Coq and Hardware Verification: A Case Study

Solange Coupet–Grimal and Line Jakubiec*

Laboratoire d'Informatique de Marseille - URA CNRS 1787
39, rue F. Joliot–Curie 13453 Marseille France
e-mail:{Solange.Coupet,Line.Jakubiec}@lim.univ-mrs.fr

Abstract. We present several approaches to verifying a class of circuits with the Coq proof-assistant, using the example of a left-to-right comparator. The large capacity of expression of the Calculus of Inductive Constructions allows us to give precise and general specifications. Using Coq's higher order logic, we state general results useful in establishing the correctness of the circuits. Finally, exploiting the constructive aspect of the logic, we can show how a certified circuit can be automatically synthesized from its specification.

1 Introduction

During the past decade, intensive research has developed in designing mechanized theorem provers, resulting in a great deal of new proof assistants. Hardware verification was one of the original motivations and main application of this area. Two of the earliest and most significant achievements were the work of Gordon using HOL [14, 6] and the work of Hunt [17] using Nqthm [5]. On the one hand, using general purpose theorem provers to state circuit correctness has several advantages over ad hoc tools. These include the precision of the specifications, enhancing the reliability of the verification process, and an increased generality leading to reusable methodologies and libraries. Now, on the other hand, meeting the requirements of the hardware verification community has been a stimulating challenge for logicians, mostly for those working in computer-aided proof-checking. Thus, despite the fact that existing theorem provers are high-level and general-purpose and cover fields of application much wider than hardware verification, verifying hardware remains a challenging domain of experiences. Among recent investigations, let us quote the verification with PVS [19] of a part of a pipelined microprocessor, the AAMP5 [25] and various uses of the prover LP to verify circuits [2, 24].
In this paper, a case study allows us presenting the capabilities of Coq in verifying and synthesizing hardware.
Coq is a proof tool developed at INRIA-Rocquencourt and ENS-Lyon [9]. It provides a very rich and expressive typed language and a higher order constructive logic. Moreover it offers the possibility of extracting automatically functional programs from the algorithmic content of the proofs.

* This work was supported by the GDR-Programmation; it was partially done during a six-month visit of Solange Coupet-Grimal at ENS-Lyon, in the Coq group.

A lot of significant developments have been performed with Coq. They can be found in the library of the users' contributions delivered with the Coq release. However, few investigation has been done to verify circuits. A multiplier first introduced by M.Gordon in [13] has been proven in [11], later extended by C. Paulin-Morhing in [23] to a more general proof of this circuit, using a codification of streams in type theory as infinite objects.[3] is a verification of a multiplier specified at the bit vector level.

The first part of the research presented here is most similar to the work done by K. Hanna, N. Daeche and M. Longley with Veritas$^+$ [16]. We follow their approach, as exemplified by a comparator studied in their paper, to specifying and proving a circuit, by making heavy use of dependent types and higher order logic. We have produced several reusable Coq modules, providing expressive and precise specifications as well as general theorems applicable to a whole class of circuits.

Several other researchers have been investigating the use of dependent types for reasoning about hardware. For example, interesting results using Nuprl have been produced [1, 18]. Like Coq, and unlike to Veritas$^+$, Nuprl relies on an intuitionistic logic. Until now, however, the intuitionistic aspect of the underlying logic has not been exploited (at least, we are not aware of any work in this direction). For us, filling this gap is worthwhile and is the aim of the second part of our study. Indeed, working with a constructive logic presents some difficulties, since it disallows the excluded-middle principle. To begin with, it may require an effort of the user who is used to classical reasoning. That is the reason invoked by Hanna for choosing classical logic for Veritas$^+$ [16]. But the computational aspect of the proofs is a valuable asset that can be used. In our opinion, this highly compensates the drawbacks, if any, of this kind of logic. In this paper, we present a methodology for synthesizing a circuit from its specification, using the Coq program extractor. As an alternative, we also give a methodology, using the tactic "program" [20], which can be seen as a mid-point between proving that a circuit is correct with respect to its specification (both being expressed in the Coq language) and "blindly" extracting the circuit as a ML function from a proof of a theorem which, roughly, states the existence of an object verifying the specification. This method consists in giving the prover both the functional description of the circuit and its specification. Thus, the proof process is guided by the knowledge of the term extracted from the proof.

The rest of this paper is organized as follows. Section 2 briefly introduces Coq. Section 3 deals with the description and the verification of the comparator. In Section 4, we present our two approaches to synthesizing the circuit. We conclude with an analysis of our results and of the performances of Coq.

2 An Overview of Coq

The Coq system is a tactic oriented proof-checker, in the style of LCF [15]. Developments can be split into various parameterized modules to be separately

verified. Thus, several developments can share modules that, being compiled once and for all, are loaded fast.
Coq's language implements a higher order typed lambda-calculus, the Calculus of Constructions [7, 8], enriched with inductive definitions [22].
Coq's logic is a higher order constructive logic and relies on the *propositions-as-types correspondence.* In Coq, a proposition is a type and a proof is a term inhabiting this type. Such a system provides an elegant unifying framework, since there is no fundamental difference between proofs and data, nor between propositions and datatypes. Therefore, proving amounts to type-checking.
However, there are two sorts of types : propositions are of sort *Prop* and sets are of sort *Set.* From a logical point of view, this distinction is not necessary, but it makes the system less confusing for the user. On the contrary, this distinction is highly significant when extracting programs from proofs, as we will show in the following.

Notations.
- $(A\ B)$ denotes the application of a functional object A to B.
- $[x : A]B$ denotes the abstraction of B with respect to a variable x of type A, (usually written $\lambda x \in A.B$).
- $(x : A)B$ as a term of type *Set*, denotes the cartesian product $\prod_{x \in A} B$. As a proposition, it corresponds to $\forall x \in A.B$. Moreover, if x does not occur in B, $A \to B$ is a shorter notation for the type of functions from A to B, or for a logical implication, depending on the sorts of A and B.

Induction and Recursion.
Selected parts of Coq specifications are depicted in Fig.1. The section *dependent_lists* is parameterized with respect to a term A of sort *Set.*
In this section is given a typical inductive definition involving dependent types, namely the definition of *list.* For each term n of type *nat, (list n)* is a type of sort *Set,* depending on the term n. *(list n)* denotes the type of the lists of elements of A whose length is n. This type is defined by means of two constructors, *nil* and *cons.* The type of *cons* expresses that it is a function which, given a natural number n, an element of A, and a length-n list, returns a length-$(n + 1)$ list. Moreover, Coq automatically generates the induction principle corresponding to the type *list.*
When the section is closed, the parameter A is *discharged* in the sense that all the terms depending on A are abstracted with respect to A. Outside the section, the type of polymorphic length-n lists will be $\lambda A : Set\ (list\ A\ n)$.
Numerals are defined in the section *numerals* which requires the module *dependent_lists.* The word *Local* introduces local definitions of the current section.
In the Coq syntax, given a set A and a predicate P on A, $\{x : A|(P\ x)\}$ denotes the subtype of A corresponding to the elements for which the property P holds. Terms of this type are pairs consisting of an element x of A and a proof of $(P\ x)$. The function *Inj,* taking such a pair as argument, erases its logical component and returns x. This function is parameterized with respect to A and P. For ex-

```
Section dependent_lists .
Variable A:Set.
Inductive  list :nat->Set:= nil:(list O)|
                            cons:(n:nat)A->(list n)->(list (S n)).
...
End dependent_lists.

Section numerals.
Require dependent_lists.

Definition BT:={b:nat|(lt O b)}.
Variable   BASE:BT.
Definition base:=(Inj nat [b:nat](lt O b) BASE).
Definition digit:={x:nat|(lt x base)}.
Definition val:digit->nat:=(Inj nat [x:nat](lt x base)).
Definition num:=(list digit).
Local Cons:=(cons digit).
Local Nil:=(nil digit).

Fixpoint Val[n:nat;X:(num n)]:nat:=<[m:nat]nat>Case X of
(*X=Nil*)          O
(*X=(Cons p d D)*) [p:nat][d:digit][D:(num p)]
                       (plus (mult (val d) (exp base p)) (Val p D)) end.
...
End numerals.
```

Fig. 1. "Dependent_lists" and "Numerals" Sections

ample, the variable $BASE$ is a pair of the form $(base, p)$ where $base$ is a natural number and p is a proof of $base > 0$. The function Inj, taking as arguments the set *nat*, the predicate $\lambda b.b > 0$, and the pair $BASE$, returns *base*.
We use subtypes to give precise specifications for systems of numeration such as base or digit definitions. For example, the type *digit* describes the set of natural numbers less than the base. The value *(val d)* of a digit d is the natural number obtained by keeping only the first component of its specification. A numeral is a list of digits, the length of which is specified.
On each concrete type inductively specified by constructors, it is possible to define functions recursively, by case analysis. The function *Val* is defined in such a way. Taking as arguments a natural number n and a length-n numeral X, it returns a natural number representing its value.
The expressions (*X=Nil*) and (*X=(Cons p d D)*) are just comments. The last line of the definition means that if X is the list whose length is $p+1$, whose head is d and whose tail is D then the function returns $(val\ d) * base^p + (Val\ p\ D)$ (note the recursive call in this last expression).

After this short presentation, we can move to the description and the verification of the comparator.

3 Verification

The particular example we choose is given in [16] as an illustration of an elegant and general methodology for specifying and proving iterative structures. As a first stage, it appeared to be an excellent benchmark in order to study the feasibility in Coq of already tested methods. But Coq's particular features lead us towards more powerful original approaches.

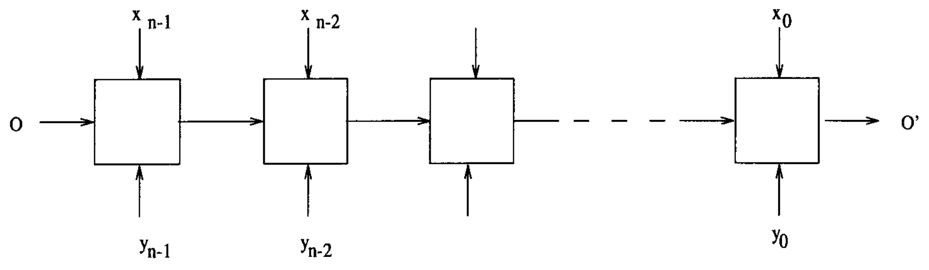

Fig. 2. A Comparator

The comparator (Fig.2) is a hardware device that accepts two numerals and determines their relative magnitude. It is composed of identical cells interconnected by a carry wire accepting comparison data in a 3-valued type. Each cell, from left to right, outputs a value that depends on the incoming carry and on the result of the comparison of two digit inputs.

3.1 Specifications

First of all, at the top level of genericity, there is the type *list* of dependent polymorphic lists presented in the previous section. It allows us to get high-level abstract specifications, more general than those in [16]. This type is particularly suitable in the framework of hardware specification where linear structures are prevalent. Numerals for example have been defined as particular lists. We have thus given in the *dependent_lists* section some additional definitions and properties that are not displayed on Fig.1 and that can be reused for any instance of *lists*. At this point, we do not go into more details about the contents of this module. A generic definition for connections of identical four ports cells is given in the *linear_structures* section (Fig. 3). It is parameterized with respect to the types A, B, C of the ports and to the relation *cell* implemented by the cells. Following the same idea as for the numerals, the type of a connection depends on a

```
Section linear_structures.

Require Dependent_lists.

Variables A,B,C:Set.
Variable cell:A->B->C->A->Prop.

Inductive connection :(n:nat)A->(list B n)->(list C n)->A->Prop:=
          C_0:(a:A)(connection 0 a (nil B) (nil C) a)|
          C_Sn:(n:nat)(a,a1,a':A)(b:B)(c:C)(lb:(list B n))(lc:(list C n))
                  (cell a b c a1)->
                  (connection n a1 lb lc a')->
          (connection (S n) a (cons B n b lb) (cons C n c lc) a').

End linear_structures.
```

Fig. 3. The "linear_structures" Section

natural number n representing the number of cells involved in the device. It also takes as arguments the input and the output carries of type A and two length-n lists of elements in A and B respectively. The term *connection* is inductively defined, in a typed Prolog style. With this analogy, the type of the constructors corresponds to the body of two Prolog rules (with reversed arrows) labeled C_O and C_Sn. The type of C_O states that, for all a in A, a connection with zero cells is just a wire carrying a. In this case, the two lists are the empty lists (*nil* B) and (*nil* C). The type of C_Sn states that any length-$(n+1)$ connection is obtained from a length-n connection whose port a_1 is connected to an additional cell. Figure 4 partially depicts a file in which various notions for comparing natural

```
Inductive order:Set:=L:order|E:order|G:order.

Definition comparison:=[v1,v2:nat]<order> Case (Lt_eq_Gt v1 v2) of
           [_:(lt v1 v2)]         L
           [_: v1=v2]             E
           [_:(gt v1 v2)]         G                    end.
```

Fig. 4. A Part of the "Compare_nat" Module

numbers are given. The set *order* $= \{L,\ E,\ G\}$ is denoted by the enumerated type *order*. The function *comparison* returns the value L, E or G depending on the relative magnitude of the natural numbers $v1$ and $v2$ it takes as arguments. This function is defined by case analysis on the term (*Lt_eq_Gt* $v1$ $v2$) which has been built before. This term is a proof of $(v1 < v2)$ *or* $(v1 = v2)$ *or* $(v1 > v2)$.

The second line of the definition of *comparison* must be interpreted by "given a proof of $(v1 < v2)$, return L". After that, several properties of *comparison* are established that are not shown on the figure.
All these tools having been defined, we are now able to describe the implementation and the expected behavior of the device. The section describing the com-

```
Section comparator.
(*system of numeration*)
Variable BASE: BT.
Local Digit:=(digit BASE).
Local ValB:=(Val BASE).
Local Num:=(num BASE).

(*semantics of the cells*)
Local f_cell:order->Digit->Digit->order:=
[o,x,y]<order>Case o of
(*o=L*)       L
(*o=E*)       (comparison (valB x) (valB y))
(*o=G*)       G                               end.

Definition cell:order->Digit->Digit->order->Prop:=
[o, x, y, o']  o'=(f_cell o x y).

(*structure of the comparator*)

Local Connection:=(connection order Digit Digit cell).
Local Comparator:=[n:nat][o:order][X,Y:(Num n)](Connection n E X Y o).

(*behavior of the comparator*)

Local Specif:(n:nat)(inf n)->(inf n)->order:=[n,X,Y]
(comparison (val_inf n  X) (val_inf n Y)).
```

Fig. 5. Implementation and Behavior of the Comparator

parator (Fig. 5) requires the section *numerals*. The first argument of the terms *digit*, *num* and *Val* is instantiated with the current base, given as a parameter *BASE*. The functional specification of a cell is given by the function *f_cell*, taking three arguments o, x, and y and defined by case analysis on the value of o. The local notion of connection is specified by the general term *connection* in which the types of the ports are $A = order$ and $B = C = Digit$. Let us point out that $(inf\ n)$ denotes the interval $[0,\ n[$ and that *val_inf* is the natural injection of type $(n : nat)(inf\ n) \to nat$. It is worth noting that the circuit has only two inputs (the numerals to be compared) since the carry input value is constrained to be E.

```
Local f_circ:(n:nat)order->(Num n)->(Num n)->order:=
[n,o,X,Y]<order>Case o of
  (*o=L*)          L
  (*o=E*)          (comparison (ValB n X) (ValB n Y))
  (*o=G*)          G                                  end.

Lemma general_correct:(n:nat)(X,Y:(Num n))(o,o':order)
       (Connection n o X Y o')->o'=(f_circ n o X Y).

Induction 1.            (*Induction on (Connection n o X Y o')*)

Clear  H o' o Y X n.    (*Erasing the useless hypothesis*)

Intros o;Case o;Simpl;Auto.  (*base case, case analysis on o*)
Apply sym_equal;Auto.
Clear  H o' o Y X n.
Intros n o o1 o' x y X Y H_cell H_n H_rec.
Inversion_clear H_cell.
Rewrite -> H_rec;Rewrite -> H.
Cut (eq ? o o);Auto.
Pattern 2 3 o ;Case o;Intros e;Rewrite -> e;Unfold f_cell ;
Unfold f_circ ;Auto.
(Cut (eq ? (comparison (valB x) (valB y))
           (comparison (valB x) (valB y)));Auto).
Pattern 2 3 (comparison (valB x) (valB y)) ;
Case (comparison (valB x) (valB y));Intros C;Apply sym_equal;
Unfold ValB ;Unfold Digit ;Auto.
Save.

Lemma correctness:(n:nat)(X,Y:(Num n))(o:order)
(Comparator n o X Y)->
o=(Specif  (exp base n) (Val_bound n X) (Val_bound n Y)).

(Unfold Comparator ;Unfold Specif ).
Intros n X Y o H.Rewrite -> (general_correct n X Y E o H).
Auto.                 (*automated resolution of the current goal*)
Save.
```

Fig. 6. Proofs of Correctness

3.2 Proving the correctness of the circuit

The theorem *correctness* in Fig.6 establishes that the implementation is correct with respect to the intended behavior and can be informally stated as follows :

For all n *in* nat, *for all* o *in* order , *for all length*-n *numerals* X *and* Y, *if* (comparator n o X Y) *then* $o = (Specif\ base^n\ \overline{X}\ \overline{Y})$ *where* $\overline{X}$ *is the value of the numeral* X *considered as a natural number of the interval* $[0,\ base^n[$.

However, because of the constant value of the input carry, the proof requires a generalization. Therefore a lemma *general_correct* is first established which sets forth the correct behavior of a connection, whatever value is given to the input carry. It is proven by induction on $(Connection\ n\ o\ X\ Y\ o')$. Fig.6 gives an idea of the length and the complexity of the proofs that have not to be read in detail.

4 Towards Synthesis

4.1 The Factorization Theorem

A more general approach oriented to verification as well to synthesis of 1-dimension arithmetic circuits is given in [16]. One can observe that, given a base b, each cell of the comparator implements a modulo-b version of the overall structure. This is also the case of right-to-left comparators, incrementors, circuits performing the multiplication of a numeral by a given natural number d, and so forth. Each cell of the latter, for example, performs the multiplication by d of a digit.
Let R be a relation of type: $(n : nat)A \rightarrow (inf\ n) \rightarrow (inf\ n) \rightarrow A \rightarrow Prop$.
We say that R is *proper* if

$$\forall n \in nat\ \forall a \in A\ (R\ 1\ a\ 0\ 0\ a).$$

We say that R is *factorizable* if the relation holds on two natural numbers x and x' as soon as it holds on the quotients and the remainders of the division of x and x' by any natural number n. More accurately, let n, m, x, x' be natural numbers such that:

$$x = nq + r\ ;\ x' = nq' + r'\ ;\ x, x' \in [0,\ mn[\ ;\ q, q' \in [0,\ m[\ ;\ r, r' \in [0,\ n[$$

R is factorizable if

$$\forall a, a1, a' \in A\ (R\ m\ a\ q\ q'\ a_1) \rightarrow (R\ n\ a_1\ r\ r'\ a') \rightarrow (R\ mn\ a\ x\ x'\ a').$$

The approaches presented in this section and in the following one apply to all linear structures whose cells implement such proper and factorizable arithmetic relation. The theorem of factorization states forth that for every relation R that is *proper* and *factorizable*, $(R\ b^n)$ is implemented by a connection of n cells implementing $(R\ b)$:

For all proper and factorizable relation R, *for all natural number* n, *for all length*-n *numerals* X *and* Y, *for all* a *and* a' *in* A, *if (Connection n a X Y a') then* $(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a')$.

The theorem is easily proven by induction on (*Connection* n a X Y a'). The proof of the comparator boils down to proving that the corresponding relation is proper and factorizable. This is done by case analysis on the variables a, $a1$ and a' occurring in the definition of *factorizable* and by using properties of the function *comparison*.
Indeed, this method is more general than that given in the previous section. However, although it is synthesis oriented, we have not really synthesized the circuit. We have established that a given linear structure satisfies a specification, but we have not obtained this structure from the specification. We show, in the following subsection, how to take advantage of the Coq proof extractor in an effective synthesis process.

4.2 Extracting the Circuit from its Specification

So far, we have used Coq as a powerful and expressive proof-checker. We intend now to take advantage of the constructive aspect of its logic.

Outline of the Coq Extraction Process Due to the *Curry-Howard isomorphism*, in Coq, proofs are λ-terms. They are thus objects of the underlying language, that can be displayed on the screen, stored, reused, exploited in various ways. Moreover, as λ-terms, proofs are nothing but functional programs.
In intuitionistic logic, a proof of a proposition of the form

$$\forall x \in A\ \exists y \in B\ (P\ x\ y)$$

necessarily contains an algorithm computing a function f of type $A \rightarrow B$ and a logical part certifying that for all x in A and y in B, if $y = (f\ x)$ then the proposition $(P\ x\ y)$ is verified. The Coq system involves a mechanism that is able to extract from such a proof a ML program computing the function f. By construction, this program is correct with respect to its specification P.
The distinction between the computational part and the logical part of a proof relies on the sorts, namely *Prop* and *Set*, in which the types are declared. A term is called *informative* if its type is of sort *Set* and *non-informative* if its type is of sort *Prop.* Analogously to program comments that are not taken into account by compilers, non-informative parts of the proof are erased during the extraction process. Moreover, all terms resulting from an extraction are terms in the system F_ω [12]. Informally, this means that the dependencies, if any, between types and terms, are lost during the extraction. For example, the term extracted from a "dependent" length-n list, as defined in this paper, is a usual list l, but with an additional parameter n of type *nat*. But l and n are not connected any more. Of course, the user gives the specifications and develops his proof in accordance to the term he wishes to obtain after extraction. As we mentioned in the section 2, the sorts *Prop* and *Set* are perfectly symmetrical and interchangeable. Let us outline, with an example, how the extraction works.
Let P be a predicate on the set of natural numbers. The proposition

$$\exists x \in nat\ (P\ x)$$

can be expressed in any of the three following terms T_1, T_2 or T_3:

- $T_1 = (ex\ nat\ P)$ of sort *Prop*, with $P : nat \to Prop$
- $T_2 = \{x : nat|\ (P\ x)\}$ of sort *Set*, with $P : nat \to Prop$
- $T_3 = \{x : nat\ \&\ (P\ x)\}$ of sort *Set*, with $P : nat \to Set$.

Now, what is the result of the extraction process on terms $t_1 : T_1$, $t_2 : T_2$ and $t_3 : T_3$?

- t_1 being non informative, it is erased by the extractor.
- As mentioned in section 2, t_2 is a pair consisting of a "witness" n of type *nat* and a non-informative proof of $(P\ n)$. The extracted term is n.
- For t_3, the result of the extraction is a pair $(n,\ b)$ where b comes from the proof of $(P\ n)$ which, this time, is informative. For example, if $(P\ x)$ is a disjunction, b is the boolean *true* if the left part of the disjunction has been proven and *false* in the other case.

This is a very short and informal presentation. For more details, one can refer to [21].

Synthesizing the Comparator The extraction principles presented in the previous paragraph leads us to a new version of the factorization theorem, that can be stated as follows:

For all proper and factorizable relation R, *for all natural number* n, *for all length-*n *numerals* X *and* Y, *for all* a *in* A, *there exists a' in* A *such that* $(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a')$.

It is organized so that a function f will be extracted from its proof. This function will take as arguments an element a of type A, a natural number n and two length-n numerals X and Y. It will return an element a' of type A. The function f is certified to be such that $(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a')$. At this point, it is necessary to give a functional specification of the relation R, that is to say to state which are the input ports and the output ports of a circuit implementing R. This is done by defining the relation R as follows:

$$(R\ n\ a\ x\ y\ a') \text{ if and only if } a' = (FR\ \ n\ a\ x\ y)$$

where FR is of type:

$$FR : (n : nat)A \to (inf\ n) \to (inf\ n) \to A.$$

The function f will be defined by an algorithm which depends on the way the proof is developed. Here, we make an induction on n.

- If $n = 0$, we give the witness $a' = a$ and prove that $(R\ 1\ a\ 0\ 0\ a)$ using the fact that R is proper.

- Let us now consider a an element of type A, n a natural number and two length-$(n+1)$ numerals $X = (Cons\ d\ D)$ and $Y = (Cons\ d'\ D')$. Let a_1 be $(FR\ b\ a\ d\ d')$. By induction hypothesis, there exists a' such that $(R\ b^n\ a_1\ \overline{D}\ \overline{D'}\ a')$. The relation being factorizable we can deduce that

$$(R\ b^{(n+1)}\ a\ \overline{(Cons\ d\ D)}\ \overline{(Cons\ d'\ D')}\ a').$$

The type *list* extracted from the type $(n : nat)(list\ A\ n)$ is nearly the usual inductively defined type for polymorphic lists (the constructors take an additional argument of type *nat*). From FR a function fr of type $nat \to A \to nat \to nat \to A$ is obtained. Note that there are no dependencies between types and terms any longer and that the logical content of $(inf\ n)$ has disappeared. The extraction process, on the proof of the theorem, results in the function f of type $nat \to A \to list \to list \to A$ defined by

- $(f\ 0\ a\ D\ D') = a$
- $(f\ (n+1)\ a\ (Cons\ d\ D)\ (Cons\ d'\ D')) = (f\ n\ a_1\ D\ D')$
  where $a_1 = (fr\ \ b\ a\ d\ d')$

From the extracted term, a ML program can be automatically generated. It produces the expected result, when taking as inputs a natural number n and two length-n numerals X and Y. If one of the numerals is shorter than n, an exception is returned. Numerals longer than n are truncated.
Synthesizing the comparator is now extremely simple. It is sufficient to apply this theorem with the particular relation *cell* implemented by the cells of the comparator and defined in Fig.5.

4.3 A Mixed Approach using the Tactic "Program"

In the previous section, we showed how the user develops his proof according to the program he has in mind and he wants to synthesize. The *Program* tactic just implements the idea that the program to be extracted contains information about the structure of the proof and thus that it can be used as a guide during the proof process. This methodology can be viewed as dual to the extraction.
Let us consider the function *Impl* defined by :

$$(Impl\ 0\ a\ X\ Y) = a$$

and

$$(Impl\ (n+1)\ a\ X\ Y) = (Impl\ n\ (FR\ b\ a\ (Hd\ X)\ (Hd\ Y))\ (Tl\ X)(Tl\ Y))$$

In these equations Hd and Tl denote respectively the functions head and tail. This program is associated with the theorem

For all proper and factorizable relation R, *for all natural number* n, *for all length-*n *numerals* X *and* Y, *for all* a *in* A, *there exists a' in* A *such that* $(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a')$

to be proven by the command *Realizer*. Then the tactic *Program_all* generates sequences of introduction, application and elimination tactics on every subgoal depending on the syntax of the program *Impl*. In particular, the induction scheme is found out by the system. Although the proof process is not fully automated, it is highly simplified.

5 Summary and Conclusions

Our aim in this paper was to demonstrate the capabilities of Coq in the field of hardware verification. We have given a general and illustrated presentation of the prover and we have investigated how to reap the greatest benefit of its particular features not only for proving circuit correctness but also for effectively synthesizing devices at the algorithmic level. Our results apply to arithmetic linear structures the cells of which implement proper and factorizable relations (incrementor, comparators, multipliers, $\cdots$). To sum up, precise and general specifications have been expressed in a natural way. Several reusable modules have been developed (for handling lists, numerals, repetitive arithmetic structures) in which generic properties and theorems have been proven. Several approaches have been investigated (verification of a particular circuit, verification of a class of circuits, synthesis of a class of circuits, intermediate approach). The synthesis methodology relies on the constructive aspect of the logic and, in practice, on the Coq extractor. A functional description of an implementation is automatically extracted from a proof of a statement of the form

$$\forall y \in A\ \exists x \in B\ (P\ x\ y).$$

P is a relation between the input x and the output y and represents the expected behavior (specification) of the circuit. In a third intermediate approach, specification and implementation are both given to the prover. As the proof process is guided by the syntactical structure of the implementation, it is more automated and thus easier to use.

Relying on the Curry-Howard isomorphism, Coq provides an elegant unifying framework for specifying and proving. Proof-checking and type-checking are the same process (in PVS for example a type checking step must precede the classical proof process). Let us also mention that, unlike in Nuprl, type-checking in Coq is decidable. Undoubtedly, Coq is a powerful tool, with advanced features, the most futuristic of them being the synthesis of certified programs. The drawback, in our point of view, is the lack of user friendliness and automation. Exploiting all Coq subtleties still requires skill and expertise.

However, various works are now in progress that will make Coq much easier to use in the future. A nice interface, CtCoq, is already available [4]. Moreover, a tool is being developed that, from the script of a Coq proof, automatically generates a text in natural mathematical language [10]. It will be of interest for analyzing, simplifying, and debugging proofs. New approaches are also being studied for improving the extraction process and the modularity. Finally, arithmetic decision procedures are about to be integrated in the system.

6 Acknowledgments

We would like to thank all the members of the Coq group at ENS- Lyon and INRIA-Rocquencourt for their stimulating seminar. We are particularly grateful to Cristina Cornes, Catherine Parent and Christine Paulin-Mohring for helpful discussions. Our thanks go also to the reviewers for their constructive comments.

References

1. M. Aagaard and M. Leeser. A Methodology for Reusable Hardware Proofs. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, 1992.
2. M. Allemand. *Modélisation Formelle et Preuve de Circuits avec LP*. PhD thesis, Université de Provence, July 1995.
3. L. Arditi. Formal Verification of Microprocessors : a First Experiment with the Coq Proof Assistant. Research Report I3S/Université de Nice - Sophia Antipolis. RR-96-31, 1996.
4. J. Bertot and Y. Bertot. Ctcoq : a System Presentation. In *CADE-13*, 1996.
5. R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press Inc., 1988.
6. A. Camilleri, M. Gordon, and T. Melham. Hardware Verification Using Higher Order Logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.
7. T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, Janvier 1989.
8. T. Coquand and G. Huet. Constructions : A Higher Order Proof System For Mechanizing Mathematics. In *EUROCAL'85*, number 203 in LNCS. Springer-Verlag, 1985.
9. C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and W. Benjamin. The Coq Proof Assistant Reference Manual. Technical report, INRIA-Rocquencourt, CNRS-ENS Lyon, Feb. 1996.
10. Y. Coscoy, G. Kahn, and L. Théry. Extracting Text from Proof. In *Typed Lambda-Calculi and Applications*, number 905 in LNCS. Springer-Verlag, April 1995.
11. S. Coupet-Grimal and L. Jakubiec. Vérification Formelle de Circuits avec COQ. In *Journées du GDR Programmation*, Sept. 1994.
12. J.-Y. Girard. The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science 45*, 1986.
13. M. Gordon. LCF-LSM. Technical Report 41, University of Cambridge, 1984.
14. M. Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. Technical Report 77, University of Cambridge Computer Laboratory, 1986. edited by G.Milne and P. A. Subrahmanyam, North Holland.
15. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF : A Mechanized Logic of Computation*, volume 78 of *LNCS*. Sringer-Verlag, Department of Computer Science, University of Edinburgh, 1979.
16. F. Hanna, N. Daeche, and M. Longley. Specification and Verification Using Dependent Types. *IEEE Transactions on Software Engineering*, 16(9):949–964, Sept. 1990.

17. W. A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasonning*, 5(4):429–460, 1989.
18. M. Leeser. Using Nuprl for the Verification and Synthesis of Hardware. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, International Series on Computer Science. Prentice Hall, 1992.
19. S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial on Using PVS for Hardware Verification. In *2nd International Conference on Theorem Provers in Circuit Design*, number 901 in LNCS, pages 258–279. Springer Verlag, Sept. 1994.
20. C. Parent. *Synthèse de Preuves de Programmes dans le Calcul des Constructions Inductives*. PhD thesis, Ecole Normale Supérieure de Lyon, Janvier 1995.
21. C. Paulin. *Extraction de Programmes dans Coq*. PhD thesis, Université Paris 7, Janvier 1989.
22. C. Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. Research Report 92-49, Ecole Normale Supérieure de Lyon, 1992.
23. C. Paulin-Mohring. Circuits as Streams in Coq. Verification of a Sequential Multiplier. *Basic Research Action "Types"*, Juillet 1995.
24. J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using Transformations and Verification in Circuit Design. *Formal Methods in System Design*, (3):181–209, Dec. 1993.
25. M. K. Srivas and S. P. Miller. Applying Formal Verification to a Commercial Microprocessor. *IFIP International Conference on Computer Hardware Description Languages*, Aug. 1995.