



# *Laboratoire de l'Informatique du Parallélisme*

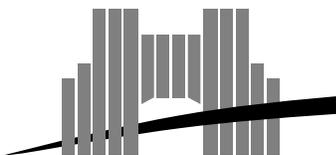
Ecole Normale Supérieure de Lyon  
Unité de recherche associée au CNRS n°1398

## *Developing certified programs in the system Coq The Program tactic*

C. Parent

October 1993

Research Report N° 93-29



### **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;

Adresses électroniques :

lip@frensl61.bitnet;

lip@lip.ens-lyon.fr (uucp).

# Developing certified programs in the system Coq

## The Program tactic

C. Parent

October 1993

### Abstract

The system *Coq* is an environment for proof development based on the Calculus of Constructions extended by inductive definitions. Functional programs can be extracted from constructive proofs written in *Coq*. The extracted program and its corresponding proof are strongly related. The idea in this paper is to use this link to have another approach: to give a program and to generate automatically the proof from which it could be extracted. Moreover, we introduce a notion of annotated programs.

**Keywords:** Calculus of Constructions,  $\lambda$ -calculus, extraction

### Résumé

Le système *Coq* est un environnement de développement de preuves basé sur le Calcul des Constructions enrichi par des définitions inductives. Des programmes fonctionnels peuvent être extraits des preuves constructives écrites en *Coq*. Le programme extrait et sa preuve sont fortement reliés. L'idée dans ce papier est d'utiliser ce lien pour une approche différente : donner un programme et générer automatiquement la preuve dont il aurait pu être extrait. De plus, une notion de programmes annotés est introduite.

**Mots-clés:** Calcul des Constructions,  $\lambda$ -calcul, extraction

# Developing certified programs in the system Coq

## The Program tactic

C. Parent \*

LIP, URA CNRS 1398, ENS Lyon  
46 Allée d'Italie, 69364 Lyon cedex 07, France  
e-mail : parent@lip.ens-lyon.fr

October 1993

### Abstract

The system *Coq* is an environment for proof development based on the Calculus of Constructions extended by inductive definitions. Functional programs can be extracted from constructive proofs written in *Coq*. The extracted program and its corresponding proof are strongly related. The idea in this paper is to use this link to have another approach: to give a program and to generate automatically the proof from which it could be extracted. Moreover, we introduce a notion of annotated programs.

## 1 Introduction

The system *Coq* is a proof development environment based on the Calculus of Constructions with inductive definitions [PM93, DFH<sup>+</sup>93]. It uses the Curry-Howard isomorphism [How80], more precisely the fact that one can identify the notion of proofs and programs and the notion of types and specifications. It follows Heyting's semantics of constructive proofs : a proof of  $\forall x.P(x) \Rightarrow \exists y.Q(x, y)$  gives a method to transform an object  $i$  and a proof of  $P(i)$  into an object  $o$  and a proof of  $Q(i, o)$ . Systems following this interpretation like *Coq* can be seen as programming languages. Indeed, in a system like *Coq*, a proof of a specification is developed and can be represented as a program corresponding to the method in the Heyting's sense. In fact, a proof contains a lot of redundant informations : there are informations about the way of calculating the result (i.e. the interesting part) and informations about the way of calculating the correctness proof (i.e. the uninteresting part). So, following this idea that informations need to be removed, programs can be extracted from proofs in *Coq*, using a notion of realizability [PM89a]. Realizability is an interpretation of the computational contents of intuitionistic proofs as programs satisfying a given specification. Such a program is called a *realization* of the specification. Realizability allows to eliminate non computational parts of proofs (to extract programs from proofs) and to certify extracted programs to be still correct with respect to the initial specification. Indeed, from proofs written in *Coq*, programs can be extracted [PM89b] into a typed functional language like ML. Some other systems like PX and NuPrl offer similar possibilities of extraction [HN88, Con86]. Both of them are using untyped theories. More precisely, PX uses an untyped theory and, in NuPrl, the

---

\*This research was partly supported by ESPRIT Basic Research Action "Types for Proofs and Programs" and by Programme de Recherche Coordonnées and CNRS Groupement de Recherche "Programmation".

theory is typed but the extracted terms are untyped. PX uses a notion of proofs-as-programs which is not the Curry-Howard one but there is two different levels (0 for proofs and 1 for programs). Then, a process of extraction is defined using a special notion of realizability called the px-realizability. A difference between the realizability used for the *Coq* extraction and the px-realizability is that the px-realizers are allowed not to terminate. In NuPrl, there is no distinction between proofs and programs. But, a process of extraction can be expressed : redundant informations can be hidden using the fact that if  $a$  is of type  $\{x : A \mid P\ x\}$  then  $a$  is as well of type  $A$ , but a consequence of this is that typing becomes undecidable.

A problem when extracting programs from proofs is that proofs are first developed and, then, programs are extracted. This is not the case of other methods [BM92, Pol92] where proofs and programs are developed hand-by-hand. The aim here is based on the idea that a proof is developed differently if one waits for a program or another (for instance, different proofs of a same specification lead to different sort algorithms). A program can then be considered as a skeleton of its proof containing exactly all its computational contents. The aim in this paper is to develop a program and then to try to generate automatically using the program the proof of its specification. In fact, a program can be supposed to be a realization of its specification and, using this information, its proof can be generated almost automatically. But, two types of propositions have to be distinguished : *specifications* which have computational contents and are typically existential formulas such as  $\forall x.P(x) \Rightarrow \exists y.Q(x, y)$  describing the properties to prove; *logical assertions* which have no computational contents (for instance, the pre and postconditions  $P$  and  $Q$ ). Specifications can be automatically proved using the program but logical assertions can be arbitrarily complex and one cannot hope to solve them mechanically. The final aim is so to solve specifications and leave logical assertions to the user representing logical properties the program has to verify. Then, it can be certified that the program is correct if these properties are verified.

The method to develop automatically the proof from the program will use the structure of the program (which are variable, constant, abstraction, application or recursion). Each structure will give a certain method of proof.

The paper is organized as follows. In a first part, an example is developed in the system *Coq* in order to illustrate the program development method and introduce what we would like to obtain. In a second part, we give the methodology for automating the proof development. Then, we discuss some optimizations and conclude.

## 2 An example of development in *Coq*

Let us consider the division algorithm as an example of a development in *Coq*. A division program would take two arguments  $a$  and  $b$  and give as outputs  $q$  and  $r$  such that they verify  $a = b * q + r \wedge b > r$ . But, a necessary condition is that  $b > 0$ , otherwise the condition on  $r$  cannot be satisfied. So, a specification of a division algorithm should be :

$$\forall b.b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \wedge b > r) \tag{1}$$

Every constructive proof of such a specification gives an algorithm by extraction (see [PMW92]). If one gives a program, one gets the existence of such an algorithm and one has a skeleton of a possible proof. This skeleton allows to do the computational parts of the proof (i.e. to solve specifications). If the left logical assertions can be solved (e.q. prove that loop invariants are preserved), then the initial program can be certified correct with respect to the initial specification.

A program (in ML) for our example could be :

```

let div b a = divrec a where rec divrec = function
  0 -> (0,0)
  | Sn -> let (q,r) = divrec n in
           if (Sr<b) then (q,Sr) else (Sq,0) ;;

```

Let us do a mathematical proof of our specification and see the link with the ML program.

One wants to prove  $\forall b.b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \wedge b > r)$ . Given  $b, b > 0$ , one wants to prove  $\forall a.\exists q.\exists r.(a = b * q + r \wedge b > r)$ . Let us do an induction on  $a$ . First case :  $a = 0$ . Then, one needs to prove  $\exists q.\exists r.(0 = b * q + r \wedge b > r)$ . The values  $q = 0$  and  $r = 0$  are good candidates since  $0 = b * 0 + 0 \wedge b > 0$ . Second case : one assumes  $\exists q.\exists r.(n = b * q + r \wedge b > r)$  for a given  $n$ , and one wants to prove  $\exists q.\exists r.(Sn = b * q + r \wedge b > r)$ . Given  $q$  and  $r$  from the induction hypothesis, let us look for  $q'$  and  $r'$  such that  $(Sn = b * q' + r' \wedge b > r')$ . Two subcases : if  $Sr < b$  then let us take  $q' = q$  and  $r' = Sr$ . Then, one has to prove  $(Sn = b * q + Sr \wedge b > r)$ . But,  $b > Sr$  by hypothesis and one knows by the induction hypothesis that  $n = b * q + r$ . So, this case is solved. If  $Sr \geq b$  then let us take  $q' = Sq$  and  $r' = 0$ . Then, one has to prove  $(Sn = b * q + b \wedge b > 0)$ . The second part of the conjunction is trivial. For the first one, one knows  $b > r$  and  $Sr \geq b$ , so one can conclude  $b = Sr$ . And the second case is solved.

Remark that the structure of the proof is closely related to the structure of the program : induction on  $a$ , recursive call on  $n$  in the second case of the induction ....

Let us now see how this proof can be developed in *Coq* and how a program can be extracted. Then, the link between proofs and programs will appear once more. First, *Coq* allows the interactive development of proofs. One gives a specification as above (1) and then one can use predefined tactics to develop a proof. The reasoning follows a natural deduction style. There are introduction (**Intro**) and elimination (**Elim**) tactics and resolution tactics (**Apply** or **Exists**). All the steps of the mathematical proof can be expressed with these tactics : the introductions by **Intro**, the inductions by **Elim**, the introductions of the existential quantifiers by **Exists** ... The proof of our example can then be expressed only using **Intro**, **Elim** and **Exists**. Comments are expressed between (\* and \*).

```

Intros b H a.          (* H : b>0 *)
Elim a.
Exists 0. Exists 0.    (* b>0 and 0=b*0+0 *)
Auto.
Intros n H0.          (* H0 : induction hypothesis *)
Elim H0.              (* getting back q *)
Intros q H1.
Elim H1.              (* getting back r *)
Intros r H2.
Elim (inf b (S r)).    (* deciding of the order of b and Sr *)
Intros Le.            (* Le : b<=Sr *)
Exists (S q). Exists 0. (* subgoal easy to resolve : b>0 and Sn=b*(Sq)+0 *)
Intros Gt.            (* Gt : b>Sr *)
Exists q. Exists (S r). (* subgoal easy to resolve : b>Sr and Sn=b*q+Sr *)

```

This proof is close to the mathematical one in the sense that one can retrieve the steps of introductions, inductions .... Moreover, if one extracts the computational part from this *Coq* proof, one gets the program above. Our aim is now to take this program and to retrieve the computational parts of its corresponding proof.

Let us explain first how proofs and programs are represented in *Coq*. Proofs are typed  $\lambda$ -terms marked with informations on their computational contents (i.e. if they are informative or logical). The process of extraction consists in forgetting from the proof term all the logical parts to obtain a program term, so a typed  $\lambda$ -term with only informative parts. The extraction function is a forgetful function. Our aim is to inverse this function to obtain a proof term from a program term. The extraction function is defined on the structure of the proof terms. Thus, we have to define a function (that we will call the automation function) on the structure of the program terms. Now we describe the strategy of this automation function.

### 3 Automation method

As we said before, the principle is to give a specification and a program and to prove it is correct with respect to the specification. The method consists in associating the program to the current specification.

Then, an automatic proof (step-by-step) consists in applying the good tactic, giving as a result a new specification (or more) which is associated to a good new program (or more), which has to be a subprogram(s) of the previous program. Our method deals with *partial programs*, associated to *partial specifications* and builds *partial proofs*.

We first need to check that the type of the program is indeed convertible to the extraction of its specification (this property is kept as an invariant by our method). Indeed, one wants the program  $p$  to be the extraction  $\mathcal{E}(P)$  of a proof  $P$  of the specification  $S$ ; but, from the condition  $P : S$ , one gets  $p : \mathcal{E}(S)$ . One has to keep in mind this important information that the type of the program has to be the extraction of the specification.

Now, we explain how it will be done by cases on the structure of programs :  $\lambda$ -abstraction, application, recursion, variable and constant. We describe some heuristics and explain more precisely why annotated programs are sometimes necessary.

#### 3.1 Programs

Programs are typed  $\lambda$ -terms given in a  $F_\omega^{Ind}$  form (we consider that they are in normal form without apparent redexes). Their structure can be a  $\lambda$ -abstraction, an application, a recursion, a variable or a constant. Note that constants have a particular state. They can be considered like variables. But, in fact, they are programs already proved and they hide the structure of this program. In some cases, this information can be needed. If, they are just considered like variables, the information they hide is lost. So, when it is useful, constants are expanded to retrieve the structure of the program they correspond to and to use this important information.

In order to explain the method, we need to define what we call *coarse programs*.

**Definition 1** *A program is coarse with respect to a specification  $S$  if it is exactly a proof of the specification  $S$ .*

Such programs are said to coarsely resolve the specification. They are interesting because they represent exactly the proof of their specification. They contain all the information useful for the proof. With such programs, a complex research of the corresponding proof can be avoided since it is directly in the program.

### 3.1.1 $\lambda$ -abstractions

Let us first consider a typical example what kind of situations could appear. Consider the previous division algorithm written in the *Coq* syntax<sup>1</sup>

```
[b:nat][a:nat]
<nat*nat>Match a with
  (* 0 *) <nat,nat>(0,0)
  (* S *) [n:nat][H:nat*nat]
    <nat*nat>let (q,r:nat) = H in
      <nat*nat> if (inf b (S r)) then
        <nat,nat>((S q),0)
      else <nat,nat>(q, (S r)).
```

whose specification is  $\forall b. b > 0 \Rightarrow \forall a. \exists q. \exists r. (a = b * q + r \wedge b > r)$ . The program is a  $\lambda$ -abstraction. This indicates to introduce the  $b$ . This case is very simple. One introduces the  $b$  and then generates a new specification  $b > 0 \Rightarrow \forall a. \exists q. \exists r. (a = b * q + r \wedge b > r)$  for the program `[a:nat]`... Now, to mimic the program, one would like to introduce the symbol  $a$ . But this introduction cannot be performed as the specification has not the shape  $\forall a. \exists q. \exists r. (a = b * q + r \wedge b > r)$  but the shape  $b > 0 \Rightarrow \forall a. \exists q. \exists r. (a = b * q + r \wedge b > r)$ . In such a situation, one has to introduce the non-computational hypothesis ( $b > 0$ ) before the computational variable  $a$ . More generally, before introducing a computational variable (corresponding to a variable of the program), one has to introduce all the non-computational hypotheses that have no equivalent in the program.

More formally, as one knows the correspondence between the program  $p$  and the specification  $S$ , one knows that if the program is a  $\lambda$ -abstraction ( $p \equiv [x : \mathcal{E}(I)]p'$ ) then the specification is a product ( $S \equiv (\vec{y} : \vec{L})(x : I)S'$  with  $\vec{L}$  representing a vector of logical terms and  $I$  an informative term), since the type of the program is  $(x : \mathcal{E}(I))A$  and is convertible to the extracted type of  $S$ . One has to do introductions. As we saw on the example, the problem is that one cannot be sure to have to do only one introduction. Indeed, a  $\lambda$ -abstraction in a program ( $[x : \mathcal{E}(I)]p'$ ) corresponds to a product in the specification ( $(x : I)S'$ ), but an informative one. The specification can contain non-informative products ( $(\vec{y} : \vec{L})\dots$ ). So, the method consists in this case in doing as many introductions as they are non-informative products in the specification and then one last introduction. This last introduction is about the proof term which corresponds to the  $\lambda$ -abstraction in the program.

This way, all the logical introductions of the proof that cannot appear in the program and then the “real” informative introduction the program indicates us are done. The new program associated to the new specification is the previous program without the corresponding  $\lambda$ -abstraction.

### 3.1.2 Applications

This case is more complex than the previous one. Let us take an example on parametric lists. Suppose one wants to prove  $\forall l. \forall n. \exists m. (n + (\text{length } l) = m)$ . A trivial proof is to explicitly give  $(n + (\text{length } l))$  as a witness that  $\forall l. \forall n. \exists m. (n + (\text{length } l) = m)$ . But, one wants to give a realizer to keep constructivity. So, a corresponding algorithm written in *Coq* could be :

```
[l:list]<nat->nat>Match l with
  [n:nat]n
  [a:A][m:list][H:nat->nat][n:nat](H (S n))
```

---

<sup>1</sup>The notation  $[x:A]B$  is for the  $\lambda$ -term  $\lambda x : A. B$ , and `<P>Match x with` is the case analysis

Let us look at the last part of this program ( $\mathbb{H} \text{ (S n)}$ ) where  $\mathbb{H}$  is the induction hypothesis. The corresponding goal to prove is  $\exists m.(n + (\text{length} (\text{cons } a \ l)) = m)$ . The specification of  $\mathbb{H}$  is  $\forall n.\exists m.(n + (\text{length } l) = m)$  since  $\mathbb{H}$  is the induction hypothesis. So, the specification of ( $\mathbb{H} \text{ (S n)}$ ) is  $\exists m.(Sn + (\text{length } l) = m)$ . This resolves the goal since  $(\text{length} (\text{cons } a \ l)) = S(\text{length } l)$ . Let us call this lemma `Length_1`.

Let us now show how programs already proved can be used in other programs. Suppose one wants to prove  $\forall l.\exists m.((\text{length } l) = m)$ . One can use the previous program and the associated program can be `[1:list](Length_1 1 0)`. Then, the specification of (`Length_1 1 0`) is  $\exists m.(0 + (\text{length } l) = m)$ . And this trivially resolves the goal.

Let us describe the method that is used. Let us write the program  $(c \ a_1 \ \dots \ a_n)$  where  $c$  is not an application. Then,  $c$  is either a variable, a constructor or a recursion. Consider first when  $c$  is a variable or a constructor. The head symbol of the corresponding proof term is the same variable<sup>2</sup>. The proof has the shape  $(c \ b_1 \ \dots \ b_p)$ . Let  $(x_1 : B_1) \ \dots \ (x_p : B_p)B$  be the type of  $c$ . One wants to generate the proof terms  $b_1 \ \dots \ b_p$  of type  $B_1 \ \dots \ B_p$ .  $B_i$  with non-computational contents are left to the user, the others are associated to their extracted terms  $a_1 \ \dots \ a_n$ . Then, one applies to them the same method recursively.

Coarse programs can here be used as an optimization. If the specification is exactly the type of the program (coarse programs), then non-computational terms will never appear in the previous method. The proof is exactly the extracted term. So, one can use this extracted term for the proof term.

When  $c$  is a recursion. Then, there is no unique solution for the corresponding proof term. So, one chooses the following heuristic : if  $c$  is a recursion  $Rec_I(m, P, lf)$ <sup>3</sup> then its corresponding proof term is a recursion. Retrieving the proof terms corresponding to  $I$  and  $P$  is not easy because there are many solutions. But, in fact, finding the value of  $I$  is not very difficult since one can use the type of  $m$ .  $P$  is definitely a problem. Let us take the terms  $(Rec_I(m, \lambda x : L.P, lf) \ x)$  and  $Rec_I(m, P, lf)$ . They are both in an  $\eta$ -long form and equivalent in terms of programs if  $x$  is a logical argument. So it is impossible to decide from programs which predicate one needs to take at the level of proof. To avoid failure, one has an heuristic corresponding to the following inference rules :

$$\frac{Rec_I(m, P, lf) : (x : A) B \quad a : A}{(Rec_I(m, P, lf) \ a) : B[x/a]}$$

$$\frac{Rec_I(m, P, lf) : A \rightarrow B \quad a : A}{(Rec_I(m, P, lf) \ a) : B}$$

One applies a generalization. The generalization of *name*, if *name* is a term on which depends the specification, replace it by the same specification quantified by the variable *name*. So, we use the previous inference rules to obtain two subproblems : one for the head of the application corresponding to a generalized specification, another for the argument (obviously, more if they are more than one arguments), corresponding to the specification of the argument (trivially solved if the argument is a coarse program).

---

<sup>2</sup>If  $c$  is a constructor  $Constr(i, ind)$  then its corresponding proof term is a constructor  $Constr(i, Ind)$  and, with respect to the *Coq* representation of inductive types, the current goal is an inductive type :  $(y_1 : B_1) \ \dots \ (y_l : B_l)(Ind \ t_1 \ \dots \ t_k)$ . Then, *Ind* is known from the current goal.

<sup>3</sup> $Rec_I(m, P, lf)$  is a notation for the *Coq* term : `<P>Match m with lf`

So, one comes back to a case of recursion (see 3.1.3). Note it is not trivial to retrieve the specification of a program. This motivates the introduction of annotations in programs (see 4).

In the foregoing, one explained a method to resolve application cases whatever the head symbol of the application is. But, the problem of possible non-computational introductions (like in 3.1.1) has not been considered. The head of the specification can contain non-computational products. Logical introductions have to be done since they have no correspondent in the program. Let us take the example of the division algorithm at the step  $((\text{Sq}), 0)$ . The specification is  $(b \leq (Sr)) \rightarrow \exists q. \exists r. (Sn = b * q + r) \wedge (b > r)$ . It is clear that one wants first to introduce  $(b \leq (Sr))$  and then to resolve the goal with  $((\text{Sq}), 0)$ . Thus, do all the logical introductions need to be done or not ?

Two kinds of introductions can be distinguished, dependent and non dependent ones. One says dependent introductions for introductions depending on the head of the specification, non dependent for the others. Consider the case of a bounded predicate variable as head symbol of the specification corresponding to induction principles. In such a case, one do not want to introduce non-computational hypotheses depending on the predicate variable (dependent hypotheses) since they could change after the induction. So, if the head symbol of the specification of  $c$  is not bounded then all the logical introductions are performed, else (bounded predicate variables case) only the logical non dependent introductions. Indeed, considering for example proofs by induction, since a proof of  $A \rightarrow P(n)$  (with  $n \notin A$ ) is equivalent and harder than a proof of  $P(n)$  in the context of  $A$ , one chooses the second one. The equivalence of the two propositions is obvious but not the fact that the first could be harder than the second. Let us take the first case. Then, the transformation of the goal can give  $(A \rightarrow P(n)) \rightarrow (A \rightarrow Q(n))$ . The same transformation gives  $P(n) \rightarrow Q(n)$  in the second case. And, it is clear, that the first case is harder and even sometimes impossible. Having explained why the logical hypotheses have to be introduced, we explain why dependent hypotheses should not be introduced. The reason is to keep the link between the hypothesis and the conclusion. Indeed, if the hypothesis is put in the context, then it can no more be modified though the conclusion can, and the link can be lost. And, the most probable situation is that this link needs to be used in the proof.

Finally, one can remark the description of the method for the application can be applied for a variable (remind that a constant is considered like a variable but expanded in case of failure). Moreover, note the importance of retrieving the specification of a part of a program and the fact that, if it is not possible, then the use of annotations is motivated.

### 3.1.3 Recursions

This case is very similar to the previous one. But, let us see on a very simple example what could happen.

Suppose one wants to prove  $\forall n. \forall m. (n \leq m) \vee (n > m)$  with the following associated program<sup>4</sup>

---

<sup>4</sup>Note that the specification of a function which returns a boolean value is a disjunction (and not an existential).

```
[n:nat](<nat->bool>Match n with
  [m:nat] true
  [n':nat] [H:nat->bool] [m:nat]
    (<bool> Match m with
      false
      [m':nat] [H':bool] (H m'))).
```

Suppose one introduces the  $n$ . Then, one has an induction on  $n$ . The result has to be two new subspecifications for each case of the induction (the basic case and the induction case) associated to two new subprograms corresponding to the different cases (i.e. the different constructors of the inductive type of the induction element). The two specifications will be  $:\forall m.(0 \leq m) \vee (0 > m)$  and  $\forall n'.(\forall m.(n' \leq m) \vee (n' > m)) \rightarrow (\forall m.(Sn' \leq m) \vee (Sn' > m))$  with the two associated programs :

```
[m:nat] true
and
[n':nat] [H:nat->bool] [m:nat]
  (<bool>Match m with
    false
    [m':nat] [H':bool] (H m'))
```

So, if the program is  $Rec_I(m, P, lf)$ , one wants to eliminate the proof corresponding to the program  $m$ . If  $m$  is coarse (previous example) then it is trivial else one needs to retrieve the specification of  $m$  by the previous method in order to eliminate it.

But note here the problem of logical introductions. The heuristics are the following. If the specification depends on  $m$ , then only all the logical non dependent introductions are done (like for the application). Otherwise, all the logical introductions are done.

But, there is another problem which we can illustrate with the following example. Let us take one more time our example of division algorithm. Suppose we take the subprogram (**inf** is a boolean function deciding the order of two natural numbers) :

```
<nat*nat>if (inf b (S r)) then <nat,nat>((S q),0)
  else <nat,nat>(q,(S r))
```

The corresponding specification is  $(b > r) \rightarrow (n = b * q + r) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$ . The program suggests to do a case analysis on  $(\mathbf{inf} \ b \ (S \ r))$ . First, the previous heuristic clearly appears to be necessary :  $(b > r)$  and  $(n = b * q + r)$  are introduced as logical non dependent hypothesis. Second, the specification does not depend on  $(\mathbf{inf} \ b \ (S \ r))$ . Then, the case analysis will generate the two following identical subgoals (since the link with  $(\mathbf{inf} \ b \ (S \ r))$  is lost) :

$$\exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

$$\exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

But, these subgoals are not useful since the information about whether  $(\mathbf{inf} \ b \ (S \ r))$  is true or not is lost. The subgoals one would like to generate would rather be :

$$((\mathbf{inf} \ b \ (S \ r)) = \mathit{true}) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

$$((\mathbf{inf} \ b \ (S \ r)) = \mathit{false}) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

That is to introduce a dependency in the specification if the specification does not depend on the term of induction. So, if  $S$  is the current specification and  $t$  the proof term corresponding to  $m$ ,

then  $S$  is modified into  $(t = t) \rightarrow S$ . This just introduces a dependency without modifying the specification and allows to obtain probably more useful subgoals. Indeed, the problem comes from the fact that one looks for an adequate generalization of the goal and there are many ways of doing it. This choice is so only a heuristic.

Finally, one obtains as many subgoals as constructors of  $m$  and the different programs corresponding to the different constructors of the type of  $m$  are the elements of  $lf$ . Note, one more time, the importance of retrieving the specification of a part of a program since, if the specification of  $m$  cannot be retrieved, then the specification cannot be generalized.

## 4 Adding annotations

We saw the importance of retrieving a specification. Because it cannot always be done automatically, one would like to help the system. One need to add informations in the program. In fact, one wants to annot the program with comments which can be interpreted by the system. Let us add a new syntax for programs : one can annot any part of a program with the syntax  $(: \text{ a specification } :)$ . Between  $(:$  and  $)$ , one gives the specification one likes the program to have. This forces the system to take this information as a specification. Note that these annotations are available in a context of programs, that is to say that annotations are informations on a part of a program but can use programs variables. But, one can want to have a context of logical variables. So, one has to introduce another new syntax for  $\lambda$ -abstractions on logical variables  $[{\mathbf{x}}:\mathbf{L}]$ .

Let us give an example. Suppose we take the example of the division algorithm and particularly the step taken in 3.1.3. The specification is  $(b > r) \rightarrow (n = b * q + r) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$  and the program is `if (Sr<b) then (q,Sr) else (Sq,0)`. In fact, in *Coq*, it is written `<nat*nat>if (inf b (Sr)) then <nat,nat>((Sq),0) else <nat,nat>(q,(Sr))` with the constant `inf` being the decidability of the ordering relation on natural numbers.

Suppose first `inf` is declared as a variable without any specification, i.e. it is just a boolean value. Then, the generated subgoals are :

$$(inf\ b\ (Sr)) = true \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

$$(inf\ b\ (Sr)) = false \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

But, then, one has to prove that :

$$(inf\ b\ (Sr)) = true \rightarrow (b \leq (Sr))$$

$$(inf\ b\ (Sr)) = false \rightarrow (b > (Sr))$$

which is not easy if `inf` has no specification.

In a second case, if `inf` is declared as a program already specified by  $\forall n. \forall m. (n \leq m) \vee (n > m)$ , then the generated subgoals are :

$$(b \leq (Sr)) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

$$(b > (Sr)) \rightarrow \exists q \exists r (Sn = b * q + r) \wedge (b > r)$$

which are directly usable.

So, if `inf` is just a boolean function, one can explicitly indicate in the program the specification one wants for it by giving an annotation to this part of the program :

```
<nat*nat>if (inf b (Sr)) (: {(le b (Sr))}+{(gt b (Sr))} :)
  then <nat,nat>((Sq),0) else <nat,nat>(q,(Sr))
```

Then, this will generate the same subgoals as in the second case plus one needed to verify that the annotated program is consistent with its annotation. This last subgoal is the following :

$$(b \leq (Sr)) \vee (b > (Sr))$$

associated to the program (`inf b (Sr)`).

Now that we give this example, it is clear that the interpretation of annotations is that the specification of an annotated term is the annotation itself. So, let us give the following definition.

**Definition 2** *A program  $p$  is said to computationally solve a specification  $S$  if there exist logical assertions  $L$  such that, if the set  $L$  is proved, then there exists a proof  $T$  of  $S$  such that the program  $p$  is an extraction of  $T$ .*

Then one can state the following claim :

**Claim 1** *Every sufficiently annotated program can be computationally solved using the automation method described above, assuming some reasonable conditions on the dependencies in the specifications.*

Indeed, annotations allow to give specifications which cannot be retrieved automatically. So, if a program is sufficiently annotated then it can be computationally solved.

## 5 Optimizations

The method described above has been added to the system *Coq* (see [DFH<sup>+</sup>93]). The entire library of *Coq* examples has been tested [Par92]. But, programs are written in a  $F_{\omega}^{Ind}$  form which is not always very practical. We would like to have a language closer to ML. We have so introduced some optimizations in the formulation of the input program.

### 5.1 Recursive programs

The basic notion of *Coq* induction follows a primitive recursive scheme (or structural induction). But usual programs use a general recursion. *Coq* defines a well founded induction principle, which can be realized by a recursive program :

$$\forall P \forall R (\text{wellfounded } R) \rightarrow (\forall x (\forall y (R y x) \rightarrow (P y)) \rightarrow (P x)) \rightarrow \forall a.(P a)$$

The first optimization introduces a notion of recursive programs (general induction). A new syntax allows to write directly general recursive programs and it is translated in the previous well founded induction principle. So one needs an ordering relation  $R$  on which the induction is based and that has to be explicitly given. Indeed, this ordering relation cannot be retrieved automatically from the program and is the base of the well-foundedness of the induction. With this new syntax, one gives the ordering relation and uses directly general recursive programs.

The syntax is `<P>rec H (: order :)` for the ML `let rec H x = ... H y ...` with  $P$  the type of the result and `order` the ordering relation on which the well-founded induction is based.

Example : Euclidean division algorithm.

An Euclidean division algorithm can be expressed by the following program :

```
[b:nat](<nat*nat>rec div (: lt :)  
  [a:nat](<nat*nat>if (inf b a) then  
    (<nat*nat>let (q,r:nat) = (div (minus a b)) in  
      <nat,nat>((S q),r))  
    else <nat,nat>(0,a))).
```

if `lt` is the natural strict ordering relation on natural integers, `div` and `minus` the division and subtraction on natural integers and `inf` the decidability on the ordering relation on natural numbers.

## 5.2 Eliminations

This optimization is in fact the inversion of an optimization done during the extraction. Suppose you have an elimination in a recursive program. An optimization of the extraction method is the following : if an hypothesis appears in each different case of the elimination, the program is transformed by taking this hypothesis off from each case and placing it just before the elimination. It is the current way of writing programs. If the extracted program is `<A->B>Match n with [x:A]t1 ... [x:A]tm`, then the more natural optimized program has a different shape : `[x:A]<B>Match n with t1 ... tm`. So, our optimization is to consider that every hypothesis which is external regarding an elimination and used in this elimination has to be placed back into each case of the elimination. This optimization is important because, if it is not done, it can generate much harder proofs.

Let us take an example. Suppose one has an hypothesis  $x$  before an elimination on a variable  $n$  (like previously), the specification of  $x$  can depend on  $n$ . If  $x$  is not moved inside the elimination then there are many chances that the hypothesis will not be the one expected, because, probably, one wants a different hypothesis for each different cases of the elimination and the proof should be more complicated and even impossible.

## 5.3 Contraction of expressions

The purpose here is to allow programs to be given in a natural form.

Let us take the case of expressions like `if b then true else false`. These expressions are not natural in programs, that is to say that a programmer writes only `b`, but the proof has to be explicitly given like this to correspond to the good specifications. For example, if  $b$  is correct with respect to  $A$ , it can be correct with respect to another specification  $B$ . But, the proof  $C$  from which  $b$  is extracted is of type  $A$ , but not of type  $B$ . For example, if  $A$  is  $n = m \vee n \neq m$  and  $B$  is  $Sn = Sm \vee Sn \neq Sm$ , the program  $b$  realizes  $A$  and proves  $A$  and does not prove  $B$ . But, the program `if b then true else false` can be extracted from a proof of  $B$ . One has then to prove that  $n = m \rightarrow Sn = Sm$  and  $n \neq m \rightarrow Sn \neq Sm$ .

So, the optimization consists in writing natural programs without `if b then true else false` and transforming them when necessary. This is just explained on this simple example but can be generalized and, then, automatic transformations of programs are generated.

## 5.4 Singleton types

Let us take the example of the Ackerman function to explain the problem of singleton types.

The definition of the function is the following :

$$\begin{aligned} ack(0, n) &= n + 1 \\ ack(n + 1, 0) &= ack(n, 1) \\ ack(n + 1, m + 1) &= ack(n, ack(n + 1, m)) \end{aligned}$$

To express this definition we use in fact a ternary predicate :

```
Inductive Definition Ack : nat->nat->nat->Prop =
  Ack0 : (n:nat)(Ack 0 n (S n))
  | Ackn0 : (n,p:nat)(Ack n (S 0) p)->(Ack (S n) 0 p)
  | AckSS : (n,m,p,q:nat)(Ack (S n) m q)->(Ack n q p)->(Ack (S n) (S m) p).
```

Suppose now we want to prove that  $\forall n. \forall m. \exists p. Ack(n, m, p)$ . The type extracted from this expression is :  $nat \rightarrow nat \rightarrow sig\ nat^5$ . Suppose we want to give the program corresponding to this proof. It will be (in a CAML form) :

```
let rec ack n m = match n with
  0    -> (fun m -> m+1)
  | n'+1 -> (match m with
             0    -> ack n' 1
             | m'+1 -> ack n' (ack (n'+1) m')) ;;
```

This program written in a  $F_\omega$  form<sup>6</sup> is :

```
[n:nat](<nat->nat>Match n with
  [m:nat] (S m)
  [y:nat] [H:nat->nat] [m:nat]
    (<nat>Match m with
      (H (S 0))
      [m':nat] [H':nat] (H H'))))
```

The type of these last programs is :  $nat \rightarrow nat \rightarrow nat$ .

It is clear this type is not the same as the one extracted from the specification. But, suppose one develops the proof by hand of this specification, then the proof term will be (the *Li* represent logical parts which are not interesting from the program point of view) :

```
[n:nat](<[n0:nat] (m:nat){p:nat|(Ack n0 m p)}>Match n with
[m:nat](exist (x:nat)([p:nat](Ack 0 m p) x) (S m) L1)
[y:nat][H:(m:nat){p:nat|(Ack y m p)}][m:nat]
  (<[n0:nat]{p:nat|(Ack (S y) n0 p)}>Match m with
    <[s:{p:nat|(Ack y (S 0) p)}]{p:nat|(Ack (S y) 0 p)}>let
      (x:nat,p:(Ack y (S 0) x)) = (H (S 0)) in
      (exist (x0:nat)([p0:nat](Ack (S y) 0 p0) x0) x L2)
    [m':nat][H':sig nat]<[s:{p:nat|(Ack (S y) y0 p)}]{p:nat|(Ack (S y) (S y0) p)}>let
      (x:nat,p:(Ack (S y) y0 x)) = H' in
      <[s:{p0:nat|(Ack y x p0)}]{p0:nat|(Ack (S y) (S y0) p0)}>let
        (x':nat,p':(Ack y x x0)) = (H x) in
        (exist (x1:nat)([p1:nat](Ack (S y) (S y0) p1) x1) x' L3)))
```

<sup>5</sup>*sig nat* is the notation for the singleton type corresponding to *nat*, that is to say the inductive type with one constructor of type  $nat \rightarrow (sig\ nat)$ .

<sup>6</sup>The notation  $[x : A]t$  denotes the  $\lambda$ -term  $\lambda x : A. t$

with *exist* being the constructor of the inductive type *sig*. This proof has the same type as the specification. But, if one uses an isomorphism between  $A$  and  $\text{sig } A$ , one would like the proof and the program to have the same structure. And, this is not the case, since there are eliminations in the proof corresponding to the extraction of the structure of some terms of type  $\text{sig } \text{nat}$  which do not appear in the natural program (since there are only terms of type  $\text{nat}$ ). So, there are many transformations to do on the program to obtain a program able to generate the proof.

We define now a new notion of convertibility, a convertibility modulo  $A \equiv \text{sig } A$  (that we will call **weak convertibility**). The sense is larger than the usual convertibility (that we will call **strong convertibility**) and allows to accept programs that have just to be modified. The method of transformation is based on a comparison of the program and its specification.

A first typical case is when the program is a  $\lambda$ -abstraction whose type is  $A \rightarrow B$  when its specification type is  $\text{sig } A \rightarrow C$ <sup>7</sup>. The program is then transformed in a new  $\lambda$ -abstraction of type  $\text{sig } A \rightarrow B$ . For our example, let us take  $H$  of type  $\text{nat} \rightarrow \text{nat}$  but the corresponding specification  $\forall m. \exists p. \text{Ack}(y, m, p)$  is of type  $\text{nat} \rightarrow \text{sig } \text{nat}$ . So, the program is transformed with  $[H:\text{nat} \rightarrow (\text{sig } \text{nat})]$ . This implies that parts of programs will be no more well typed and this fact will help us to transform programs.

Another case of transformation is when the program is an application. There are two cases :

1. the program is ill typed. This implies that arguments are not of the good type  $A$  but of type  $\text{sig } A$  and have to be replaced. For our example, let take  $(H \ H')$ .  $H'$  is of type  $\text{sig } \text{nat}$  and  $H$  of type  $\text{nat} \rightarrow \text{sig } \text{nat}$ . This term is ill typed. So, it is transformed into :

`<sig nat>let (x:nat) = H' in (exist nat (H x)).`

We see the transformation is not complete. This is because of another problem. If we look at the specification of  $(H \ x)$  which is  $\exists p. \text{Ack}(y+1, m', p)$  and at the specification it is associated to which is  $\exists p. \text{Ack}(y+1, m'+1, p)$ , we see there are not identical. The program is then one more time transformed into :

`<sig nat>let (x:nat) = H' in <sig nat>let (x':nat) = (H x) in (exist nat x').`

This is in fact analogous to the transformation described in 5.3.

2. the program is well typed but of type  $A$  when its specification is of type  $\text{sig } A$ . For our example, let us take  $(S \ m)$ . It has type  $\text{nat}$  when its specification has type  $\text{sig } \text{nat}$ . The program is transformed into `(exist nat (S m))`.

This gives some typical cases of a method to transform programs which are weakly convertible but not strongly convertible to the specification. This allows to write programs in a more convivial form.

## 6 Comparaison with other works

As we said in the introduction, this method can be compared to the approach of [Pol92] and to the deliverables of [BM92]. [Pol92] describes a development of proofs and programs hand by hand. There is a separation of the programming language and of the logic language, which are two versions of the Calculus of Constructions. So, this is close to our approach but different in the sense that we first give the program and then develop automatically the proof. Moreover, there is a possibility of annotating programs to represent properties of these programs. With the deliverables approach,

---

<sup>7</sup> $C$  because it is  $B$  modulo  $A \equiv \text{sig } A$ .

proofs and programs are too developed hand by hand. But, there is no separation between the programming language and the logical language. Proofs and programs are developed together using strong sums in the Luo's Extended Calculus of Constructions [Luo90]. This is what are called deliverables. There is a distinction between two kinds of deliverables : first-order ones which do not allow to express a relation between the input and the output, and second-order ones, which allow the expression of such a relation. Moreover, deliverables are more rigid than our approach in the sense that one cannot consider specifications not of the form  $\forall x.(P\ x).\exists y.(Q\ x\ y)$ .

## 7 Conclusion

The method presented above allows to obtain a system in which one can write programs and prove them automatically to be correct with respect to a specification. In fact, this method is not completely automatic since one usually has to solve logical assertions on the program by hand. Moreover, one has to comment programs with annotations, not in all cases but often. This allows to guide the proof but is not always trivial. One should have a more natural way of writing annotated programs, for example a possibility to suppress the type information in the programs and to replace it by annotations. Moreover, the future versions of *Coq* with existential variables [Dow91, Dowar] would allow to delay the instantiation of some parameters (like the ordering relation in the recursive programs) which could be fixed by the user when he solves the logical lemmas. Moreover, one could increase the synthesis power by using unification.

## References

- [BM92] R. Burstall and J. McKinna. Deliverables : a categorical approach to program development in type theory. Technical Report 92-242, LFCS, October 1992. Also in [NPP92].
- [Con86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [DFH<sup>+</sup>93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide - Version 5.8. Technical Report 154, Projet Formel - INRIA-Rocquencourt-CNRS-ENS Lyon, May 1993.
- [Dow91] G. Dowek. *Démonstration Automatique dans le Calcul des Constructions*. PhD thesis, Université Paris 7, 1991.
- [Dowar] G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal of Logic and Computation*, To appear.
- [HN88] S. Hayashi and H. Nakano. *PX : A Computational Logic*. Foundations of Computing. MIT Press, 1988.
- [How80] W.A. Howard. The formulaes-as-types notion of construction. In J.R. Hindley, editor, *To H.B. Curry : Essays on Combinatory Logic , lambda-calculus and formalism*. Seldin, J.P., 1980.
- [Luo90] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

- [NPP92] B. Nordström, K. Petersson, and G. Plotkin, editors. *Proceedings of the 1992 workshop on types for proofs and programs*, June 1992.
- [Par92] C. Parent. Automatisation partielle du développement de programmes dans le système Coq. Master's thesis, Ecole Normale Supérieure de Lyon, June 1992.
- [PM89a] C. Paulin-Mohring. Extracting  $F_\omega$ 's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
- [PM89b] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris VII, 1989.
- [PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, March 1993. Also in research report 92-49, LIP-ENS Lyon, December 1992.
- [PMW92] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation-special issue on automated programing*, 1992. To appear.
- [Pol92] E. Poll. A programming logic for  $F_\omega$ . Technical Report 92/25, Eindhoven University of Technology, September 1992.