Circuits as streams in COQ: verification of a sequential multiplier.

Christine Paulin-Mohring

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Circuits as streams in Coq
Verification of a sequential multiplier

Christine Paulin-Mohring

September 1995

Research Report N$^{\mathrm{O}}$ 95-16

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00   Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Circuits as streams in Coq
Verification of a sequential multiplier

Christine Paulin-Mohring

September 1995

Abstract

This paper presents the proof of correctness of a multiplier circuit formalized in the Calculus of Inductive Constructions. It uses a representation of the circuit as a function from the stream of inputs to the stream of outputs. We analyze the computational aspect of the impredicative encoding of coinductive types and show how it can be used to represent synchronous circuits. We identify general proof principles that can be used to justify the correctness of such a circuit. The example and the principles have been formalized in the Coq proof assistant.

Keywords: Specification, Hardware Verification, Co-inductive definitions

Résumé

Cet article présente la preuve formalisée dans le Calcul des Constructions Inductives de la correction d'un circuit réalisant la multiplication sur les entiers. Le circuit est représenté par une fonction transformant la suite infinie d'entrées en une suite infinie de sorties. Nous analysons l'aspect calculatoire de la représentation imprédicative des définitions co-inductives et montrons comment cette représentation peut servir à coder un circuit synchrone. Nous identifions des principes de preuve généraux pour justifier de tels circuits. Les exemples et les principes ont été formalisés dans l'assistant à la démonstration Coq.

Mots-clés: Specification, Vérification de Matériel, Définition Co-inductives

Circuits as streams in Coq
Verification of a sequential multiplier

Christine Paulin-Mohring *
LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon cedex 07, France

September 12, 1995

1 Introduction

1.1 Motivations

General theorem provers such that NqThm [1, 2] or HOL [7] have been investigated in the domain of hardware verification. They are useful for doing abstract reasoning. A few investigations have been done in this area using the Coq theorem prover.

When reasoning about a circuit, we need first to choose a certain view of it, corresponding to the level of abstraction we are interested in. For a certain level of abstraction we need to choose a mathematical representation and also an implementation of it in a particular theorem prover. NqThm manipulates mainly functions, while HOL is a logical system in which one easily represents relations. Coq implements both a programming language on which computation can be done and a logical language in which one defines and reasons about relations. We try to take advantage of these features to get more natural proofs.

S. Coupet and L. Jakubiec have first investigated proving simple circuits in Coq (factorial, and the multiplier studied here). After discussion with them about the representation of circuits in various theorem provers, it came out that interpreting a circuit as a transformer of streams could give new interesting proof schemes. This paper investigates this area.

The system Coq now provides primitive co-inductive definitions [5, 6] but at that time, it was only possible to encode these infinite structures using an impredicative encoding. The encoding of co-inductive types in Girard-Reynolds second-order lambda-calculus was described in [10] and also used in a previous experiment proving Eratosthenes Sieve [9]. In this paper we choose a representation of co-inductive types as greatest fixpoints using types defined by constructors and higher-order quantification. We insist on the computational aspect of this representation which seems particularly well suited for the representation of circuits.

1.2 Outline

The remaining part of this section is devoted to the introduction of Coq notations used in this paper. The section 2 gives a brief presentation of the impredicative representation of infinite objects in type theory. We emphasize the concrete aspect of this representation as a process. In section 3 we show how to represent a generic sequential circuit specified by the type of inputs,

*This research was partly supported by ESPRIT Basic Research Action "Types" and by the GDR "Programmation" co-financed by MRE-PRC and CNRS.

outputs and registers, and both the output and updating functions. We derive proof principles using invariants for this circuit. In section 4, a circuit is formalized, specified and finally proven using the methodology previously described. This circuit implements a multiplier and was taken as an example by M. Gordon [7] for the HOL theorem prover also studied in Coq [4] using a representation of the circuit by a primitive recursive function.

These developments have been formalized using the Coq proof assistant and are available with the Coq distribution as a contribution.

1.3 Notations

The Calculus of Inductive Constructions which is the theoretical basis of the Coq system [3, 8] is an higher-order typed lambda-calculus that is used both for the representation of functions, propositions and proofs. It is not our purpose here to give a general presentation of the calculus but we shall give an informal understanding of the constructions that will be used in this paper.

1.3.1 Terms and Types

The calculus manipulates terms and types.

Sorts : Set and Prop. The types are special objects of the calculus. They can be interpreted both as ordinary data-types or as logical propositions using the well-known Curry-Howard isomorphism. In that case a term inhabiting the type witnesses a proof of the proposition.

The judgment $A : \mathsf{Set}$ will represent the fact that the type A is well-formed, while the judgment $A : \mathsf{Prop}$ represents the fact that A is a well-formed logical formula.

A type can be abstracted or applied to terms in order to represent predicates or type families.

Types. Atomic type families are either variables or concrete types specified by a set of constructors (also called *inductive types*).

Composed types are built using quantification $(x : A)B$. In case x does not occur in B, this quantification may be written $A \to B$.

The quantification can be read from different ways. If both A and B are data-types, $A \to B$ represents the type of functions from A to B. If both A and B are propositions then $A \to B$ represents the proposition "A implies B". If A is a data-type and B is a proposition then $(x : A)B$ represents the proposition "for all x of type A, B". The variable x may also be a type or predicate variable in which case, A represents its arity and we get an higher-order quantification like in $(A : \mathsf{Set})A \to A$.

Terms. Terms are built from variables, using application and abstraction. The application of the term t to the term u is written $(t\ u)$ with $(t\ u_1 \ldots u_k)$ representing $(\ldots(t\ u_1)\ldots u_k)$. The abstraction of the term t with respect to the variable x of type A is written $[x : A]t$ with $[x_1, \ldots x_k : A]t$ representing $[x_1 : A]\ldots[x_k : A]t$ and $[x_1, \ldots, x_k]t$ representing $[x_1 : A_1]\ldots[x_k : A_k]t$ when the types of the variables are clear from the context.

The constructors of a concrete type are terms corresponding to the introduction rules of the corresponding proposition. There is a generic construction representing the elimination rule written $<P>\mathsf{Case}\ x\ \mathsf{of}\ f_1 \ldots f_n\ \mathsf{end}$. It corresponds to a definition by case analysis. The term x should be in a concrete type specified by n constructors. The whole expression has type P (or more generally an instance of P given by x). Each term f_i represents how to build a justification of P in the case x starts with the i-th constructor c_i. The expression $<P>\mathsf{Case}\ (c_i\ a_1 \ldots\ a_k)\ \mathsf{of}\ f_1 \ldots f_n\ \mathsf{end}$ is intensionally equal to $(f_i\ a_1 \ldots a_k)$.

The language contains the possibility to define a function by structural recursion, but this is not strictly needed in our development, so we shall not give more details on this aspect.

1.3.2 Examples

Representation of data-types We first define the type *unit* with only one element *tt*. Then we define the type of booleans and the type of unary natural numbers.

Inductive $unit$: Set := tt : $unit$.
Inductive $bool$: Set := $true$: $bool$ | $false$: $bool$.
Inductive nat : Set := O : nat | S : $nat \rightarrow nat$.

Sum and product It is possible to define the disjoint sum and the product of two data-types using concrete type definition. These types are parameterized by two types variables A and B.

Inductive sum $[A, B$: Set$]$: Set := inl : $A \rightarrow (sum\ A\ B)$
$|inr$: $B \rightarrow (sum\ A\ B)$.
Inductive $prod$ $[A, B$: Set$]$: Set := $pair$: $A \rightarrow B \rightarrow (pair\ A\ B)$.

We shall use the following notations:

$A + B$	$(sum\ A\ B)$
$A{*}B$	$(prod\ A\ B)$
(a, b)	$(pair\ A\ B\ a\ b)$
$(f\ u_1 \ldots u_k, g\ v_1 \ldots v_k)$	$((f\ u_1 \ldots u_k), (g\ v_1 \ldots v_k))$
$A{*}B{*}C$	$A{*}(B{*}C)$
(a, b, c)	$(a, (b, c))$

Terms defined by case analysis Using the Case operator, it is easy to define for instance, the predecessor function, the *If* functional doing case analysis of booleans or the two projections for products.

Definition $pred$: $nat \rightarrow nat$:= $[n]$<nat>Case n of O $[p : nat]p$ end.
Definition If : $(C$: Set$)bool \rightarrow C \rightarrow C \rightarrow C$:= $[C, b, x, y]$<C>Case b of $x\ y$ end.
Definition fst : $(A, B$: Set$)A{*}B \rightarrow A$:= $[A, B, p]$<A>Case p of $[x, y]x$ end.
Definition snd : $(A, B$: Set$)A{*}B \rightarrow B$:= $[A, B, p]$<B>Case p of $[x, y]y$ end.
Definition trd : $(A, B, C$: Set$)A{*}B{*}C \rightarrow C$:= $[A, B, C, p](snd\ (snd\ p))$.

2 Representation of infinite objects

2.1 Encoding of infinite objects

One way to represent infinite objects in a strongly typed language uses the proof of existence of greatest fixed points for monotonic operators on types.

Formally we do the following construction. Let F be a type transformer, such as for any type X, $(F\ X)$ is a type. We assume F is a monotonic operator, it means that for each term f of type $A \rightarrow B$ one can build a term $(Fmon\ f)$ of type $(F\ A) \rightarrow (F\ B)$. This construction can be automatically computed if X occurs only positively in $(F\ X)$.

2.1.1 Greatest fixed points in Coq

Building the greatest fixed point of F corresponds to finding a type nu for which we have an object Out of type $nu \rightarrow (F\ nu)$ and an object $Intro$ of type $(F\ nu) \rightarrow nu$. These two operators witnesses the fact that nu is a fixed point. We require also the existence of an object $CoIter$ of type $(X \rightarrow (F\ X)) \rightarrow X \rightarrow nu$ representing the fact that nu is a greatest fixed point (actually post-fixed point of F). A possible representation of nu in Coq is the following :

Inductive $nu : \mathsf{Set} := \ \ CoIter : (X : \mathsf{Set})(X \to (F\ X)) \to X \to nu.$

A closed normal object of this type can be written $(CoIter\ A\ f\ x)$ with $A : \mathsf{Set},\ f : A \to (F\ A)$, and $x : A$. This type can be seen as an encoding of the second-order existential quantifier $\exists X : \mathsf{Set}.(X \to (F\ X)) \wedge X$. We shall give a more precise computational interpretation of this type in the section 2.3.

From this definition, we get directly the operator *CoIter* with the expected type.

We get also the following elimination principles as particular cases of the general elimination pattern for inductive types. The first one says that any object m is essentially built from a type X, a function f with type $X \to (F\ X)$ and an object x with type X, such that in order to prove $(P\ m)$ it is enough to prove $(P\ (CoIter\ X\ f\ x))$. The second one is similar but seen from the computational point of view: from m one can build an object in a data P by using the above X, f and x.

$$\frac{m : nu \quad P : nu \to \mathsf{Prop} \quad H : (X : \mathsf{Set})(f : X \to (F\ X))(x : X)(P\ (CoIter\ X\ f\ x))}{<P>\mathsf{Case}\ m\ \mathsf{of}\ H\ \mathsf{end} : (P\ m)}$$

$$\frac{m : nu \quad P : \mathsf{Set} \quad H : (X : \mathsf{Set})(X \to (F\ X)) \to X \to P}{<P>\mathsf{Case}\ m\ \mathsf{of}\ H\ \mathsf{end} : P}$$

The operator Case enjoys the following computational behavior :

$$<P>\mathsf{Case}\ (CoIter\ X\ f\ x)\ \mathsf{of}\ H\ \mathsf{end} \leadsto (H\ X\ f\ x)$$

The operators *Intro* and *Out* can be deduced using the following terms :

Definition $Out\ :\ nu \to (F\ nu)\ :=$
$[m]\ <(F\ nu)>\ \mathsf{Case}\ m\ \mathsf{of}$
$\quad [X : \mathsf{Set}][f : X \to (F\ X)][x : X](Fmon\ (CoIter\ X\ f)\ (f\ x))$
$\mathsf{end}.$
Definition $Intro\ :\ (F\ nu) \to nu\ :=\ (CoIter\ (F\ nu)\ (Fmon\ Out)).$

2.2 Streams

A typical example of a type built this way is the type Str_A of streams (infinite lists of objects in a given type A). It is obtained with the operator $F \equiv [X : \mathsf{Set}](A * X)$.

In that case, the function *Fmon* can be defined as :

Definition $Fmon\ :\ (X, Y : \mathsf{Set})(X \to Y) \to (A * X) \to (A * Y)\ :=\ [X, Y, f, p](fst\ p, f\ (snd\ p)).$

From the function *Out* of type $Str_A \to A{*}Str_A$ and the projections, we get easily the two functions $Hd : Str_A \to A$ and $Tl : Str_A \to Str_A$ giving respectively the head and tail of a stream. We can also derive a more convenient operator for constructing streams :

Definition $StrIt\ :\ (X : \mathsf{Set})(X \to A) \to (X \to X) \to X \to Str_A\ :=$
$[X, h, t, x](CoIter\ X\ [y : X](h\ y, t\ y)\ x).$

The following computational rules hold :

$$(Hd\ (StrIt\ X\ h\ t\ x)) \leadsto (h\ x) \qquad (Tl\ (StrIt\ X\ h\ t\ x)) \leadsto (StrIt\ X\ h\ t\ (t\ x))$$

2.3 Concrete representation of coinductive constructions

We explain now the computational aspect of this representation of infinite objects.

As we said before, a closed normal term of type *nu* is equal to $(CoIter\ X\ f\ x)$. It means that it is a structure with three elements: a type X, an object x of type X and a function f of type $X \rightarrow (F\ X)$.

We can represent this object with a picture :

$x : X$
$f : X \rightarrow (F\ X)$

We call this object a process, X is the type of the state variable whose value is x and f is the transformation function that can give raise to new processes built on the same type and to various "observational" values. This type behaves like an abstract data type, which means that if we have an object s of type Str_A we know it has the form $(CoIter\ X\ f\ x)$ for some arbitrary type X but we cannot access this type. In particular when we build from s an object in a type T, this type T cannot mention X.

2.3.1 Pictorial specification of streams

In case of the type of streams, the *Hd* and *Tl* functions can be represented the following way :

$$\begin{array}{c} \begin{array}{|c|}\hline x : X \\ \hline f : X \rightarrow A * X \\ \hline \end{array} \\ Hd \downarrow \\ \boxed{(fst\ (f\ x)) : A} \end{array} \quad \xrightarrow{Tl} \quad \begin{array}{|c|}\hline (snd\ (f\ x)) : X \\ \hline f : X \rightarrow A * X \\ \hline \end{array}$$

2.3.2 Other coinductive types

Infinite integers Assume F is $[X : \mathsf{Set}](unit + X)$ then $Nw = (nu\ F)$ represents the type of possibly infinite integers.

Given a finite integer n of type *nat* one can represent the corresponding infinite integer by the process :

$n : nat$
$[x]\langle unit + nat\rangle \mathsf{Case}\ x\ \mathsf{of}\ (inl\ tt)\ inr\ \mathsf{end} : nat \rightarrow unit + nat$

The infinite integer can be represented by the simple process :

$tt : unit$
$inl : unit \rightarrow unit + unit$

The *Out* function gives from an object in Nw an object in $unit + Nw$ representing the predecessor.

When this object is a left injection, it means that the process represents 0 and taking the predecessor has the effect to end the process, when it is a right injection we got the process representing the predecessor.

Pictorially we have one of the two situations :

$$\begin{array}{|c|}\hline x : X \\ \hline p : X \rightarrow unit + X \\ \hline \end{array} \longrightarrow () \qquad \text{when } (p\ x) = (inl\ tt)$$

$$\begin{array}{|c|}\hline x : X \\ \hline p : X \rightarrow unit + X \\ \hline \end{array} \longrightarrow \begin{array}{|c|}\hline y : X \\ \hline p : X \rightarrow unit + X \\ \hline \end{array} \qquad \text{when } (p\ x) = (inr\ y)$$

Infinite binary trees Assume F is $[X : \mathsf{Set}](A * X * X)$ the type $Trw = (nu\ F)$ represents the type of infinite binary trees. The Out function gives from an object in Trw an object in $A * Trw * Trw$ built from the label in the node and the left and right sons of the tree.

More computationally, applying an Out step to an object in Trw raises the label of type A plus two new processes of the same sort.

$$\begin{array}{ccc} & \begin{array}{|c|}\hline x : X \\ \hline p : X \to A * X * X \\ \hline \end{array} \longrightarrow \boxed{a : A} & \\ & \downarrow \quad \downarrow & \text{when } (p\ x) = (a, l, r) \\ \begin{array}{|c|}\hline l : X \\ \hline p : X \to A * X * X \\ \hline \end{array} & \begin{array}{|c|}\hline r : X \\ \hline p : X \to A * X * X \\ \hline \end{array} & \end{array}$$

2.4 Co-iteration vs Co-recursion

We can remark that the Out step applied to an object of type $M \equiv (nu\ F)$ seen as a process produces a composite object in which may appear one or several objects of type M which are processes sharing the same implementation than the original object. It means that the type X of the implementation and the transformation function are the same. Only the state, that is the particular value of type X changes.

If we see a stream as a process then any tail of the stream will represent the same process but at various stages of its life.

Sometimes this only way to build streams is too rigid. For instance, how can we build the function for the concatenation of an element a of type A in front of a stream s ?
We want the first Out step to give us the pair (a, s) and then the next Out steps to behave like the Out steps of s.

Using the $CoIter$ operator, one can implement the concatenation function by adding a boolean information for the identification of the first step. The following stream implements the concatenation of a to s:

$$\begin{array}{|c|}\hline (true, s) : bool * Str_A \\ \hline [x](If\ (fst\ x)\ (a, false, s)\ (Hd\ (snd\ x), false, Tl\ (snd\ x))) : bool * Str_A \to A * bool * Str_A \\ \hline \end{array}$$

but it does not look like a very efficient implementation because each step tests whether it is the first one...

One may prefer to use a more powerful scheme $CoRec$ known as co-recursion which has type $(X : \mathsf{Set})(X \to A * (Str_A + X)) \to X \to Str_A$.

If a stream s is built from $(CoRec\ X\ f\ x)$ then $(f\ x)$ has type $A * (Str_A + X)$
If $(f\ x)$ is $(a, inl\ s')$ with $s' : Str_A$, we expect $(Tl\ s)$ to be s'. If $(f\ x)$ is $(a, inr\ y)$ with $y : X$, we expect $(Tl\ s)$ to be $(CoRec\ X\ f\ y)$.

Computationally, it means that the transformation step may not only modify the current value of the state like in the iterative case, but instead it may provide a new process built on a new implementation.

Pictorially, if a stream defined as $(CoRec\ X\ f\ x)$ is represented by

$$\begin{array}{|c|}\hline x : X \\ \hline f : X \to A * (Str_A + X) \\ \hline \end{array}$$

we have one of the two following situations :

$$\begin{array}{|c|}\hline x : X \\ \hline f : X \to A * (Str_A + X) \\ \hline \end{array} \xrightarrow{Tl} \boxed{s : Str_A} \qquad \text{when } (snd\ (f\ x)) = (inl\ s)$$

$$\begin{array}{|c|}\hline x : X \\ \hline f : X \to A * (Str_A + X) \\ \hline \end{array} \xrightarrow{Tl} \begin{array}{|c|}\hline y : X \\ \hline f : X \to A * (Str_A + X) \\ \hline \end{array} \qquad \text{when } (snd\ (f\ x)) = (inr\ y)$$

The *cons* operation becomes trivial when using the co-recursion scheme. Given $a : A$ and $s : Str_A$ it can be implemented efficiently as:

$tt : unit$
$[x : unit](a, inl\ s) : unit \rightarrow A * (Str_A + unit)$

General co-recursion More generally, for an arbitrary functor F the type of the recursion scheme is :

$$CoRec : (X : \mathsf{Set})(X \rightarrow (F\ (nu + X))) \rightarrow X \rightarrow nu$$

As was noticed by H. Geuvers, one can easily build a coinductive type enjoying a co-recursion scheme instead of a co-iteration scheme :

Inductive $nur : \mathsf{Set} := \ CoRec : (X : \mathsf{Set})(X \rightarrow (F\ (nur + X))) \rightarrow X \rightarrow nur.$

This approach has the drawback that our inductive definition mechanism should accept the occurrence of nur to be positive in $(F\ (nur + X))$.

With this definition we can easily build the $Outr$ function.

Definition $Outr\ :\ nur \rightarrow (F\ nur)\ :=$
$[m]$ <$(F\ nur)$> $\mathsf{Case}\ m\ \mathsf{of}$
$[X : \mathsf{Set}][f : X \rightarrow (F\ nur + X)][x : X]$
$(Fmon\ [z : nur + X]$<nur>$\mathsf{Case}\ z\ \mathsf{of}\ [m : nur]m\ [y : X](CoRec\ X\ f\ y)\ \mathsf{end}$
$(f\ x))$
$\mathsf{end}.$

Consequently the following reduction trivially holds :

$$(Outr\ (CoRec\ X\ f\ x)) \rightsquigarrow (Fmon\ [z : nur{+}(F\ nur)]<nur>\mathsf{Case}\ z\ \mathsf{of}\ [m]m\ [y](CoRec\ X\ f\ y)\ \mathsf{end}\ (f\ x))$$

One can notice that we only make use of the existence of the Case operator for the type nur, it means that we do not use the fact that it is a least fixed point in order to build the $Outr$ function. This representation provides also an easy way to program the $Intror$ function.

Definition $Intror\ :\ (F\ nur) \rightarrow nur\ :=\ [m](CoRec\ (F\ nur)\ [n : (F\ nur)](Fmon\ inl\ n)\ m).$

Furthermore we get, assuming $(Fmon\ (f \circ g)) = (Fmon\ f) \circ (Fmon\ g)$ and $(Fmon\ [x : X]x) = [x : (F\ X)]x$ the fact that $(Outr\ (Intror\ m))$ is convertible with m.

$$\begin{aligned}
(Outr\ (Intror\ m)) &= (Fmon\ [z : nur + X]<nur>\mathsf{Case}\ z\ \mathsf{of}\ [m : nur]m\ Intror\ \mathsf{end}\ (Fmon\ inl\ m)) \\
&= (Fmon\ [z : nur]<nur>\mathsf{Case}\ (inl\ z)\ \mathsf{of}\ [m : nur]m\ Intror\ \mathsf{end}\ m) \\
&= (Fmon\ [z : nur]z\ m) \\
&= m
\end{aligned}$$

We shall not use this type in our encoding of circuits for which the iterative representation is computationally more relevant.

Anyway it is well-known that a kind of co-recursion operator can be mimicked with the iterative version of coinductive types. Given $X : \mathsf{Set}$, $f : X \rightarrow (F\ (nu + X))$ and $x : X$, an object of type nu representing an object defined by co-recursion $(CoRec\ X\ f\ x)$ can be implemented as :

$(inr\ x) : nu + X$
$[z]<F\ (nu + X)>\mathsf{Case}\ z\ \mathsf{of}\ [m](Fmon\ inl\ (Out\ m))\ f\ \mathsf{end} : (nu + X) \rightarrow (F\ (nu + X))$

But this operator does not enjoy exactly the expected reduction rules. The corresponding equalities are only provable in an extensional way (we can only prove that the two streams generates equal values).

2.4.1 Streams versus functions

Obviously there is a correspondence between streams of elements of a type A and functions from nat to A. It is easy to build a function nth which takes an integer n and associates to an arbitrary stream the n-th element of this stream.

We first define iteratively the function which takes the n-th tail of a stream.

$$(nthtl\ s\ O) = s \qquad (nthtl\ s\ (S\ n)) = (Tl\ (nthtl\ s\ n))$$

Then we define the function which picks the n-th element of the stream by

$$(nth\ s\ n) = (Hd\ (nthtl\ s\ n))$$

Reciprocally, given a function f there is a uniform way to build a stream s such that $(nth\ s\ n)$ reduces to $(f\ n)$ for instance : $(StrIt\ nat\ f\ S\ O)$.

But obviously, the two representations does not have the same computational behavior. The computation of the n-th value of s using an eager evaluation always computes the sequence $(f\ 0)\ldots(f\ n-1)$ which may not be very efficient. On the other side, assume f is defined in a primitive recursive way, $((f\ 0) = x_0\quad (f\ n+1) = (g\ n\ (f\ n)))$ such that the computation of $(f\ n)$ takes n steps. In order to compute the sequence $(f\ 0)\ldots(f\ n-1)$ with a functional representation it will take n^2 steps. But if we choose a clever stream representation as

$$(CoIter\ A{*}nat\ [na : A{*}nat](fst\ na, g\ (snd\ na)\ (fst\ na), S\ (snd\ na))\ (x_0, O))$$

then the cost of the computation of the sequence will be linear.

Clearly the co-iterative representation of streams is closer to the physical representation of circuits. Our purpose will be to use this representation internally in order to reason about circuits in Coq.

3 Circuits

We shall now describe the representation of a circuit as a stream transformer. In that case, streams defined using the co-iteration principle suits perfectly.

3.1 Specification of a sequential circuit

When we are describing a circuit, we have to choose the level of representation. The circuit realizes a function from the set of inputs to the set of outputs. When we have a combinational circuit, the function which is realized depends only on the structure of the circuit.

When the circuit contains registers (sequential circuit), the output is computed from the inputs and the current value of registers, the new value of registers is also obtained from the old values of registers and the current value of inputs. So the function which is realized depends in general on the value of the registers. The value of the registers is itself a function which depends on the structure of the circuit, the initial value of the register and the finite list of previous values of inputs. One way to represent the function realized by a synchronous sequential circuit is to add as an extra parameter an integer n representing the current stage of the circuit.

From the structure of the circuit we can deduce two functions one (called *output*) computing the output from the input and registers, the other one (called *update*) updating the registers from the inputs and current values of registers. Let us call TI the type of inputs, TO the type of outputs and TR the type of registers, we have $output : TI \rightarrow TR \rightarrow TO$ and $update : TI \rightarrow TR \rightarrow TR$.

Circuits as functions It is possible to represent the inputs as a function $input : nat \to TI$. Assume the initial value of registers is r_0, we can define a function $register : nat \to TR$ representing the value of registers at each time and finally the function $circuit : nat \to TO$ representing the value of outputs. These functions can be defined in a primitive recursive way by :

$$(register\ 0) = r_0 \quad (register\ (S\ n)) = (update\ (input\ n)\ (register\ n))$$
$$(circuit\ n) = (output\ (input\ n)\ (register\ n))$$

This approach is taken for the verification of the multiplier circuit in Coq done by S. Coupet and L. Jakubiek [4].

3.2 Representing a circuit as a stream transformer

In this paper we choose another approach namely to represent the circuit as a function from the stream of inputs to the stream of outputs whose implementation makes reference to the type of registers.

More precisely the previous circuit will be represented as a process built on the type $Str_{TI}*TR$. Assume the current state is a pair (s, r), the process will first consume the stream of inputs s to produce the current input i and the stream of remaining inputs t, the output will be $(output\ i\ r)$ and the next value of the state will be $(t, (update\ i\ r))$.

This can be represented pictorially the following way :

$(si, ri) : Str_{TI} * TR$
$[sr]<A*Str_{TI}*TR>$ Case sr of $[s,r](output\ (Hd\ s)\ r, Tl\ s, update\ (Hd\ s)\ r)$ end

Definition 1 *The* Coq *code for a circuit of entry type TI, output type TO, updating function update and output function output is the following :*

$$
\begin{array}{l}
\textbf{\textit{Definition}}\ \text{circ}\ :\ TR \to Str_{TI} \to Str_{TO}\ := \\
\quad [ri, si](CoIter\ \ Str_{TI} * TR \\
\qquad\qquad [sr]<TO*Str_{TI}*TR>\ \ \textsf{\textit{Case}}\ sr\ \textsf{\textit{of}} \\
\qquad\qquad\qquad\qquad [s,r](output\ (Hd\ s)\ r,\ Tl\ s, update\ (Hd\ s)\ r) \\
\qquad\qquad\qquad\quad \textsf{\textit{end}} \\
\qquad\qquad (si, ri)).
\end{array}
$$

3.3 Reasoning on circuits

Clearly this particular representation suggests also particular proof methods for reasoning on circuits.

One property which has to be checked for circuits is "given two circuits, prove that they realize the same relation between inputs and outputs". Usually one circuit represents the implementation to be checked and the other one the specification which is another implementation using a less efficient but more comprehensible circuit. The drawback of this kind of verification is that the specification has to be given as a circuit which can itself contains errors. Another kind of verification can be to check that a circuit satisfies a certain logical property.

Usually, assume we have a circuit specified by the functions $output$ and $update$ as before. Let us call $circ$ the same function of type $TR \to Str_{TI} \to Str_{TO}$ as defined above in definition 1. Given an input stream I and an initial value for register R, we denote by $CIRC$ the object of type Str_{TO} build as $(circ\ R\ I)$. We want to prove that a certain relation holds on outputs that will depend on the stream input I and also on a time parameter. From now on we write $s[n]$ instead of $(nth\ s\ n)$. We assume given a property $Q : nat \to TO \to \mathsf{Prop}$. And we expect to prove:

$$\forall n : nat.(Q\ n\ CIRC[n])$$

This property can be proven, as an instance of a more general scheme applicable to any iteratively defined function.

3.4 Properties of iteratively defined functions

Assume we have a type X, a function f of type $X \to X$, and x of type X, one can define a function *iter* of type $nat \to X$ such that $(iter\ n)$ iterates n times f from x.

Let Q be a property of type $nat \to X \to \mathsf{Prop}$, we are interested by proving two kinds of properties of Q with respect to *iter*. The first one is $\forall n : nat.(Q\ n\ (iter\ n))$ (written in Coq as $(n : nat)(Q\ n\ (iter\ n)))$ and the second one is $\exists n : nat.(Q\ n\ (iter\ n))$ (written in Coq as $(Ex\ [n : nat](Q\ n\ (iter\ n))))$

Both can be proven using the existence of an invariant *Inv* with type $nat \to X \to \mathsf{Prop}$.

We now give the precise lemmas.

Lemma 1 *If one can find* $Inv : nat \to X \to \mathsf{Prop}$*, such that the following is provable :*

$$(n : nat)(y : X)(Inv\ n\ y) \to (Q\ n\ y) \wedge (Inv\ (S\ n)\ (f\ y))$$
$$(Inv\ O\ x)$$

then there is a proof of $(n : nat)(Q\ n\ (iter\ n))$.

Proof: One first prove $(n : nat)(Inv\ n\ (iter\ n))$ by induction on n and the result follows immediately.

□

Lemma 2 *If one can find* $Inv : nat \to X \to \mathsf{Prop}$, $Rel : nat * X \to nat * X \to \mathsf{Prop}$ *such that the following is provable :*

$(Acc\ Rel\ (O, x))$ *(ie there is no infinite decreasing sequence for Rel starting from* (O, x)*)*
$$(n : nat)(y : X)(Inv\ n\ y) \to (Q\ n\ y) \vee ((Inv\ (S\ n)\ (f\ y)) \wedge (Rel\ (S\ n, f\ y)\ (n, y)))$$
$$(Inv\ O\ x)$$

then there is a proof of $(Ex\ [n : nat](Q\ n\ (iter\ n)))$.

One first prove $(p : nat)(Acc\ Rel\ (p, x)) \to (Inv\ p\ x) \to (Ex\ [n : nat](Q\ (plus\ p\ n)\ (iter\ n)))$ by well-founded induction on (p, x) from which the result follows.

□

Remark The fact that *nat* is involved in the well-founded relation may seem unnecessarily complicated. It is actually very useful, for instance in order to express that the object of type X will decrease only after a finite number of steps.

3.5 Application to streams and circuits

3.5.1 Universal properties

Lemma 3 *Let* Q *be a relation of arity* $nat \to A \to \mathsf{Prop}$*, and* s *a stream of type* Str_A*. If there exists* Inv *which has type* $nat \to Str_A \to \mathsf{Prop}$ *such that the following property holds:*

$$(n : nat)(s : Str_A)(Inv\ n\ s) \to (Q\ n\ (Hd\ s)) \wedge (Inv\ (S\ n)\ (Tl\ s))$$
$$(Inv\ O\ s)$$

then we have : $(n : nat)(Q\ n\ s[n])$.

Proof: It is just the lemma 1 with the function *Tl* for the iterated function and the predicate $[n : nat][s : Str_A](Q\ n\ (Hd\ s))$.

□

Invariant on implementation If we know the implementation of the stream, then we can derive a more precise principle using an invariant on the implementation itself.

Lemma 4 *Let Q be a relation of arity* $\mathrm{nat} \rightarrow A \rightarrow \mathsf{Prop}$*. Let X be a type, f be a function with type $X \rightarrow A * X$ and x_0 an element of type X. If there exists* Inv *which has type* $\mathrm{nat} \rightarrow X \rightarrow \mathsf{Prop}$ *such that the following property holds:*

$$(n : \mathrm{nat})(x : X)(\mathrm{Inv}\ n\ x) \rightarrow (Q\ n\ (\mathrm{fst}\ (f\ x))) \wedge (\mathrm{Inv}\ (S\ n)\ (snd\ (f\ x)))$$
$$(\mathrm{Inv}\ O\ x_0)$$

then we have : $(n : \mathrm{nat})(Q\ n\ (CoIter\ X\ f\ x_0)[n])$

PROOF: It is still the application of lemma 1 with the iterated function $[x : X](snd\ (f\ x))$ and the predicate $[n : nat][x : A](Q\ n\ (\mathrm{fst}\ (f\ x)))$.
□

Invariant on a circuit In the case of a circuit we furthermore can use the properties :

$$(Hd\ (circ\ s\ r)) = (output\ (Hd\ s)\ r) \qquad (Tl\ (circ\ s\ r)) = (circ\ (Tl\ s)\ (update\ (Hd\ s)\ r))$$

Corollary 4.1 *If there exists an invariant* inv *which has type* $\mathrm{nat} \rightarrow Str_{TI} \rightarrow TR \rightarrow \mathsf{Prop}$ *such that the following properties hold:*

$$(n : \mathrm{nat})(s : Str_{TI})(r : TR)$$
$$\quad (inv\ n\ r) \rightarrow (Q\ n\ (output\ (Hd\ s)\ r)) \wedge (inv\ (S\ n)\ (Tl\ s)\ (update\ (Hd\ s)\ r))$$
$$(inv\ O\ I\ R)$$

then he have: $(n : \mathrm{nat})(Q\ n\ CIRC[n])$

PROOF: We apply lemma 4 with $X = Str_{TI} * TR$, $x_0 = (I, R)$ and the invariant $[n : nat][x : Str_{TI} * TR](inv\ n\ (\mathrm{fst}\ x)\ (snd\ x))$.
□

We can also use the fact that the stream of inputs is the input stream I at time n.

Corollary 4.2 *If there exists an invariant* inv *which has type* $\mathrm{nat} \rightarrow TR \rightarrow \mathsf{Prop}$ *such that the following properties hold:*

$$(n : \mathrm{nat})(r : TR)(inv\ n\ r) \rightarrow (Q\ n\ (output\ I[n]\ r)) \wedge (inv\ (S\ n)\ (update\ I[n]\ r))$$
$$(inv\ O\ R)$$

then we can prove: $(n : \mathrm{nat})(Q\ n\ CIRC[n])$

PROOF: We apply the previous corollary with the invariant: $[n : nat][s : Str_{TI}][r : TR](s = (nthtl\ I\ n)) \wedge (inv\ n\ r)$
□

3.5.2 Existential properties

We can apply the lemma 2 to various instances in order to get proofs that the property Q will be reached. We only give here the counterpart of the lemma 4.2.

Lemma 5 *If there exists an invariant* inv *which has type* $\mathrm{nat} \rightarrow TR \rightarrow \mathsf{Prop}$ *and a relation* Rel *with type* $\mathrm{nat} * TR \rightarrow \mathrm{nat} * TR \rightarrow \mathsf{Prop}$ *such that the following properties hold:*

$$(n:nat)(r:TR)(inv\ n\ r) \to \ (Q\ n\ (output\ I[n]\ r)) \vee ((inv\ (S\ n)\ (update\ I[n]\ r)) \wedge (Rel\ (S\ n, update\ I[n]\ r)\ (n, r)))$$

$(inv\ O\ R)$

$(Acc\ Rel\ (O, R))$

then the following property holds $(Ex[n:nat](Q\ n\ CIRC[n]))$

PROOF: We apply the lemma 2 to:

the function implementing the circuit,
the invariant : $[n:nat][p:Str_{TI}*TR]((fst\ p)=(nthtl\ I\ n)) \wedge (inv\ n\ (snd\ p))$,
the property $[n:nat][p:Str_{TI}*TR](Q\ n\ (output\ (Hd\ (fst\ p))\ (snd\ p)))$,
and to the relation $[p,q:nat*Str_{TI}*TR](Rel\ (fst\ p, trd\ p)\ (fst\ q, trd\ q))$

4 The multiplier circuit

We study a very simple example introduced in [7]. This circuit implements a multiplier.

4.1 Description

We give a graphical representation of the circuit in figure 4.1.

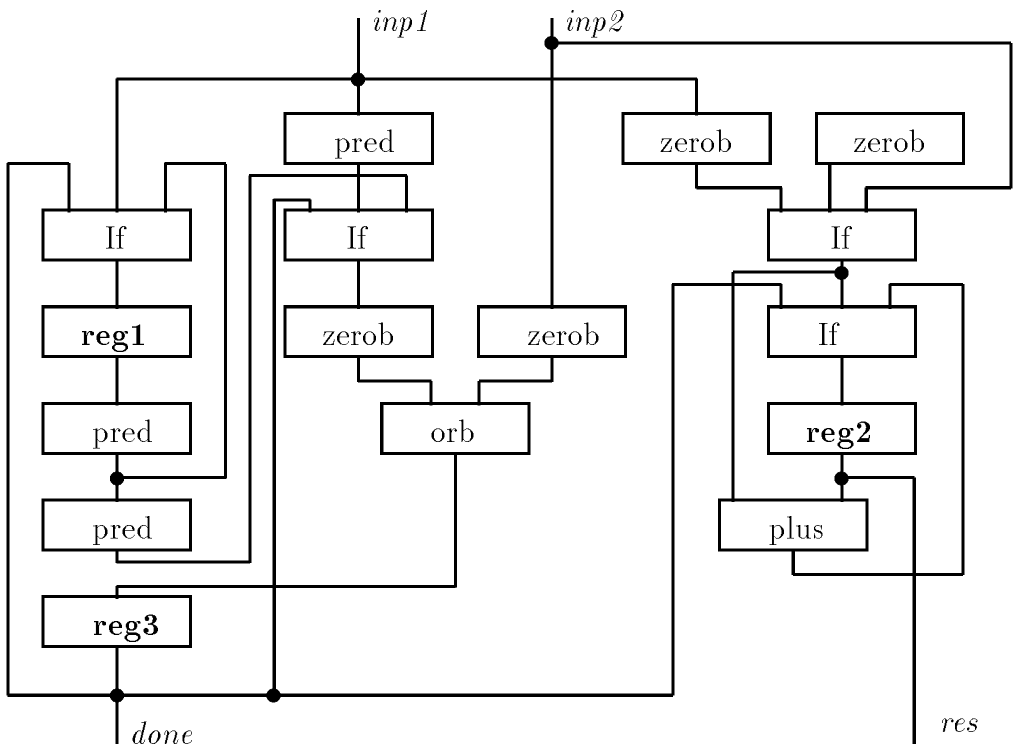

Figure 1: A multiplier circuit

4.2 Representation

Each combinational part of the circuit can be interpreted as a COQ function working on natural numbers and booleans. For the definition and the specification of the circuit, we use the

Coq modules `Arith` and `Bool` which defines the basic operations (*plus*, *mult*, *pred*) on natural numbers, (*orb*, *zerob*) on booleans and provide proofs of the basic properties of this operations.

Now we can introduce the functions for computing the outputs and updating the registers.

Each function depends a priori on the values of the inputs *inp*1 and *inp*2 of type *nat* and of the values of the registers *reg*1, *reg*2 of type *nat* and *reg*3 of type *bool*.

Section *definitions*.
Variables $i1, i2 : nat$.
Variables $r1, r2 : nat$.
Variables $r3 : bool$.
Definition *upd1* : *nat* := (*If* $r3$ $i1$ (*pred* $r1$)).
Definition *upd2* : *nat* := (*If* $r3$ (*If* (*zerob* $i1$) O $i2$) (*plus* (*If* (*zerob* $i1$) O $i2$) $r2$)).
Definition *upd3* : *bool* := (*orb* (*zerob* (*If* $r3$ (*pred* $i1$) (*pred* (*pred* $r1$)))) (*zerob* $i2$)).
Definition *res* : *nat* := $r2$.
Definition *done* : *bool* := $r3$.
End *definitions*.

The types for registers, entries and outputs are defined using the macro command Record which is equivalent to the definition of an inductive definition with only one constructor representing a product and which furthermore automatically build the projections whose names are specified.

Record *TR* : Set := *reg* {*reg*1 : *nat*; *reg*2 : *nat*; *reg*3 : *nat*}.
Record *TI* : Set := *inp* {*inp*1 : *nat*; *inp*2 : *nat*}.
Record *TO* : Set := *out* {*res* : *nat*; *done* : *bool*}.

The initial values for *reg*1 and *reg*2 can be arbitrary, we call them *ri1*,*ri2*. The initial value of *reg*3 needs to be *true*. The *update* and *output* function can easily be defined, as well as the initial value.

Definition *update* : $TI \rightarrow TR \rightarrow TR$:=
  $[i, r]$(*reg* (*upd*1 (*inp*1 i) (*reg*1 r) (*reg*3 r))
  (*upd*2 (*inp*1 i) (*inp*2 i) (*reg*2 r) (*reg*3 r))
  (*upd*3 (*inp*1 i) (*inp*2i) (*reg*1 r) (*reg*3 r))).
Definition *output* : $TI \rightarrow TR \rightarrow TO$:= $[i, r]$(*out* (*reg*2 r) (*reg*3 r)).
Definition *init* : *TR* := (*reg ri1 ri2 true*).
Definition *circ_mult* : $Str_{TI} \rightarrow Str_{TO}$:= (*circ output update init*).

4.3 Specification

The informal specification of the circuit is the following: assume the values of *inp*1 and *inp*2 are constants equal to X and Y then the next time *done* will be *true*, the value of *out* will be equal to $X \times Y$.

In order to express the specification, we introduce the property *stable* with type $nat \rightarrow$Prop which means that for all $k < n$, $I[k]=(inp\ X\ Y)$. We shall use the following properties of this predicate.

$$(stable\ O)$$
$$(n : nat)(stable\ (S\ n)) \rightarrow (stable\ n)$$
$$(n : nat)(stable\ (S\ n)) \rightarrow I[n]=(inp\ X\ Y).$$

The property to be proved for this circuit is :

Definition Q : $nat \rightarrow TO \rightarrow$Prop :=
  $[n, o](stable\ n) \rightarrow n{\neq}O \rightarrow (done\ o)=true \rightarrow (res\ o)=(mult\ X\ Y)$.

For the invariant, we use the construction *IfProp* with type $\mathsf{Prop} \to \mathsf{Prop} \to bool \to \mathsf{Prop}$ such that $(\mathit{IfProp}\ A\ B\ b)$ is equivalent to $(b{=}true \to A) \wedge (b{=}false \to B)$. The invariant is defined as:

$$
\begin{array}{l}
\textbf{Definition } \mathit{InvM} \ : \ nat \to TR \to \mathsf{Prop} \ := \\
\quad [n, r](\mathit{stable}\ n) \\
\quad \to (\mathit{IfProp}\ \ (n{\neq}O) \to (\mathit{reg2}\ r){=}(\mathit{mult}\ X\ Y) \\
\qquad\qquad\quad (\mathit{pred}\ (\mathit{reg1}\ r)){\neq}O \wedge X{\neq}O \wedge (\mathit{plus}\ (\mathit{mult}\ (\mathit{pred}\ (\mathit{reg1}\ r))\ Y)\ (\mathit{reg2}\ r)){=}(\mathit{mult}\ X\ Y) \\
\qquad\qquad\quad (\mathit{reg3}\ r)).
\end{array}
$$

Formally we have to check the two properties stated in proposition 4.2. The second condition which checks that the invariant is satisfied by the initial state of the circuit is trivially true by absurdity because at the initial stage *r3* is equal to *true* and $n{=}O$.

The second property requires a bit more working.

4.4 Proof of termination

It is not enough to prove that we get the expected result when *done* is equal to *true*, one need also to show that at some point *done* will be equal to *true*.

For this, it is enough to apply the lemma 5 with the property $[n : nat][o : TO]n{\neq}O \wedge (\mathit{done}\ o){=}true$. We have to find both a decreasing relation and an invariant. It is easy to remark that for the register r if $(\mathit{reg3}\ r){=}\mathit{false}$ then $(\mathit{reg1}\ r){\neq}O$ and consequently $(\mathit{reg1}\ r)$ decreases strictly. This is true except for the first step, consequently we can take the order:

$$
\begin{array}{l}
\textbf{Definition } \mathit{Rel} \ : \ nat{*}TR \to nat{*}TR \to \mathsf{Prop} \ := \\
\quad [p, q](\mathit{lt}\ (\mathit{fst}\ q)\ (\mathit{fst}\ p)) \wedge ((\mathit{lt}\ O\ (\mathit{fst}\ q)) \to (\mathit{lt}\ (\mathit{reg3}\ (\mathit{snd}\ p))\ (\mathit{reg3}\ (\mathit{snd}\ q))))
\end{array}
$$

This order can be proven to be well-founded (the first component increases a finite number of times then the second component decreases).

The invariant will be $[n : nat][r : TR](\mathit{reg3}\ r){=}\mathit{false} \to (\mathit{reg1}\ r){\neq}O$ which satisfies the expected properties.

4.5 Conclusion

In this paper we first showed the concrete representation of coinductive definitions (encoded impredicatively) as a sort of simple process.

Then we applied this representation to the type of streams. We showed principles using invariants for proving that a property holds for any element of the stream or for one of them. Finally we showed how to represent a sequential circuit as a function from a stream of inputs to a stream of outputs starting from functions describing how to update the registers and produce the outputs. Using this representations and the proof principles over streams, we completely derived the proof of a simple multiplier circuit.

The type of streams of objects of type A is isomorphic to the type of functions from *nat* to A. Consequently the development we made and principles we proved could equivalently have been done with functions like in [4].

The difference between the two types is intentional, a stream is a process which can iteratively produce values while a function is an arbitrary method to produce outputs from inputs. The notion of streams seems closer to the actual structure of a circuit, and we consequently believe that it should model it more accurately and suggests interesting proof methods. Besides the kind of proofs done in this paper, we can prove the equivalence of two circuits using a bisimulation or try to develop the circuit starting form its specification using parameterized streams like was experimented in [9].

Many experiments in hardware verification have been done with the NqThm or HOL theorem provers. In NqThm, circuits are represented as functions and proofs are done using induction and computation over functions, while in HOL they are represented as relations and proofs are done at the logical level. In Coq, we can freely choose one or the other representations as well as mixing them together or use other representation like the streams suggested in this paper. Few experiments have been performed on this topic, and further investigations remains to be done in order to see the advantages of Coq in this area.

Acknowledgments

We thank the team on hardware verification at Université de Provence in Marseille, especially S. Coupet and L. Pierre for fruitful discussions on the way circuits were represented for formal verification. These discussions suggested the study of this example.

References

[1] R. S. Boyer and J. S. Moore. *A computational logic*. ACM Monograph. Academic Press, 1979.

[2] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press, 1988.

[3] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual version 5.10. Rapport Technique 0177, INRIA-Rocquencourt-CNRS-ENS Lyon, July 1995. Available by anonymous ftp on ftp.inria.fr.

[4] S. Coupet-Grimal and L. Jakubiec. Verification formelle de circuits avec Coq. In *Journée du GDR-Programmation*, Lille, September 1994. Also available as a Coq contribution.

[5] E. Giménez. Co-inductive types in Coq. Technical report, Projet Coq, INRIA Rocquencourt, CNRS ENs Lyon, July 1995. To appear.

[6] E. Giménez. Implementation of co-inductive types in Coq: an experiment with the alternating bit protocol. Rapport de recherche, LIP-ENS Lyon, 1995. In preparation.

[7] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, 1986. also issued as University of Cambridge Computer Laboratory Technical Report No. 77, 1985.

[8] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant - a tutorial. Rapport Technique 0178, Projet Coq-INRIA Rocquencourt-ENS Lyon, July 1995. Available by anonymous ftp on ftp.inria.fr.

[9] F. Leclerc and C. Paulin-Mohring. Programming with streams in Coq. a case study : The sieve of eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, Types' 93*, volume 806 of *LNCS*. Springer-Verlag, 1994.

[10] G. C. Wraith. A note on categorical data types. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*. Springer-Verlag, 1989. LNCS 389.