qquad x \leq y}{\texttt{cons}\, a\, x \leq \texttt{cons}\, b\, y}$$

It is simpler than in our case because we know exactly where the `cons` constructors are. The construction of lubs is also simplified, when the clock is true, there is a `cons` constructor in each element of the sequence of streams, we produce a `cons` constructor in the output with a value corresponding to the lub of the heads. However some extra properties have to be proved on the sequence if we want to ensure that there is no `cons` $\bot$ left in the result.

Equations. Kahn's paper refers to the paper of Courcelle, Kahn and Vuillemin [9] which proves the decidability of equivalence in a language of fixpoints. We started to write a Coq version of this paper and in particular we defined the notion of terms and built the cpo corresponding to the *canonical interpretation* of equations built on sequences of terms. But the full formalization of this part still remains to be done.

References

[1] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. In Tarmo Uustalu, editor, *Mathematics of Program Construction, MPC 2006*, volume 4014 of *Lecture Notes in Computer Science*, Kuressaare, Estonia, July 2006. Springer.

[2] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 2007. Extended version of [1].

[3] Yves Bertot. Filters on coinductive streams, an application to eratosthenes'sieve. In P. Urzyczyn, editor, *International Conference of Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2005.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[5] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, La Havana, Cuba, December 2001. Springer. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

[6] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2:1):1–28, 2005.

[7] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[8] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[9] Bruno Courcelle, Gilles Kahn, and Jean Vuillemin. Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. In Jacques Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 200–213. Springer, 1974. Translation from French by T. Veldhuizen with original text, a few comments and additional references.

[10] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*. North-Holland, 1974.

[12] Gilles Kahn and Dave MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*. North-Holland, 1977.

[13] Gilles Kahn and Gordon D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1& 2):187–277, 1993.

[14] Daniel Pilaud, Paul Caspi, Nicolas Halbwachs, and John Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Conference on Principles of Programming Languages*, pages 178–188, Munich, January 1987.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.