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

Abstract. The system *Coq* is an environment for proof development based on the Calculus of Constructions extended by inductive definitions. The specification of a program can be represented by a logical formula and the program itself can be extracted from the constructive proof of the specification. In this paper, we look at the possibility of inverting the extraction process. More precisely, we present a method which, given a specification and a program, builds the logical conditions to be verified in order to obtain a correctness proof of the program. We build a proof of the specification from the program from which the program can be extracted. Since some information cannot automatically be inferred, we show how to annotate the program by specifying some of its parts in order to guide the search for the proof.

1 Introduction

In this paper, we focus on the general problem of proving program correctness. Type theory is a particular framework for program development. Indeed, there is a constructive interpretation of proofs, i.e. proving a formula implies explicitly constructing a typed λ-term. This suggests a uniform framework for representing formulas, proofs and programs. Via the Curry-Howard isomorphism [How80], the notion of proofs and programs and the notion of types and specifications can be identified. To have a correct program in such a formalism, a proof of a logical formula $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ (called the specification) has to be derived. This 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 $Q(i,o)$. In some cases, the specification (essentially the pre- and post-conditions P and Q) and the program p are naturally given. A first idea is to prove $\forall x.P(x) \Rightarrow Q(x,(p\ x))$. But, this is in general laborious and implies reasoning about the executions of p. We propose here a method which justifies p by building a proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ from which p can be extracted. The program p is obtained by extraction from the proof by forgetting

* 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".

everything concerning the logical correctness of the program, in particular proofs of logical assertions and intermediate properties of sub-programs. Our method of building a proof from a program will necessary leave "holes" i.e. logical properties that will have to be satisfied to ensure the correctness of the program. That is why we will introduce a language of annotated programs to guide the method by indicating the specification of sub-programs.

The system *Coq* is an implementation of a particular type theory called the Calculus of Constructions enhanced with inductive definitions [PM93, DFH+93]. As we explained earlier, proofs of specifications can be developed and programs can be extracted from these proofs [PM89b]. The extraction in *Coq* is based on a realizability notion [PM89a]. Realizability is an interpretation of the computational content 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, including dependent types, and to extract correct programs from proofs. Some other systems like PX and NuPrl offer similar possibilities of extraction [HN88, Con86]. Both of them use untyped theories. More precisely, PX uses an untyped theory and, in NuPrl, the theory is typed but the extracted terms are untyped. PX uses a notion of proofs-as-programs which is not Curry-Howard but there are 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. A process of extraction can be expressed : redundant information can be hidden using the fact that if a is of type $\{x : A | P\ x\}$ then a is also of type A, but a consequence of this is that typing becomes undecidable.

Coq is thus a good framework for our method of building proofs from programs. The plan of our presentation will be in a first section to briefly present the Pure Calculus of Constructions. In a second section, we show an example of proof development in *Coq* to focus on what we are looking for. In a third section, we present the extraction mechanism and deduce from this our method. In a fourth section, we add inductive types. In a fifth section, we discuss some optimizations and then conclude.

2 The Pure Calculus of Constructions

We present here the Pure Calculus of Constructions and not the Calculus of Constructions with Inductive Definitions because it is simpler. It appears that, for our method, the adjunction of inductive types is neither a difficulty nor the central problem. We will just briefly talk about inductive definitions in a later section.

The Calculus of Constructions is a typed lambda-calculus. Thus, it is a language of terms and types where terms can be seen as representing proofs or programs and types as representing specifications or data types.

Types can be defined by their arity. For example, the type of natural numbers can be defined as `nat:Set`. Objects on these types can be defined. For example, the successor function can be defined as `S:nat->nat`. Then, predicates on these objects can be defined. For example, the predicate less-or-equal can be defined as `le:nat->nat->Prop` and if `n:nat` then `(le n n):Prop`. Now, predicates and objects can be mixed in specifications. For example, {`n:nat|(le n (S O))`}`:Set` is a specification to express $\exists n.(n \leq 1)$. A proof of such a specification is inhabited by a pair consisting of a natural number and a proof that this natural number satisfies the specification. For example, a proof can be the pair `(O,(le O (S O)))`.

We saw two different arities `Set` and `Prop`. The basic difference between `Set` and `Prop` is their meaning : both of them are used to mark terms (since they are types), but the first one indicates that the term has a computational content and the second that the term has no computational content. These marks on terms allow the extraction to treat differently computational terms and non computational terms and to forget all the non computational ones.

We can now explain the different possible formations for terms and types[2]. First, terms are variables or constants (both can be first-order or higher-order), abstractions (on any order variables) or applications (for example, instantiations of predicates). For instance, `x=4` is an instantiation of the predicate `[y:nat](x=y)` with the constant `4`.

`[x:A]B`	Abstraction of `B` w.r.t. `x` of type `A` (usually noted $\lambda x^A B$)
`(A B)`	Application of `A` to `B`
`(x:A)B`	Product of `B` w.r.t. `x` of type `A` (usually noted $\forall x : A.B$)
`A` $\rightarrow$ `B`	Type of functions from `A` to `B` ($\equiv$ `(x:A)B` when `x` not in `B`)
`Prop`	Type of propositions
`Set`	Type of specifications and data types

Second, types are propositional variables, products on types (dependent or not) or applications. One can quantify over predicates. For instance, the expression `(P:nat→Set)(P O)→((n:nat)(P n)→(P (S n)))→(n:nat)(P n)` is a case
of quantification over predicates. One can instantiate `P` with `[n:nat](n≥O)` and obtain the expression `(O≥O)→((n:nat)(n≥O)→((S n)≥O))→(n:nat)(n≥O)`.

This defines a typed λ-calculus usually called λC, which is the highest point in Barendregt's cube [Bar91], and includes polymorphism, higher order and dependent types. *Coq* is a system which implements this calculus and in the next section, we present a simple example of development in this system.

3 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

[2] The notations are given in the following.

and r such that they satisfy $a = b * q + r \ \wedge \ b > r$. 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 [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 remaining logical assertions can be solved, such as proving loop invariants are preserved, then the initial program can be certified correct with respect to the initial specification.

An ML program corresponding to a naive algorithm 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 and elimination tactics and resolution tactics. 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` and is developed in figure 1 (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. (* easy to resolve : b>0 and Sn=b*(Sq)+0 *)
Intros Gt.             (* Gt : b>Sr *)
Exists q. Exists (S r). (* easy to resolve : b>Sr and Sn=b*q+Sr *)
```

Fig. 1. A proof example in *Coq*

This proof is close to the mathematical one in the sense that one can retrieve the same 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 leaving to the user logical properties to prove to ensure the correctness of the program.

4 Extraction and inversion of extraction

Let us first recall how proofs and programs are represented in *Coq*. Proofs are typed λ-terms marked with information on their computational contents (i.e. `Set` for informative terms or `Prop` for logical terms). A *Coq* formula P can be represented by a judgment $\Gamma \vdash P : A$ where A can be `Set` or `Prop` (for first order), `TypeSet` or `Type` (for higher order). A *Coq* proof t of a specification P can be represented by a judgment $\Gamma \vdash t : P$.

The extraction is a function from proof terms to program terms which consists in forgetting from a proof term all the logical parts to obtain a program term, i.e. a typed λ-term with only informative parts. The extraction function is a forgetful function. There is an extraction at the level of terms and at the level of types. Both of these aspects forget the logical parts, but it is perhaps not so clear at the level of types. For example, one can have a type of lists of given length (for the proof point of view). This type is extracted into the type of lists (for the program point of view). The information related to the length is lost.

To present the extraction, we introduce a new judgment $\Gamma \vdash t \rightarrow p : P$ when P : `Set` whose interpretation is in Γ, t is a proof of P and p can be extracted from t.

Though there is an extraction on types and on proof terms, we focus on the extraction on proof terms. Our rules of extraction[3] are thus directed by the syntax of proofs. A type is noted `L` when it is a logical - non-computational - type and `I` when it is an informative - computational - type.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \rightarrow x \ : \ A}$$

$$\frac{\Gamma, x : L \vdash t \rightarrow p \ : \ B(x)}{\Gamma \vdash [x : L]t \rightarrow p \ : \ (x : L)B(x)}$$

$$\frac{\Gamma, x : I \vdash t \rightarrow p \ : \ B(x)}{\Gamma \vdash [x : I]t \rightarrow [x : \mathcal{E}(I)]p \ : \ (x : I)B(x)}$$

$$\frac{\Gamma \vdash t \rightarrow p \ : \ (x : I)B(x) \quad \Gamma \vdash u \rightarrow q \ : \ I}{\Gamma \vdash (t\ u) \rightarrow (p\ q) \ : \ B(u)} \quad (\Gamma \vdash I : Set)$$

$$\frac{\Gamma \vdash t \rightarrow p \ : \ (x : I)B(x) \quad \Gamma \vdash P \ : \ I}{\Gamma \vdash (t\ P) \rightarrow (p\ \mathcal{E}(P)) \ : \ B(P)} \quad (\Gamma \vdash I : TypeSet^4)$$

$$\frac{\Gamma \vdash t \rightarrow p \ : \ (x : L)B(x) \quad \Gamma \vdash u \ : \ L}{\Gamma \vdash (t\ u) \rightarrow p \ : \ B(u)} \quad (\Gamma \vdash L : Prop \ or \ \Gamma \vdash L : Type)$$

It appears clearly that the extraction is directed by the structure of proof terms and that terms are disappearing from proof terms to program terms. Higher order predicates are "lost", in the sense that only their informative parts are kept. For example, the predicate $\exists n : nat.(n \ is \ odd)$ is extracted into nat. If it is easy to go from the first to the second, it seems hard to retrieve the first from the second.

Since we want to generate proofs from programs, since higher order predicates are essential for the proof, and since they disappear in program terms, we are going to modify the extraction a little bit, and more precisely the rule about higher order predicates. We modify it into the following one.

$$\frac{\Gamma \vdash t \rightarrow p \ : \ (x : I)B(x) \quad \Gamma \vdash P \ : \ I}{\Gamma \vdash (t\ P) \rightarrow (p\ \{P\}) \ : \ B(P)} \quad (\Gamma \vdash I : TypeSet)$$

[3] The extraction function is denoted by the symbol $\mathcal{E}$.

[4] $P : I : TypeSet$ means P is a higher order informative predicate.

This allows to keep at the level of program terms (in comments) ALL the information about higher order predicates. Now, since we want to generate proofs from programs, we have to transform these rules such that they were directed by the syntax of program terms.

We just want here to mention a trivial case which is the case of **coarse programs**.

Definition 1. A program `p` is coarse with respect to a specification `S` if its type is exactly this specification, i.e. if `p:S` ($\equiv \mathcal{E}$`(S)`).

These are pointed out because they are very simple cases since they exactly correspond to the proof term.

In the following, we are going to give a new set of inference rules for a new judgment denoted by $\leadsto$. This set is supposed to be more deterministic, i.e. it will be directed by the syntax of program terms. For a judgment $\Gamma \vdash t' \leadsto p : S'$, if p is given it will be in general possible to infer t' and S' complete enough in the sense that if $\Gamma \vdash t \rightarrow p : S$ then t and t', and S and S' will be "equivalent" in a sense defined later.

Now, we can give the following rules[5] directed by the syntax of program terms with a new type of judgment denoted by $\leadsto$.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \leadsto x \ : \ A}$$

$$\frac{\Gamma, \boldsymbol{y} : \boldsymbol{L}, x : A \vdash t \leadsto p \ : \ B(\boldsymbol{y}, x)}{\Gamma \vdash [\boldsymbol{y} : \boldsymbol{L}][x : A]t \leadsto [x : \mathcal{E}(A)]p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A)B(\boldsymbol{y}, x)}$$

$$\frac{\Gamma \vdash t \leadsto p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A(\boldsymbol{y}))B(x, \boldsymbol{y}) \quad \Gamma \vdash \boldsymbol{l} : \boldsymbol{L} \quad \Gamma \vdash u \rightarrow q \ : \ A(\boldsymbol{l})}{\Gamma \vdash (t\ \boldsymbol{l}\ u) \leadsto (p\ q) \ : \ B(u, \boldsymbol{l})}$$

$$\frac{\Gamma \vdash t \leadsto p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A(\boldsymbol{y}))B(x, \boldsymbol{y}) \quad \Gamma \vdash \boldsymbol{l} : \boldsymbol{L} \quad \Gamma \vdash P \ : \ A(\boldsymbol{l})}{\Gamma \vdash (t\ \boldsymbol{l}\ P) \leadsto (p\ \{P\}) \ : \ B(P, \boldsymbol{l})}$$

Definition 2. A term t is logically more general than a term t', noted $t \succ_T t'$, iff $\exists \boldsymbol{x} : \boldsymbol{L}.\exists \boldsymbol{l}$ s.t. $t = [\boldsymbol{x} : \boldsymbol{L}](t'\ \boldsymbol{l})$.

Definition 3. A specification S is logically more general than a specification S', noted $S \succ_S S'$, iff $\exists \boldsymbol{x} : \boldsymbol{L}.\exists \boldsymbol{l}$ s.t. $S = (\boldsymbol{x} : \boldsymbol{L})S'(\boldsymbol{l})$.

Theorem 1 *If $\Gamma \vdash t \rightarrow p : S$ then $\exists t'.\exists S'$ s.t. $\Gamma \vdash t' \leadsto p : S'$ with $t \succ_T t'$ and $S \succ_S S'$.*

Proof. Proof by induction on the derivation of $\rightarrow$.

[5] We use a vectorial notation $(\boldsymbol{y} : \boldsymbol{L})$ for a list of variables $(y_1 : L_1) \ldots (y_n : L_n)$.

The new set of inference rules corresponds to a system where, from a program containing all the information on higher order predicates, a specification and a proof can be inferred. The theorem allows to prove that each proof having the same extracted program (annotated with all the information on higher order predicates) can be deduced from the inferred proof.

Practically, this means that if a user gives a program , then a specification S' and a proof t' can be inferred. Then, the initial specification S is logically more general than S' and $S = (\boldsymbol{x} : \boldsymbol{L})S'(\boldsymbol{l})$. To obtain the searched proof, $\boldsymbol{x}$ and $\boldsymbol{l}$ are needed. The $\boldsymbol{x}$ are known and the problem is to retrieve the $\boldsymbol{l}$. It can be resolved by unification : there are rigid constraints only on logical variables and it is easy.

The problem is now to know if these rules are deterministic from the program point of view. Let us see the four cases separately, each one illustrated with an example. The case of variables is trivial and we do not say anything about it.

4.1 λ-abstractions

We give once more the corresponding rule.

$$\frac{\Gamma, \boldsymbol{y} : \boldsymbol{L}, x : A \vdash t \leadsto p \ : \ B(\boldsymbol{y}, x)}{\Gamma \vdash [\boldsymbol{y} : \boldsymbol{L}][x : A]t \leadsto [x : \mathcal{E}(A)]p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A)B(\boldsymbol{y}, x)}$$

As one knows the correspondence between the program `p` and the specification `S`, one knows that if the program is a λ-abstraction `[x:`$\mathcal{E}$`(A)]p` then the specification is a product `(`$\boldsymbol{y}$`:`$\boldsymbol{L}$`)(x:A)B(`$\boldsymbol{y}$`,x)`. These logical products in the specification correspond to logical λ-abstractions in the proof terms. These λ-abstractions can thus all be retrieve from the specification in a deterministic way.

Practically, logical introductions not indicated in the program have to be done at the proof level. 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.

Let us consider the problem on an example. Consider the previous division algorithm written in the *Coq* syntax[6] :

```
[b:nat][a:nat]
  <nat*nat>Match a with
    (* O *) <nat,nat>(O,O)
    (* 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),O)
                              else <nat,nat>(q,(S r)).
```

[6] The notation `<P>Match x with` is the case analysis.

whose specification is $\forall b.b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r)$. Since the program is a λ-abstraction, this suggests to introduce the b and then generate 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.

4.2 Applications

There are two rules corresponding to the application of a program to a program or to a type.

$$\frac{\Gamma \vdash t \leadsto p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A(\boldsymbol{y}))B(x, \boldsymbol{y}) \quad \Gamma \vdash \boldsymbol{l} : \boldsymbol{L} \quad \Gamma \vdash u \rightarrow q \ : \ A(\boldsymbol{l})}{\Gamma \vdash (t \ \boldsymbol{l} \ u) \leadsto (p \ q) \ : \ B(u, \boldsymbol{l})}$$

$$\frac{\Gamma \vdash t \leadsto p \ : \ (\boldsymbol{y} : \boldsymbol{L})(x : A(\boldsymbol{y}))B(x, \boldsymbol{y}) \quad \Gamma \vdash \boldsymbol{l} : \boldsymbol{L} \quad \Gamma \vdash P \ : \ A(\boldsymbol{l})}{\Gamma \vdash (t \ \boldsymbol{l} \ P) \leadsto (p \ \{P\}) \ : \ B(P, \boldsymbol{l})}$$

If the program is $(p \ q)$ then there are two cases : the argument is a higher order predicate (between comments) or not.

The first case is easy. Indeed, the problem is to find the higher order predicate and it is given in comments in the program. There is thus no problem.

The second case is harder. If one supposes the specification of p is known, then the problem is to retrieve the specification of q. With the given rules it is not possible because we do not want to be directed by the syntax of the proof term but rather by the syntax of the program term. One strategy is to look for a specification S of q with the inference rules of $\leadsto$. But, the specification S is then surely not the right one but an equivalent one in the sense of equivalence on specifications $(A(\boldsymbol{l}) = (\boldsymbol{x} : \boldsymbol{L'})(S \ \boldsymbol{l'}))$. Moreover, this implies to unify S with $A(\boldsymbol{x})$ and this is not deterministic.

In practice, this is what is done. In many practical cases, the unification gives a solution (by using heuristics). If this fails, then our method fails.

Thus, we have a method to retrieve proofs from programs which is deterministic (modulo some heuristics on unification) and complete since it inverts the extraction process. But, we suppose that all the higher order predicates are given. Naturally, it is not what we want to impose to the user. We want to allow the user not to give the higher order predicates and to find them mechanically. Once more, this is a problem of unification but now involving informative predicates. Since this is not decidable and since *Coq* uses first order unification, we will be able via heuristics to solve it in some cases but, it will fail in some other cases.

Let us take a successful example. Suppose one wants to prove :

$\forall n.\exists m.((Sm = n) \vee (n = m = 0))$. A program can be :

```
[n:nat](nat_rec nat O [y:nat][H:nat]y)
```

where `nat_rec` is the induction principle on natural numbers whose type is `(P:nat→Set)(P O)→((n:nat)(P n)→(P (S n)))→(n:nat)(P n)`. Note that we do not give explicitly the predicate `P` and that we only give the type `nat`.

Suppose `n` is introduced. Then, the specification is $\exists m.((Sm = n) \vee (n = m = 0))$ with the program :

```
(nat_rec nat O [y:nat][H:nat]y)
```

The desired result is two subgoals with two subprograms :

- $\exists m.((Sm = O) \vee (O = m = O))$ with `O`.
- $\forall n.\exists m.((Sm = n) \vee (n = m = O)) \rightarrow \exists m.((Sm = Sn) \vee (Sn = m = O))$ with `[y:nat][H:nat]y`.

The program is an application with the constant `nat_rec` as head. Its specification is known and is `(P:nat→Set)(P O) → ((n:nat)(P n) → (P (S n))) → (n:nat)(P n)`. To generate the subgoals, one needs to instantiate the predicate `P` of `nat_rec`. Since one knows the specification to prove, one can use it to find this predicate. The head of the type of `nat_rec` i.e. `(P n)` and the specification $(\exists m.(Sm = n)) \vee (O = n)$ have thus to be unified (in this case, a higher-order unification is necessary). After that, `P` can be instantiated by $[n](\exists m.(Sm = n)) \vee (O = n)$ and allows to generate the two previous subgoals.

Thus, in many practical cases, our method will be able to generate a proof of a program for a given specification leaving logical properties to prove to ensure the correctness. We identify the cases of failure as unification problems for higher order predicates. A solution is a proposition of annotated programs. This will be presented in a next section but we first discuss the problem of adding inductive types.

5 Adding inductive types

We just talked about Pure Calculus of Constructions. The system *Coq* implements this calculus enhanced with inductive definitions. We will here discuss the addition of inductive definitions in our problem.

First of all, adding inductive definitions gives two more rules from the extraction point of view expressed in the following, one for constructors, one for dependent elimination. The notations are :

$\mathsf{Ind}(X : A)\{C_1(X)|\ldots|C_n(X)\}$: inductive type whose constructors are C_i.

$\mathsf{Constr}(i, Ind)$: i-th constructor of the inductive type Ind.

$C\{Ind, P, c\}$: term defined by induction over the structure of the type of constructor C (see [PM93]).

$\mathsf{Elim}_{Ind}(c, P)\{f_1|\ldots|f_n\}$: notation for $<P>Match\ c\ with\ f_1 \ldots f_n$ with $c : Ind$.

The rules are the following :

$$\frac{Ind \equiv \mathsf{Ind}(X:A)\{C_1(X)|\ldots|C_n(X)\}}{\Gamma \vdash \mathsf{Constr}(i, Ind) \rightsquigarrow \mathsf{Constr}(i, \mathcal{E}(Ind))\ : C_i(Ind)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash c \rightsquigarrow p\ :\ (Ind\ \boldsymbol{a}) & (\forall i = 1\ldots n)\\ \Gamma \vdash P : (\boldsymbol{x}:\boldsymbol{A})(Ind\ \boldsymbol{x}) \to s' & \Gamma \vdash f_i \rightsquigarrow q_i\ :\ C_i\{Ind, P, \mathsf{Constr}(i, Ind)\}\end{array}}{\Gamma \vdash \mathsf{Elim}_{Ind}(c, P)\{f_1|\ldots|f_n\} \rightsquigarrow \mathsf{Elim}_{\mathcal{E}(Ind)}(p, \mathcal{E}(P))\{q_1|\ldots|q_n\}\ :\ (P\ \boldsymbol{a}\ c)}$$

The first rule is not as trivial as it seems. Indeed, the problem is to retrieve from the inductive type $\mathcal{E}(Ind)$ the inductive type Ind. This uses the property that for all S s.t. $S \succ_S C_i(Ind)$, the conclusion of S is the type Ind applied to logical arguments. With this property, the type Ind can be retrieved from the specification.

For the second rule, as for application of higher order predicates, we modify the elimination rule to keep elimination predicates.

$$\frac{\begin{array}{ll}\Gamma \vdash c \rightsquigarrow p\ :\ (Ind\ \boldsymbol{a}) & (\forall i = 1\ldots n)\\ \Gamma \vdash P : (\boldsymbol{x}:\boldsymbol{A})(Ind\ \boldsymbol{x}) \to s' & \Gamma \vdash f_i \rightsquigarrow q_i\ :\ C_i\{Ind, P, \mathsf{Constr}(i, Ind)\}\end{array}}{\Gamma \vdash \mathsf{Elim}_{Ind}(c, P)\{f_1|\ldots|f_n\} \rightsquigarrow \mathsf{Elim}_{\mathcal{E}(Ind)}(p, \{P\})\{q_1|\ldots|q_n\}\ :\ (P\ \boldsymbol{a}\ c)}$$

If the predicates are given, it is trivial. But, as for applications, we want not to impose to the user to give the elimination predicates. The problem is thus to retrieve the right predicate and we will just show on examples what happens.

Let us take our first example of division algorithm. Suppose we take the sub-program (`inf` is a boolean function deciding the order of two natural numbers, `if` is a macro for the elimination on booleans) :

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

The corresponding specification is $(b > r) \to (n = b*q+r) \to \exists q \exists r\ (Sn = b*q+r) \wedge (b > r)$. The program suggests to do a case analysis on `(inf b (S r))`.

If `(inf b (S r))` has type $(b < Sr) \vee (b \geq Sr)$ then the elimination predicate is `(P:bool->Set)(P (b<Sr))->(P (b>=Sr))->(B:bool)(P B)`. Since the goal is $(b > r) \to (n = b*q+r) \to \exists q \exists r\ (Sn = b*q+r) \wedge (b > r)$, `P` can be instantiated with $[B](b > r) \to (n = b*q+r) \to \exists q \exists r\ (Sn = b*q+r) \wedge (b > r)$. But, this is very heavy. The heuristic is to introduce in the context all the logical informations that do not depend on the elimination term. Indeed, a proof by induction on n of $A \Rightarrow B(n)$ generates a subgoal $(A \Rightarrow B(n)) \Rightarrow (A \Rightarrow B(Sn))$ when a proof by induction on n of $B(n)$ in the context A generates $B(n) \Rightarrow B(Sn)$ in the context A. Clearly, the two proofs are equivalent but the second one is easier. That is why we use this heuristic. In our example, we introduces all the logical hypotheses that do not depend on `(inf b (S r))`. Then, `P` can be instantiated with $[B]\exists q \exists r\ (Sn = b*q+r) \wedge (b > r)$. The generated subgoal are :

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

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

But, if `(inf b (S r))` has type `bool` then the elimination predicate is `(P:bool->Set)(P true)->(P false)->(B:bool)(P B)` and the generated subgoals are :

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

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

These subgoals are not useful since the information as to whether `(inf  b (S r))` is true or not is lost. The subgoals one would like to generate would rather be :

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

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

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 $(t = t) \to S$. 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 thus 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.

Note that the previous heuristic points out the fact that recursion without parameters may or may not correspond to proof by induction without parameters, depending on the computational meaning of those parameters. In the previous example, it corresponds to proof by induction with a logical parameter which implies the generalization of the goal.

Note that the heuristic of generalizing the goal is only used for proofs by case analysis and it does not affect the provability of the goal in this case since it consists in adding logical parameters to the case analysis.

Afterwards, we would like to mention here another problem due to the introduction of inductive definitions. In our description of the problematic case of applications, we described the problem of retrieving the specification of the head of the application. Suppose this head is now a recursion. Then, the problem has many solutions since there are many possible predicates for the recursion. Thus, once more, a heuristic is applied and the goal is generalized to reduce to a known case i.e. the case of recursion. This heuristic corresponds to the following rule :

$$\frac{\Gamma \vdash \mathsf{Elim}_I(m, P)\{lf\} : (x : A)\ B \quad \Gamma \vdash a : A}{\Gamma \vdash (\mathsf{Elim}_I(m, P)\{lf\}\ a) : B[x/a]}$$

To conclude, the addition of inductive types does not affect the validity of theorem 1. But, if we do not want the user to give all predicates, then there are once more heuristics which can fail. This leads us to the notion of annotated programs.

6 Adding annotations

We have already discussed the importance of indicating higher order predicates in programs. We saw it is in some cases possible to not give them in a program and to retrieve them automatically. We want here to give a new syntax to the user to annotate a program with higher order predicates or elimination predicates. Annotations are then comments which can be interpreted by the system. The new syntax for programs is that one can annotate any part of a program with the syntax `(:` a specification `:)`. Between `(:` and `:)`, one gives the specification one would like the program to have. This forces the system to take this information as a specification. The corresponding rule is :

$$\frac{\Gamma \vdash t \leadsto p : S \quad S \succ_S S'}{\Gamma \vdash t \leadsto p\ (:\ S\ \ :) : S'}$$

Note that these annotations can contain free variables. These free variables can be program variables but can be logical variables too. One must thus introduce another new syntax for λ-abstractions on logical variables `[{x:L}]`. The corresponding rule is :

$$\frac{\Gamma, x : L \vdash t \leadsto p : S}{\Gamma \vdash [x : L]t \leadsto [\{x : L\}]p : (x : L)S}{}^{7}$$

Note that these rules do not affect the validity of the theorem 1. This just adds two trivial rules.

Let us give an example. Suppose we take the division algorithm at a particular step. The specification is $(b > r) \to (n = b{*}q{+}r) \to \exists q \exists r\ (Sn = b{*}q{+}r) \wedge (b > r)$ 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),O)
                         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 :

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

[7] A logical λ-abstraction in a program always corresponds to the first logical product in the specification. If one wants to mention the second logical product, one needs to indicate the first and the second corresponding λ-abstraction.

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

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),O) 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. This can be useful to direct the tactic in its choices i.e. to explicitly give predicates which could perhaps not be automatically found by the tactic.

7 Optimizations

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.

7.1 Constants

First, and this is not exactly an optimization, constants have a particular status. Two alternatives appear : the constant can be taken as an object hiding a more complex structure, an already specified program, and allowing us to not repeat work already done. This implies that one forgets this complex structure. But, in some cases, this structure can be useful. So, our decision was to try our method with the unexpanded constant and if it fails to retry with the expanded constant.

A future alternative could be to give the possibility to the user to control the expansion of constants.

7.2 Recursive programs

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

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

The first optimization is only syntactic sugar. It 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 basis 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>(O,a))).
```

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

The corresponding specification can be $\forall b.(b > 0) \Rightarrow \forall a.\exists q \exists r.(a = b * q + r) \wedge (b > r)$. The following proof obligations are generated :

- $(wellfounded\ lt)$ as a logical subgoal
- $\forall a.\exists q \exists r.(a = b * q + r) \wedge (b > r)$ associated to the program `(<nat*nat>if (inf b a) then ...`

7.3 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[8].

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 the example of the Quicksort. It can be expressed by the specification $\forall l.\exists m.(sorted\ m) \wedge (permut\ l\ m)$ with the following program where Splitting represent a previous program which splits a list into two lists from a pivot point :

```
(<list>rec quick (: ltl :)
   [l:list](<list>Match l with
       (* nil *) nil
  (* cons a m *) [a:A][m:list][h:list]
                   <list>let (l1,l2:list) = (Splitting a m) in
                          (mil a (quick l1) (quick l2)))).
```

which is a macro for :

```
[x:list](well_founded_induction list list
  [l:list][quick:list->list](<list>Match l with
       (* nil *) nil
  (* cons a m *) [a:A][m:list][h:list]
                   <list>let (l1,l2:list) = (Splitting a m) in
                          (mil a (quick l1) (quick l2))) x).
```

and for the proof, one would like to have :

```
[x:list](well_founded_induction list list
  [l:list](<(list->list)->list>Match l with
     (* nil *) [quick1:list->list]nil
(* cons a m *) [a:A][m:list][h:(list->list)->list]
                [quick2:list->list]
                 <list>let (l1,l2:list) = (Splitting a m) in
                        (mil a (quick2 l1) (quick2 l2))) x).
```

Indeed, in the second program, the quick hypothesis has type $\forall y.(ltl\ y\ l) \Rightarrow \exists m.(sorted\ m) \wedge (permut\ y\ m)$. This cannot be used in the proof since one knows nothing about l.

In the third program, the quick hypothesis has two different occurrences quick1 and quick2 depending on what is l. If l is nil, the quick1 hypothesis has type $\forall y.(ltl\ y\ nil) \Rightarrow \exists m.(sorted\ m) \wedge (permut\ y\ m)$. If l is (cons a m), the quick2 hypothesis has type $\forall y.(ltl\ y\ (cons\ a\ m)) \Rightarrow \exists m.(sorted\ m) \wedge (permut\ y\ m)$. Then, in the second case of the induction, quick2 can be used.

The optimization consists thus in taking the first program and transforming it into the third one.

[8] Note that this optimization can only be done with non recursive eliminations

7.4 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 right 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, and 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 B but a proof of A is not a proof of 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.

7.5 Singleton types

Let us take the example of the Ackermann 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 O n (S n))
   | Ackn0 : (n,p:nat)(Ack n (S O) p)->(Ack (S n) O 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 : $nat \rightarrow nat \rightarrow sig\ nat$[9]. 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')) ;;
```

[9] *sig nat* is the notation for the singleton type corresponding to *nat*, that is to say the inductive type with one constructor of type $nat \rightarrow (sig\ nat)$.

This program written in a F_ω form[10] 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 O))
                        [m':nat][H':nat](H H')))
```

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

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

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

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 $sig\ A$, 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 $sig\ nat$ which do not appear in the natural program (since there are only terms of type 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 $A \equiv sig\ A$ (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 λ-abstraction whose type is $A \to B$ when its specification type is $sig\ A \to C$[11]. The program is then transformed in a new λ-abstraction of type $sig\ A \to B$. For our example, let us take `H` of type $nat \to nat$ but the corresponding specification $\forall m.\exists p.Ack(y, m, p)$ is of

[10] The notation $[x : A]t$ denotes the λ-term $\lambda x : A.t$

[11] C because it is B modulo $A \equiv sig\ A$.

type $nat \to sig\ nat$. So, the program is transformed with `[H:nat->(sig nat)]`. This implies that parts of programs will no longer be 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 right type A but of type $sig\ A$ and have to be replaced. For our example, consider `(H H')`. `H'` is of type $sig\ nat$ and `H` of type $nat \to sig\ nat$. 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 $\exists p.Ack(y+1, m', p)$ and at the specification it is associated to which is $\exists p.Ack(y+1, m'+1, p)$, we see there are not identical. The program is then once more 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 7.4.
2. the program is well typed but of type A when its specification is of type $sig\ A$. For our example, let us take `(S m)`. It has type nat when its specification has type $sig\ nat$. 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.

8 Results

8.1 Theoretical results

We give now some results about the "meta-theory" of the method described above.

Proposition 1 (Validity). *If a program* `p` *associated to a specification* `S` *leads to logical lemmas by the use of the Program tactic and if these lemmas are provable, then* `S` *is provable and the program extracted from the generated proof is the input program* `p` *modulo the weak convertibility.*

Proof. Trivial by the theorem 1. □

Remark. The generated logical lemmas are atomic since, for a specification $\forall x.(P\ x) \Rightarrow \exists y.(Q\ x\ y)$ and a program `p`, subspecifications of the initial specification associated to subprograms of the initial program are generated until only logical lemmas are left.

Indeed, the tactic iterates a method on the structure of the program term. A complete proof iterates this until only logical lemmas are left. Then, the user solves these lemmas. The user can influence the proof between two steps of the tactic by unfolding constants, generalizing goals ...

Proposition 2 (Completeness). *If a specification* `S` *is provable and if an associated program* `p` *explicitly contains all the information about higher order predicates then logical generated lemmas are provable.*

Proof. We will give an informal proof.

Suppose `S` is provable, then there exists a program extracted from a proof of `S`. The tactic can be applied with this program. It may succeed if all the logical informations can be retrieved. We saw problematic informations are informations about predicates. If one supposes that the input program of the tactic contains all the informations about predicates via annotations, then all logical informations can be retrieved by the tactic and logical lemmas can be generated. Then, since the program is a skeleton of the desired proof and since `S` is provable, logical lemmas are provable. □

8.2 Practical results

The method described above has been added to the system *Coq* (see [DFH+93]) with a tactic called the Program tactic. The entire library of *Coq* examples has been tested [Par92]. This means that we took every example of the library and try to use the Program tactic for each computational lemma or theorem of each example. This done, we have been able to rewrite all computational theorems with our tactic. We gave the programs and solved the logical lemmas.

We prove algorithms on graphs (Floyd, Dijkstra, Warshall), trees (insertion in avl trees), sorts (Quicksort, Mergesort, Heapsort), Ackermann function, natural numbers (euclidean division) and a tautology checker. Some of them were very quick to prove with our method since there were not a lot of logical lemmas (even sometimes none). Others (like the algorithms on graphs) were harder to prove (subtle logical lemmas) and require the use of annotations. The main problem was to find when where and how to use annotations.

9 Conclusion

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 in hand. There is a separation between the programming language and 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, proofs and programs are also 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 $\forall x.(P\ x) \Rightarrow \exists y.(Q\ x\ y)$.

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. 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. The future versions of *Coq* with existential variables [Dow91, Dow] 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. This is under development in a future version of *Coq*. Finally, we hope to be able to use ProPre [MS92] which has been integrated into *Coq*. This allows to define functions via equations. This definition is transformed into a primitive recursive definition which can be used as input of our tactic. This can allow to write more easily input programs of our tactic. In the current state, this is possible but for only a restricted number of functions. We hope the number of functions easily expressible with Propre will increase and then allow us to use it in more general way.

References

[Bar91] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, September 1991.

[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+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 58. 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.

[Dow] 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.

[MS92] P. Manoury and M. Simonot. *Des preuves de totalité de fonctions comme synthèse de programmes.* PhD thesis, Université PARIS 7, December 1992.

[NPP92] B. Nordström, K. Petersson, and G. Plotkin, editors. *Prooceedings of the 1992 worshop 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_ω'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 Fω. Technical Report 92/25, Eindhoven University of Technology, September 1992.

This article was processed using the LaTeX macro package with LLNCS style