Synthesizing proofs from programs in the Calculus of Inductive Constructions

Catherine Parent *

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

Abstract. We want to prove "automatically" that a program is correct with respect to a set of given properties that is a specification. Proofs of specifications contain logical parts and computational parts. Programs can be seen as computational parts of proofs. They can then be extracted from proofs and be certified to be correct. We focus on the inverse problem : is it possible to reconstruct proof obligations from a program and its specification ? The framework is the type theory where a proof can be represented as a typed λ-term [Con86, NPS90] and particularly the Calculus of Inductive Constructions [Coq85]. A notion of coherence is introduced between a specification and a program containing annotations as in the Hoare sense. This notion is based on the definition of an extraction function called the weak extraction. Such an annotated program can give a method to reconstruct a set of proof obligations needed to have a proof of the initial specification. This can be seen either as a method of proving programs or as a method of synthetically describing proofs.

1 Introduction

This paper talks about proving correctness of functional programs and, in particular, how to prove "automatically" that a program is correct with respect to a set of given properties that is a specification.

As an example, the specification of a division algorithm can then be expressed by the logical formula $\forall a, b.(b > 0) \rightarrow \exists q, r.(a = b*q+r) \wedge (b > r)$. A constructive proof of this specification describes a method that transforms a, b and a proof of $b > 0$ into two objects q, r and a proof of $(a = b * q + r) \wedge (b > r)$. In such a proof, two parts can be distinguished : a computational part (how to construct q and r from a and b) and a logical part (why $a = b * q + r$ is true knowing $b > 0$). The proof follows an induction scheme on a. The computational part of the proof corresponds to the case analysis on a and to the construction of q and r from a and b in the two cases, by recursion on $a - 1$ if it is not equal to 0. The logical part of the proof corresponds to the justification that $a = b * q + r$ is true in the two cases.

* 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".

A proof then represents a functional program which validates the specification. However, in general, to have more efficient programs, all the logical parts of a term are suppressed. This is the principle of an extraction function [PM89b, PM89a] that suppresses some parts of proofs and that we call the strong extraction.

An extraction function forgets all the non-computational parts of a proof to give a program. Different proofs can then correspond to different programs. Indeed, there can be many different proofs for a specification of a sorting algorithm. Each different proof can give a different algorithm for sorting a list.

It is also important to note that an extracted program is a skeleton of its corresponding proof. For the example of the division, the ML program that could be extracted is :

```
let rec div a b = match a with
                   0   -> (0,0)
                 | n+1 -> let (q,r) = div n b in
                            if b<=(r+1) then (q+1,0)
                                        else (q,r+1) ;;
```

In this paper, we focus on the inverse problem. Given a specification and a description of an algorithm that solves the problem, is it possible to generate proofs obligations that allow to reconstruct the proof of the specification ? This approach has two applications : proof of functional programs and proofs description.

We choose a framework where proofs are seen as programs. This is the Curry-Howard isomorphism [How80]. Each proof is a priori seen as a program. Nevertheless, some propositions can be distinguished if their proofs are not interpreted as computational ones. Then, a program can be extracted from a proof.

The reconstruction of a proof from a program is then particularly natural. Control structures in the program naturally correspond to proofs methods (recursion corresponds to induction, pattern matching to case analysis ...). Reconstructing proof obligations then corresponds to retrieve more or less a weakest precondition (in the Hoare sense) and, so, subspecifications for subparts of the program. In the division example, the following can be found :

```
let rec div a b =
match a with
   0   -> { (0 = b ∗ 0 + 0) ∧ (b > 0) }
          (0,0)
| n+1 -> { ∃q, r(n = b ∗ q + r) ∧ (b > r) → ∃q, r(n + 1 = b ∗ q + r) ∧ (b > r) }
          let (q,r) = div n b in
             { (n + 1 = b ∗ q + r) ∧ (b > r) }
             if b<=(r+1)
             then { (n + 1 = b ∗ (q + 1) + 0) ∧ (b > 0) }
                  (q+1,0)
             else { (n + 1 = b ∗ q + r + 1) ∧ (b > r + 1) }
                  (q,r+1);;
```

We choose a particular framework that is the Calculus of Inductive Constructions [Coq85, PM89b] which is a typed λ-calculus with polymorphism, higher-order and dependent types. We focus on the *Coq* [DFH$^+$93, CCF$^+$94] system which is an implementation of this calculus. It is a system for formalizing and checking the mathematical reasoning [Bar91, ML84]. It contains a specification language and a programming language. By the Curry-Howard isomorphism , a proof can be represented by a typed λ-term whose type is the proposition that it proves. In the Calculus of Inductive Constructions, the specification of a program is regarded as a proposition.

In [Par93], we described an implemented method that generates proof obligations from a program and a specification. Here, we give a theoretical description of this method. More precisely, a new extraction function called the weak extraction is defined. The strong extraction applies on terms and types. It suppresses in both of them all logical terms. The weak extraction applies only on terms and keeps all the informations in types. Then, a new notion of typing on weak extracted terms is necessary. It can be proved that a program, coherent with a given specification, can be transformed into a proof of an equivalent specification modulo some logical lemmas. This corresponds to invert the weak extraction.

This problem is not simple because the framework is very general. The Calculus of Inductive Constructions is a powerful language. Some restrictions on the proof system have to be made. Given a program p that is exactly a trace of a proof P, it can be proved that the generated logical lemmas have a proof in P. This method is the basis of an effective heuristic method corresponding to a tactic in the *Coq* system described in [Par93, Par95]. This heuristic method considers strong extracted programs and synthesizes types by unification from the initial specification.

The plan is in three main parts. First, we present the Calculus of Inductive Constructions and the extraction of [PM89b]. Secondly, we show the weak extraction, why and how it can be inverted. Thirdly, we explain a derived heuristic method as a consequence of the inversion. Finally, a non trivial example is developed in the appendix, for people familiar with a *Coq* syntax.

2 Extraction in the Calculus of Inductive Constructions

This section presents the framework. The proof language is the Calculus of Inductive Constructions. We first present the Pure Calculus of Constructions and how it can be extended with primitive inductive types. Then, we explain the motivations for extracting programs and present the extraction of [PM89a].

2.1 The Calculus of Constructions

We present a variant due to [PM89b] of the standard Calculus of Constructions [Coq85]. In this variant, two sorts are distinguished : $Set : TypeSet$ and $Prop : Type$. These two sorts allow to separate computational terms and logical terms by marking them. Terms marked with Set are computational terms and terms marked with *Prop* are logical terms. The inference rules for the calculus are then the following (types are generally in upper case, terms in lower case).

Definition 1.

$$\frac{}{[]\ well\ formed}$$

$$\frac{\Gamma \vdash A : s}{\Gamma[x : A]\ well\ formed}\ s \in \mathcal{S} = \{Set, Prop, TypeSet, Type\}$$

$$\frac{\Gamma\ well\ formed}{\Gamma \vdash Set : TypeSet} \quad \frac{\Gamma\ well\ formed}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\ well\ formed \quad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma[x : A] \vdash B : s'}{\Gamma \vdash (x : A)B : s'}\ (s, s') \in \mathcal{R} = \mathcal{S} \times \mathcal{S}$$

$$\frac{\Gamma[x : A] \vdash t : B \quad \Gamma \vdash (x : A)B : s}{\Gamma \vdash [x : A]t : (x : A)B}$$

$$\frac{\Gamma \vdash t : (x : A)B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : B[x \leftarrow t']}$$

$$\frac{\Gamma \vdash A' : s \quad \Gamma \vdash t : A \quad A =_\beta A'}{\Gamma \vdash t : A'}\ s \in \mathcal{S}$$

2.2 Inductive Types

We consider the Calculus of Constructions without universes. Since data types cannot be easily coded in this calculus, we consider the addition of primitive inductive types. This allows to define concrete data types in a more natural way. For example, natural numbers can be defined as an inductive structure with two constructors 0 and S (natural numbers are constructed from 0 with the successor). This section presents the syntax and rules for inductive types. Our presentation is much inspired by [PM93, PC89], but we treat explicitly the parameters of inductive types. In [PM93], inductive types are presented without parameters and these ones are treated as λ-abstractions.

Definition 2. Arity :
An arity is a term with the following syntax, where M is any term :

$$Ar ::= Set \mid Prop \mid (x : M)Ar$$

In the following, T is a generic term for the syntax of types and t for the syntax of terms.

Definition 3. Inductive Types :

$$T ::= Ind[p_1 : P_1| \dots |p_n : P_n](X : Ar)\{c_1 : C_1| \dots |c_m : C_m\}$$

Ar is an arity, p_i are the parameters of the inductive type, n the number of parameters and c_i the constructors of the inductive type.

Definition 4. Syntax for the constructors type :

$$Co ::= X \mid (Co\ m) \mid (P \to Co) \mid (x : M)Co$$

with X not appearing in m nor in M and P strictly positive w.r.t. X, where m and M are any terms.

Definition 5. A term is strictly positive with respect to X if it has the following syntax :

$$Pos ::= X \mid (Pos\ m) \mid (x : M)Pos \quad with\ X \notin FV(M, m)$$

Definition 6. Constructors :

$$t ::= \mathsf{Constr}(i, T)$$

Example 1.

- The $\exists$ connective can be defined by :

  $$sig := Ind[A : Set; P : A \to Prop](X : Set)\{exist : (x : A)(P\ x) \to X\}$$

- The $\vee$ connective can be defined by :

  $$sumor := Ind[A{:}Set; B{:}Prop](X{:}Set)\{inleft : A \to X|inright : B \to X\}$$

  The inductive type *sumor* is a simulation for exceptions. Indeed, this is a sum type. The first component has a computational sense (marked with *Set*) and the second a logical sense (marked with *Prop*). The first component contains the result if it exists, the second simulates raising an exception.

Notations :
We note *constructor(Co_i, X)* for Co_i *is a constructor type w.r.t.* X.
A vectorial notation is used for lists of variables. We note $\boldsymbol{a}$ for $a_1 \dots a_n$ and $(\boldsymbol{a} : \boldsymbol{A})t$ for $(a_1 : A_1) \dots (a_n : A_n)t$.

Then, a notion of elimination can be defined.

Definition 7. Elimination :

$$t ::= \mathsf{Elim}(t_0, T)\{t_1| \dots |t_n\}$$

where t_0 is the term on which the elimination is done, T the type of the elimination term and t_i the different branches of the elimination.

There are two kinds of eliminations : the dependent elimination ($Dep(s, s')$) and the non-dependent elimination ($Nodep(s, s')$). The non-dependent one is a particular case of the dependent one. It corresponds to the situation where the proposition to be proved does not depend on the term on which the elimination is done. We present here only the dependent elimination. For this, an auxiliary definition is needed.

Definition 8. Given s and s' two sorts, $A \equiv (\boldsymbol{x} : \boldsymbol{A})s$, X a variable of type A, Q a variable of type $(\boldsymbol{x} : \boldsymbol{A})(X\ \boldsymbol{x}) \rightarrow s'$, C a constructor type w.r.t. X and c a term of type C, we define a new type noted $C\{X, Q, c\}$ by induction on C :

$$(P \rightarrow C)\{X, Q, c\} = (p : P)P\{X, Q, p\} \rightarrow C\{X, Q, (c\ p)\} \quad X \text{ strict. pos. in } P$$
$$(x : M)C\{X, Q, c\} = (x : M)C\{X, Q, (c\ x)\} \quad X \text{ not in } M$$
$$(X\ \boldsymbol{a})\{X, Q, c\} = (Q\ \boldsymbol{a}\ c)$$

Then, three new rules are defined for typing an inductive type, a constructor and an elimination.

Definition 9. Inductive Types in the Calculus of Constructions :

$$\frac{\begin{array}{c}(\forall i = 1 \ldots m)\\ \Gamma[\boldsymbol{p} : \boldsymbol{P}] \vdash A : s \quad \Gamma[\boldsymbol{p} : \boldsymbol{P}; X : A] \vdash C_i : s' \quad \mathit{constructor}(C_i, X)\end{array}}{\Gamma \vdash \mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A)\{\boldsymbol{c} : \boldsymbol{C}\} : (\boldsymbol{p} : \boldsymbol{P})A}$$

$$\frac{\Gamma \vdash I : (\boldsymbol{p} : \boldsymbol{P})A \quad 1 \leq i \leq m}{\Gamma \vdash \mathsf{Constr}(i, I) : (\boldsymbol{p} : \boldsymbol{P})C_i[X \leftarrow (I\ \boldsymbol{p})]}$$

$$\frac{\begin{array}{c}\forall i = 1 \ldots m, \Gamma \vdash f_i : C_i\{(I\ \boldsymbol{p}), Q, (\mathsf{Constr}(i, I)\ \boldsymbol{p})\}\\ \Gamma \vdash c : (I\ \boldsymbol{p}\ \boldsymbol{a}) \quad \Gamma \vdash Q : (\boldsymbol{x} : \boldsymbol{A})(I\ \boldsymbol{p}\ \boldsymbol{x}) \rightarrow s'\end{array}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1 | \ldots | f_m\} : (Q\ \boldsymbol{a}\ c)}\ Dep(s, s')$$

The rules of elimination are parametered by two sorts s and s'. s is the sort of the arity of the inductive type and s' the sort of the type of the elimination predicate.

There is a particular notion of reduction on inductive types called the ι-reduction that we do not detail here. A definition of this reduction can be found in [PM93].

Example 2.

Let us give an example on lists. We show an elimination to illustrate the definition 8 and an example of ι-reduction. The inductive type $list$ has one parameter corresponding to the type of the elements. If $A : Set$ is this parameter, then an elimination predicate Q on lists is of type $(list\ A) \rightarrow Set$. The elimination has two cases f_1 and f_2 with the following types :

$$f_1 : (Q\ (nil\ A))$$
$$f_2 : (x : A)(l : (list\ A))(Q\ l) \rightarrow (Q\ (cons\ A\ x\ l))$$

The ι-reduction corresponds to the following reduction for the two cases *nil* and *cons* :

$$\mathsf{Elim}(nil\ A, Q)\{f_1|f_2\} \succ_\iota f_1$$
$$\mathsf{Elim}(cons\ A\ x\ l, Q)\{f_1|f_2\} \succ_\iota (f_2\ x\ l\ \mathsf{Elim}(l, Q)\{f_1|f_2\})$$

***Coq* notations :**

We present here the *Coq* syntax for some particular inductive types and for eliminations. This syntax will be used in the following sections.
`A+{B}` is the syntax for $(sumor\ A\ B)$.
`{x:A|P(x)}` is the syntax for $(sig\ A\ [x:A](P\ x))$.
`<P>Match c with f1...fn end` is the syntax for $\mathsf{Elim}(c, P)\{f_1|\ldots|f_n\}$.

2.3 The strong extraction function

A proof is a very inefficient program. For the division example, a program does not need to keep the proof of $a = b*q+r$. The extraction of [PM89b] consists in forgetting all the logical parts of the proof. These parts have been marked by the programmer initially with *Prop* for logical parts and *Set* for computational (or informative) parts. Moreover, dependent types are suppressed. Then, resulting terms are typable in the system ${F_\omega}^{Ind}$ (F_ω [Gir72, PM89b] plus inductive types). ${F_\omega}^{Ind}$ corresponds to the Calculus of Inductive Constructions with only $Set : TypeSet$ and without the rule $(Set, TypeSet)$. This corresponds to a calculus without the possibility for types to depend on terms.

An auxiliary definition of level on terms is needed. Indeed, terms and types are nested in the Calculus of Constructions. Levels can be introduced and are given by the typing. Indeed, if $\Gamma \vdash M : N$ then either $N = Type(Set)$ or $\exists s$ s.t. $N : s$. Three levels can then be defined.

Definition 10. Levels of terms :

- *Type* and *TypeSet* have level -1.
- A term is of level 0 if it is an arity. This is the domain of propositional types.
- A term is of level 1 if its type is of level 0. This is the domain of propositional schemes or predicates.
- A term is of level 2 if its type is of level 1. This is the domain of proofs.

Remark.

- Each term has a unique level defined by its type.
- $(B\ A)$, $(x : A)B$ and $[x : A]B$ have the same level as B.
- In $(x : A)B$ and $[x : A]B$, A has necessarily level 0 or 1.
- In $(B\ A)$, A has level 1 or 2.

Example 3.

- Level 0 : if A is a type, $A \to Set$ has level 0.
- Level 1 : inductive types have level 1.

– Level 2 : constructors of inductive types and weak eliminations have level 2. The arithmetical variable $+ : nat \rightarrow nat \rightarrow nat$ has level 2 if one supposes that the type nat is an inductive type with two constructors 0 and S. Then, the term $[x : nat](+\ x\ (S\ 0))$ has level 2.

The property for a term to be informative (computational) or logical can now be formally defined.

Definition 11. Informative arity :
An informative arity is a term with the following syntax :

$$InfAr ::= Set \mid (x : M)InfAr$$

Definition 12. Informative term, logical term :

– A term of level 0 is informative if it is an informative arity.
– A term of level 1 is informative if its type is an informative arity.
– A term of level 2 is informative if the type of its type is Set.

A term is logical if it is not informative.

The extraction function of [PM89b] can now be presented. In the following, we refer to this function as the strong extraction function. Intuitively, this function suppresses all the logical informations from a proof term. Moreover, it suppresses all the dependences of types with terms. This function can be applied only on informative terms.

Notation : The application of a constructor to its parameters is treated as a particular case. Indeed, in an application, all the informative arguments are kept (whatever their level is). For the parameters of a constructor, only the informative ones of level 1 have to be kept. Parameters of constructors are then not treated as usual arguments of an application. A constructor applied to its parameters is noted $\mathsf{Constr}(i, ind, \boldsymbol{p})$ where i is the number of the constructor, ind the corresponding inductive type and $\boldsymbol{p}$ the vector of parameters.

Definition 13. Strong extraction on terms :

$$\varepsilon_0(Set) = Set$$

$$\varepsilon_0((x : A)B) = \begin{cases} \varepsilon_0(A) \rightarrow \varepsilon_0(B) & \text{if A informative of level 0} \\ \varepsilon_0(B) & \text{otherwise} \end{cases}$$

$$\varepsilon_1(x) = \bar{x} \ \text{ where } \bar{x} : \varepsilon_0(T_x) \text{ if } x : T_x$$

$$\varepsilon_1((x : A)B) = \begin{cases} (\bar{x} : \varepsilon_i(A))\varepsilon_1(B) & \text{if A informative of level } i \leq 1 \\ \varepsilon_1(B) & \text{otherwise} \end{cases}$$

$$\varepsilon_1([x : A]B) = \begin{cases} [\bar{x} : \varepsilon_0(A)]\varepsilon_1(B) & \text{if A informative of level 0} \\ \varepsilon_1(B) & \text{otherwise} \end{cases}$$

$$\varepsilon_1(A\ B) = \begin{cases} (\varepsilon_1(A)\ \varepsilon_1(B)) & \text{if B informative of level 1} \\ \varepsilon_1(A) & \text{otherwise} \end{cases}$$

$$\varepsilon_1(\mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A)\{\boldsymbol{c} : \boldsymbol{C}\}) =$$

$$\mathsf{Ind}[\overline{\boldsymbol{p}} : \varepsilon_0(\boldsymbol{P})](\bar{X} : \varepsilon_1(A))\{\overline{\boldsymbol{c}} : \varepsilon_1(\boldsymbol{C})\} \ \text{ for informative } P_i \text{ of level } 0$$

$$\varepsilon_2(x) = \bar{x} \ \text{ where } \bar{x} : \varepsilon_1(T_x) \text{ if } x : T_x$$

$$\varepsilon_2([x : A]B) = \begin{cases} [\bar{x} : \varepsilon_i(A)]\varepsilon_2(B) & \text{if A informative of level } i \leq 1 \\ \varepsilon_2(B) & \text{otherwise} \end{cases}$$

$$\varepsilon_2(A\ B) = \begin{cases} (\varepsilon_2(A)\ \varepsilon_i(B)) & \text{if B informative of level } i \leq 2 \\ \varepsilon_2(A) & \text{otherwise} \end{cases}$$

$$\varepsilon_2(\mathsf{Elim}(c, P)\{f_1|\ldots|f_n\}) = \mathsf{Elim}(\varepsilon_2(c), \varepsilon_1(P))\{\varepsilon_2(f_1)|\ldots|\varepsilon_2(f_n)\}$$

$$\varepsilon_2(\mathsf{Constr}(i, ind, \boldsymbol{p})) = \mathsf{Constr}(i, \varepsilon_1(ind), \varepsilon_1(\boldsymbol{p}))$$

Remark. The extraction function corresponding to a term of level i is noted ε_i, since each term has a unique level.

Definition 14. Strong extraction on contexts :
$\varepsilon_i(\Gamma)$ is the extracted context of Γ defined by induction on Γ :

- $\varepsilon_i([]) = []$
- $\varepsilon_i([\Gamma; x : A]) = \varepsilon_i(\Gamma)$ if A logical or if level of $A > i$.
- $\varepsilon_i([\Gamma; x : A]) = [\varepsilon_i(\Gamma); \bar{x} : \varepsilon_j(A)]$ otherwise.

Proposition 15. *If $\Gamma \vdash t : P$ then $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}(P)$.*

Proof. By induction on the length of the derivation. □

Example 4.
Let us consider the case of the predecessor function which can be specified through the proposition : $(n : nat)\{m : nat|(S\ m) = n\} + \{0 = n\}$. The proof of this proposition proceeds by induction on n. If $n = 0$ then the right part of the specification is true. Otherwise, if $n = y{+}1$, then the left part of the specification is true and y is the witness. A proof term for this specification is then :

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
           (inright {m:nat|(S m)=0} (0=0) (refl_equal nat 0))
           [y:nat][H:{m:nat|(S m)=y}+{0=y}]
             (inleft {m:nat|(S m)=(S y)} (0=(S y))
                 (exist nat [m:nat](S m)=(S y) y
                        (refl_equal nat (S y))))
       end
```

The extraction of this term is, if over-lines are forgotten :

```
[n:nat]<(sumor (sig nat))>Match n with
          (inright (sig nat))
          [y:nat][H:(sumor (sig nat))]
            (inleft (sig nat) (exist nat y))
       end
```

Since *sumor* is a simulation for exceptions, the extraction keeps only the argument of the first component (the witness y) and forgets the argument of the second component.

In an ML-like language, this term could be written :

```
let pred n = match n with
               O -> raise except_pred
             | y+1 -> y ;;
```

2.4 Non-inversibility of the strong extraction

The extraction function allows to synthesize programs from proofs. However, a complementary idea could be to start from a program and its specification and try to retrieve the corresponding proof. A natural idea for this is to invert the extraction function. Unfortunately, the function of [PM89b] cannot be inverted. Indeed, not all the intelligence to prove properties is reflected by the program.

Example 5.

For a specification $\{x : A|(P\ x)\}$ and a program a, we need to reconstruct a proof of $(P\ a)$ for an arbitrary P. This is impossible. Indeed, this is an undecidable property. It is impossible to retrieve a proof only from its type.

There is another problem : intermediate specifications disappear. Let us take the case of a specification $(n : nat)\{p : nat|2 * p = n\} \vee \{p : nat|2 * p + 1 = n\}$. In a proof, this specification can be enforced by a distinction between n even and n odd. The intermediate specification is then $(n : nat)((even\ n) \wedge \{p : nat|2 * p = n\}) \vee ((odd\ n) \wedge \{p : nat|2*p+1 = n\})$. It is impossible to retrieve from a program such a specification because logical informations are not in the program.

Then, it seems natural not to be able to retrieve logical proofs. It corresponds to logical properties the programmer will have to prove on the program. However, one can hope to be able to retrieve intermediate specifications. That is why we introduce a new extraction function.

3 The weak extraction function

The aim of this new extraction function is to find a way of keeping a sufficient number of informations to be able to retrieve a proof term from a weak extracted term. Since we show the problem is to retrieve intermediate specifications, this new function keeps the specifications and no more suppresses them. Then, this function only suppresses logical proofs.

3.1 Definitions and properties

In this case, arguments of constructors are treated as usual arguments. Indeed, we now want to keep all the informative parameters of a constructor (whatever their level is). Then, we go back now to the notation $\mathsf{Constr}(i, ind)$.

Definition 16. Weak extraction on terms :

$$\sigma_i([x:A]B) = \begin{cases} \sigma_i(B) & \text{if } A : Prop \\ [\bar{x} : \sigma_j(A)]\sigma_i(B) & \text{otherwise} \end{cases}$$

$$\sigma_i(A\ B) = \begin{cases} \sigma_i(A) & \text{if } B \text{ logical of level 2} \\ (\sigma_i(A)\ \sigma_j(B)) & \text{otherwise} \end{cases}$$

$$\sigma_i(\mathsf{Elim}(c,P)\{f_1|\ldots|f_n\}) = \mathsf{Elim}(\sigma_2(c), \sigma_{i-1}(P))\{\sigma_i(f_1)|\ldots|\sigma_i(f_n)\}$$

$$\sigma_i(x) = \bar{x} \text{ where } \bar{x} : \sigma_{i-1}(T_x) \text{ if } x : T_x$$

$$\sigma_2(\mathsf{Constr}(i, ind)) = \mathsf{Constr}(i, \sigma_1(ind))$$

$$\sigma_1((x:A)B) = (\bar{x} : \sigma_i(A))\sigma_1(B)$$

$$\sigma_1(\mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A)\{\boldsymbol{c} : \boldsymbol{C}\}) =$$

$$\mathsf{Ind}[\overline{\boldsymbol{p}} : \sigma_j(\boldsymbol{P})](\bar{X} : \sigma_0(A))\{\overline{\boldsymbol{c}} : \sigma_1(\boldsymbol{C})\} \text{ for } P_i \text{ not of type } Prop$$

$$\sigma_i(K) = K \text{ where } K \text{ is a sort}$$

$$\sigma_0((x:A)B) = \begin{cases} \sigma_0(B) & \text{if } A : Prop \\ (\bar{x} : \sigma_0(A))\sigma_0(B) & \text{otherwise} \end{cases}$$

All the dependences w.r.t. logical proofs are suppressed as in definition 13 but only these ones (in definition 13 for strong extraction, informative proofs and logical specifications are suppressed too). To keep a coherence, logical types of level 1 are suppressed in λ-expression (but not informative types of level 1 as in definition 13). Note the rule on product of level 1. Here, nothing is suppressed. This is the case of a specification. We said before we do not want to suppress specifications. So, we keep them all. Let us look at the example given to show the non-inversibility of the strong extraction. We wanted to keep the informations on *even* and *odd*. This is not possible with the strong extraction in definition 13 but possible with the weak extraction with the rule on product of level 1.

This is the main difference between the two definitions (13 and 16).

Definition 17. Weak extraction on contexts :

$\sigma_c(\Gamma)$ is the weak extracted context of Γ defined by induction on Γ :

- $\sigma_c([]) = []$
- $\sigma_c([\Gamma; x : A]) = \sigma_c(\Gamma)$ if $A : Prop$.
- $\sigma_c([\Gamma; x : A]) = [\sigma_c(\Gamma); \bar{x} : \sigma_i(A)]$ otherwise.

Example 6.

Let us show the weak extraction on the previous example of the predecessor. Note that the weak extraction does not change the inductive types *sig* and *sumor*. The weak extracted term is then the following :

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
          (inright {m:nat|(S m)=0} (0=0))
          [y:nat][H:{m:nat|(S m)=y}+{0=y}]
            (inleft {m:nat|(S m)=(S y)} (0=(S y))
                    (exist nat [m:nat](S m)=(S y) y))
       end
```

Note that this program has the same computational contents than the strong extracted program. The only difference is that there are annotations corresponding to intermediate specifications.

The terms obtained by application of this function on terms of the Calculus of Inductive Constructions are ${F_\omega}^{Ind}$ programs annotated with specifications in the Calculus of Inductive Constructions. Such terms contain logical informations which allow the inversion of the weak extraction (the informations on *even* and *odd* in our previous example for non-inversibility). We need a new notion of typing for these particular terms since they are neither typable in the Calculus of Inductive Constructions nor in ${F_\omega}^{Ind}$.

Definition 18. Typing on weak extracted terms :

$$\frac{}{[]\ well\ formed}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x:T]\ well\ formed}\quad s \in \mathcal{S} = \{Prop, Set, Type, TypeSet\}$$

$$\frac{\Gamma\ well\ formed}{\Gamma \vdash Prop \in Type}$$

$$\frac{\Gamma\ well\ formed}{\Gamma \vdash Set \in TypeSet}$$

$$\frac{\Gamma\ well\ formed \quad x:T \in \Gamma}{\Gamma \vdash x \in T}$$

$$\frac{\Gamma \vdash T \in s \quad \Gamma[x:T] \vdash T' \in s'}{\Gamma \vdash (x:T)T' \in s'}\quad (s,s') \in \mathcal{R} = \mathcal{S}\times\mathcal{S} \setminus \left\{ \begin{matrix}(Prop, Type)\\ (Prop, TypeSet)\end{matrix} \right\} \tag{1}$$

$$\frac{\Gamma \vdash (x:T)T' \in s \quad \Gamma[x:T] \vdash t \in T'}{\Gamma \vdash [x:T]t \in (x:T)T'}\quad s \in \mathcal{S}$$

$$\frac{\Gamma \vdash t \in (x:A)B \quad \Gamma \vdash t' \in A}{\Gamma \vdash (t\ t') \in B[x \leftarrow t']}\quad \text{A of level 0 or informative of level 1}$$

$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \to U}{\Gamma \vdash t \in U} \tag{2}$$

$$\frac{\Gamma \vdash t \in U \quad \Gamma \vdash A \in Prop}{\Gamma \vdash t \in A \to U} \tag{3}$$

$$\frac{\Gamma \vdash T' \in s \quad \Gamma \vdash t \in T \quad T =_{\beta\iota} T'}{\Gamma \vdash t \in T'}$$

$$\frac{\begin{matrix}(\forall i = 1 \dots m)\\ \Gamma[\boldsymbol{p}:\boldsymbol{P}] \vdash A \in s \quad \Gamma[\boldsymbol{p}:\boldsymbol{P}; X:A] \vdash C_i \in s' \quad constructor(C_i, X)\end{matrix}}{\Gamma \vdash \mathsf{Ind}[\boldsymbol{p}:\boldsymbol{P}](X:A)\{\boldsymbol{c}:\boldsymbol{C}\} \in (\boldsymbol{p}:\boldsymbol{P})A}\quad \nexists P_j : Prop$$

$$\frac{\Gamma \vdash Ind \equiv \mathsf{Ind}[\boldsymbol{p}:\boldsymbol{P}](X:A)\{\boldsymbol{c}:\boldsymbol{C}\} \in B \quad 1 \le i \le m}{\Gamma \vdash \mathsf{Constr}(i, Ind) \in (\boldsymbol{p}:\boldsymbol{P})C_i[X \leftarrow (Ind\ \boldsymbol{p})]}$$

$$\frac{\begin{matrix}\forall i = 1 \dots m, \Gamma \vdash f_i \in C_i\{(I\ \boldsymbol{p}), Q, (\mathsf{Constr}(i, I)\ \boldsymbol{p})\}_\sigma\\ \Gamma \vdash c \in (I\ \boldsymbol{p}\ \boldsymbol{a}) \quad \Gamma \vdash Q \in (\boldsymbol{x}:\boldsymbol{A})(I\ \boldsymbol{p}\ \boldsymbol{x}) \to s'\end{matrix}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1| \dots |f_m\} \in (Q\ \boldsymbol{a}\ c)}$$

The definition of $C\{X,Q,c\}$ is modified into $C\{X,Q,c\}_\sigma$. Indeed, no proof variables have to appear in constructors. If a constructor is $c:(n:nat)(n>0)\rightarrow C$, the elimination principle should be $(n:nat)(n>0)\rightarrow(P\ (c\ n))$ and not $(n:nat)(h:n>0)\rightarrow(P\ (c\ n\ h))$. The new definition of $C\{X,Q,c\}_\sigma$ allows to avoid this problem.

Definition 19. Given s and s' two sorts, $A\equiv(\boldsymbol{x}:\boldsymbol{A})s$, X a variable of type A, Q a variable of type $(\boldsymbol{x}:\boldsymbol{A})(X\ \boldsymbol{x})\rightarrow s'$, C a constructor type w.r.t. X and c a term of type C, we define $C\{X,Q,c\}_\sigma$ by induction on C :

$$(P\rightarrow C)\{X,Q,c\}_\sigma=(p:P)P\{X,Q,p\}_\sigma\rightarrow C\{X,Q,(c\ p)\}_\sigma$$

$$(x:M)C\{X,Q,c\}_\sigma=\begin{cases}C\{X,Q,c\}_\sigma & \text{if } M:Prop\\ (x:M)C\{X,Q,(c\ x)\}_\sigma & \text{otherwise}\end{cases}$$

$$(X\ \boldsymbol{a})\{X,Q,c\}_\sigma=(Q\ \boldsymbol{a}\ c)$$

The above rules for typing are close to the ones of the Calculus of Inductive Constructions (see definition 1) but not exactly the same. The rules who are differing are the rules 1, 2, 3 and the application formation. The motivations for the rule 1 is given in the following remarks. The application formation is allowed only with arguments having a type of level 0 or informative of level 1 : this is due to the fact that we suppress only logical proofs in application. Finally, logical proofs are suppressed in applications but the corresponding logical specifications are not suppressed in types (due to the rule on the product of level 1). The rules 2 and 3 are direct consequences of this fact. Rules are needed to play with logical products in types that have no corresponding argument in an application. The rules 2 and 3 deal with this and allow to add or suppress any logical products in a type.

Remark.

1. In the first case of the precedent definition, P cannot have the type $Prop$. Indeed, an inductive type in an elimination is always informative. Moreover, P is strictly positive w.r.t. X. Then, $P:Set$.
2. In the rule 1, $(Prop,Type)$ and $(Prop,TypeSet)$ are not necessary because they allow the formation of dependent types on logical proofs and these have been suppressed.
3. In the following, σ applies to all terms except logical proofs. ε applies only on informative terms.

Definition 20. If $\Gamma\vdash t\in S$ then the program t is said to be coherent with the specification S.

We can now give a list of standard properties on this weak extraction and the corresponding new typing judgment.

Proposition 21. *Coherence of the new typing judgment w.r.t. the strong extraction :*

If $\Gamma\vdash t\in P$ and P is informative then $\varepsilon_i(\Gamma)\vdash\varepsilon_i(t):\varepsilon_{i-1}(P)$.

Proof. By induction on the length of the derivation of $\Gamma \vdash t \in P$. □

Proposition 22. *First substitutivity lemma :*
For all term t of level 0, 1 or informative of level 2 :

$$\sigma_i(T[x \leftarrow t]) = \sigma_i(T)[x \leftarrow \sigma_j(t)]$$

where i is the level of T and j the level of t and x. Moreover, informations of x and t have to be coherent (if one is logical, the other one too).

Proof. By induction on the structure of t. □

Proposition 23. *Second substitutivity lemma :*
For all logical term t of level 2 :

$$\sigma_i(T[x \leftarrow t]) = \sigma_i(T)$$

where i is the level of T with levels and informations of x and t coherent.

Proof. By induction on the structure of t. □

Proposition 24. *Coherence of the weak extraction :*
$\forall \Gamma$, $\forall t$, $\forall P$, such that P is not of type Prop, $\Gamma \vdash t : P \Rightarrow \sigma(\Gamma) \vdash \sigma(t) \in \sigma(P)$.

Proof. By induction on the length of the derivation of $\Gamma \vdash t : P$. □

3.2 Inversion of the weak extraction

We now want to invert the weak extraction. Given a weak extracted program and a specification, we want to retrieve a proof term. We show that we can reconstruct a partial proof term with "holes" corresponding to logical proofs to be proved by the user. Then, all the logical proofs of one specification have to be identified. The proof of inversibility gives such an algorithm. Some restrictions on the model have to be stated in order to be able to invert the function.

1. We want to forbid some situations in which we do not know how to retrieve a proof term from a weak extracted term.Typically, let us take a term T whose type is $(x : nat)(P : (y : nat)(x > y) \rightarrow Set)(P\ 1\ h_1) \rightarrow (P\ 2\ h_2)$. Its weak extraction is $\overline{T} : (x : nat)(P : nat \rightarrow Set)(P\ 1) \rightarrow (P\ 2)$. The link between h_1 and h_2 is lost. Moreover, let us take a term $root$ of type $(x : nat)(x > 0) \rightarrow nat$. Its weak extraction is $\overline{root} : nat \rightarrow nat$. Suppose we consider a weak extracted proposition $(root\ x)$, then a proof term $(root\ x\ h_1)$ where h_1 is an existential variable of type $x > 0$ can be reconstructed.
   In fact, we want to be able to define types that depend on logical proofs (as for $root$) but no arities that depend on logical proofs (as the type of T). This implies we consider a PTS without the rules $(Prop, Type)$ and $(Prop, TypeSet)$.

Remark. In a PTS without $(Prop, Type)$ and $(Prop, TypeSet)$, the following points are always valid :

- $\sigma_i((x : A)B) = (x : \sigma_j(A))\sigma_i(B)$ whatever the level of B is (since this is always true for level 1 and the restrictions of the PTS implies it is true for level 0).
- if $\sigma_0(A) = s$ where s is a sort then $A = s$ (for the same reason).
- if $\sigma_i(A) = (x : B)C$ then $A = (x : B')C'$.
- if $\sigma_i(A) = [x : B]C$ and if A has level 1 then $A = [x : B']C'$.

2. Now, if we know how to generate a proof obligation h_1 for $x > 0$ in the example of *root*, there exist many possibilities for h_1. We want to identify all the proofs of a same logical proposition. We need a *proof-irrelevance* of logical proofs. For this, the conversion rule between two terms as to be extended to be a rule between typed terms.

   Definition 25. Extended equality between typed terms (noted $=_e$) :
   If $B =_{\beta\iota} B'$ and if l and l' are logical terms of level 2 then $B[x \leftarrow l] =_e B'[x \leftarrow l']$.

   We can give a method to explicitly compare two typed terms without any problems (by comparing their normal form recursively).

Proposition 26. *Proof-irrelevance of logical proofs :*
If A and A' are two typed terms, equal modulo $=_e$, verifying the previous conditions 1 and 2 and that are not logical proofs, then $\sigma_i(A) = \sigma_i(A') \Rightarrow A = A'$.

Proof. We consider the terms in their η-long normal form and compare them. By induction on their shape, there is no problem. □

We can now state the theorem of inversibility. It corresponds to the following intuition. A program p is invertible if, given a context Γ and a specification S where S is provable and p coherent with S in Γ, a proof term for p can be reconstructed with holes corresponding to logical proofs. We place us in a model verifying the two precedent restrictions and all judgments are restricted to this model.

Notation : we note $\vdash_r$ for restricted judgments.

Theorem 27. *Inversibility of the weak extraction :*
If $\Delta \vdash t \in T$ and if Γ is a well formed environment such that $\sigma_c(\Gamma) = \Delta$ then $\exists t', \mathcal{L}$ well formed logical context and S' such that $\Gamma, \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$, $\sigma(S') = T$.

Proof. By induction on the derivation of $\in$. We do not detail the cases of context formation. Each step gives the construction of the proof term t. At each step, Γ is known such that $\sigma_c(\Gamma) = \Delta$. In all the proof, the L are logical terms of level 2, and $\mathcal{L}$ is a logical context.

– Variable :

$$\frac{\Delta\ well\ formed \quad x : T \in \Delta}{\Delta \vdash x \in T}$$

The proof term is the variable x of type Γ_x. As $\sigma_c(\Gamma) = \Delta$, $\sigma(\Gamma_x) = \Delta_x = T$. Then, $\Gamma \vdash x : \Gamma_x$ with $\sigma(x) = x$ and $\sigma(\Gamma_x) = T$.
The searched environment is Γ, the proof term x and its type Γ_x.

– Application formation :

$$\frac{\Delta \vdash t \in (x : A)B \quad \Delta \vdash u \in A}{\Delta \vdash (t\ u) \in B[x \leftarrow u]} \ \text{A of level 0 or informative of level 1}$$

By induction hypothesis, $\exists t', \mathcal{L}$ and T' such that $\Gamma, \mathcal{L} \vdash_r t' : T'$ with $\sigma(t') = t$ and $\sigma(T') = (x : A)B$. Moreover, $\exists u', \mathcal{L}'$ and A' such that $\Gamma, \mathcal{L}' \vdash_r u' : A'$ with $\sigma(u') = u$ and $\sigma(A') = A$.
Necessarily, $T' = (x : A'')B'$ with $\sigma(A'') = A$ and $\sigma(B') = B$. There are no logical products at the head of T'. Indeed, the condition 1 imposes arities not depending on logical proofs. Then, $\Gamma, \mathcal{L} \vdash_r t' : (x : A'')B'$ and $\sigma(A') = \sigma(A'') = A$. To apply the standard typing rule for application, $A' = A''$ is needed. The irrelevance of logical proofs is then necessary (see proposition 26. $A' = A''$ can be deduced from $\sigma(A') = \sigma(A'')$. Then, $\exists \mathcal{L}''$, union of $\mathcal{L}$ and $\mathcal{L}'$, such that $\Gamma, \mathcal{L}'' \vdash_r (t'\ u') : B'[x \leftarrow u']$. $(t'\ u')$ is the proof term with the following properties :

$$\begin{aligned} \sigma(t'\ u') &= (\sigma(t')\ \sigma(u')) \\ &= (t\ u) \end{aligned}$$

$$\begin{aligned} \sigma(B'[x \leftarrow u']) &= \sigma(B')[x \leftarrow \sigma(u')] \\ &= B[x \leftarrow u] \end{aligned}$$

– Product formation :

$$\frac{\Delta \vdash T_1 \in s_1 \quad \Delta[x : T_1] \vdash T_2 \in s_2}{\Delta \vdash (x : T_1)T_2 \in s_2} \ (s_1, s_2) \in \mathcal{R} = \mathcal{S} \times \mathcal{S} \setminus \left\{ \begin{matrix} (Prop, Type) \\ (Prop, TypeSet) \end{matrix} \right\}$$

By induction hypothesis, $\exists t', \mathcal{L}$ and S' such that $\Gamma[x : A], \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = T'$, $\sigma(S') = s_2$ and $\sigma(A) = T$. As $s_2 \in \mathcal{S}$, $S' = s_2$. Indeed, the condition 1 imposes arities not depending on proofs, then S' is necessarily a sort. Then, $\Gamma[x : A], \mathcal{L} \vdash_r t' : s_2$.
To give a proof term, the logical variables of $\mathcal{L}$ have to be commuted in the context. A vector $\boldsymbol{l'} : \boldsymbol{L'}$ is defined with $L'_i = (x : A)L_i$ if $\mathcal{L} = [\boldsymbol{l} : \boldsymbol{L}]$. In t', l_i are substituted by $(l'_i\ x)$ and a term t'' is obtained. Then, $\Gamma[\boldsymbol{l'} : \boldsymbol{L'}; x : A] \vdash_r t'' : s_2$.
The application of the standard typing rule for the product gives : $\Gamma[\boldsymbol{l'} : \boldsymbol{L'}] \vdash_r (x : A)t'' : s_2$. The conditions are :

$$\begin{aligned} \sigma((x : A)t'') &= (x : \sigma(A))\sigma(t'') \\ &= (x : T_1)T_2 \end{aligned}$$

$$\sigma(s_2) = s_2$$

– Abstraction formation :

$$\frac{\Delta \vdash (x:T_1)T_2 \in s \quad \Delta[x:T_1] \vdash t \in T_2}{\Delta \vdash [x:T_1]t \in (x:T_1)T_2} \quad s \in \mathcal{S}$$

By induction hypothesis, $\exists t', \mathcal{L}$ and S' such that $\Gamma[x:A], \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$, $\sigma(S') = T_2$ and $\sigma(A) = T$.
To give a proof term, the logical variables of $\boldsymbol{l}$ have to be commuted in the context. A vector $\boldsymbol{l'} : \boldsymbol{L'}$ is defined with $L'_i = (x:A)L_i$ if $\mathcal{L} = [\boldsymbol{l} : \boldsymbol{L}]$. We note $(\boldsymbol{l'}\ \boldsymbol{x})$ for the vector $((l'_1\ x), \ldots, (l'_n\ x))$. The, $\Gamma[\boldsymbol{l'} : \boldsymbol{L'}; x:A] \vdash_r t'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})] : S'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})]$.
The application of the standard typing rule for the abstraction gives : $\Gamma[\boldsymbol{l'} : \boldsymbol{L'}] \vdash_r [x:A]t'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})] : (x:A)S'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})]$. The conditions are :

$$\begin{aligned}\sigma([x:A]t'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})]) &= [x:\sigma(A)]\sigma(t') \\ &= [x:T_1]t\end{aligned}$$

$$\begin{aligned}\sigma((x:A)S'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})]) &= (x:\sigma(A))\sigma(S'[\boldsymbol{l} \leftarrow (\boldsymbol{l'}\ \boldsymbol{x})]) \\ &= (x:T_1)T_2\end{aligned}$$

– Rule 2 :

$$\frac{\Delta \vdash A \in Prop \quad \Delta \vdash t \in A \to U}{\Delta \vdash t \in U}$$

By induction hypothesis, $\exists t', \mathcal{L}$ and S' such that $\Gamma, \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$ and $\sigma(S') = A \to U$. Then, $S' = (x:A')B'$ with $\sigma(A') = A$ and $\sigma(B') = U$. Then, $\Gamma, \mathcal{L} \vdash_r t' : (x:A')B'$.
This is exactly the same case as application and the proof term can be reconstructed. $\exists \mathcal{L}', x'$ such that $\Gamma, \mathcal{L}' \vdash_r x' : A'$. Then, $\exists \mathcal{L}''$ union of $\mathcal{L}$ and $\mathcal{L}'$ such that $\Gamma, \mathcal{L}'' \vdash_r (t'\ x') : B'[x \leftarrow x']$ with $\sigma(t'\ x') = t$ and $\sigma(B'[x \leftarrow x']) = U$. B' can depend on x but the weak extraction suppresses this dependence. This is the only place where a logical lemma is introduced. In the other cases, the logical lemmas of the induction hypotheses have repercussions but no new one is introduced.

– Rule 3 :

$$\frac{\Delta \vdash t \in U \quad \Delta \vdash A \in Prop}{\Delta \vdash t \in A \to U}$$

By induction hypothesis, $\exists t', \mathcal{L}$ and S' such that $\Gamma, \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$ and $\sigma(S') = U$.
Then, $\mathcal{L} = [l_1 : L_1; \ldots; l_i : L_i; x : A'; l_{i+1} : L_{i+1}; \ldots; l_n : L_n]$ with $\sigma(A') = A$. Commutations are done in $\mathcal{L}$, to obtain $\mathcal{L}' = [l_1 : L_1; \ldots; l_i : L_i; l'_{i+1} : L'_{i+1}; \ldots; l'_n : L'_n; x : A']$ where $L'_i = (x:A')L_i$.
Then, $\Gamma', \mathcal{L}' \vdash_r t' : S'$.
Then, $\Gamma'[l_1 : L_1; \ldots; l_i : L_i; l'_{i+1} : L'_{i+1}; \ldots; l'_n : L'_n] \vdash_r [x:A']t' : A' \to S'$ with $\sigma([x:A']t') = \sigma(t') = t$ and $\sigma(A' \to S') = A \to U$.

– Inductive type formation :

$$\frac{\begin{array}{c}(\forall i = 1 \ldots m)\\ \Delta[\boldsymbol{p} : \boldsymbol{P}] \vdash A \in s_1 \;\; \Delta[\boldsymbol{p} : \boldsymbol{P}; X : A] \vdash C_i \in s_2 \;\; \mathit{constructor}(C_i, X)\end{array}}{\Delta \vdash \mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A)\{\boldsymbol{c} : \boldsymbol{C}\} \in (\boldsymbol{p} : \boldsymbol{P})A} \; \nexists P_j : Prop$$

By induction hypothesis, $\exists \mathcal{L}, t'_i$ and S'_i such that $\Gamma[\boldsymbol{p'} : \boldsymbol{P'}; X' : A'], \mathcal{L} \vdash_r t'_i : S'_i$ with $\sigma(t'_i) = C_i$, $\sigma(S'_i) = s_2$, $\sigma(P'_j) = P_j$ and $\sigma(A') = A$. Moreover, $\exists \mathcal{L}', t''$ and S' such that $\Gamma, \mathcal{L}' \vdash_r t'' : S'$ with $\sigma(t') = A$ and $\sigma(S') = s_1$.
Then, $\Gamma[\boldsymbol{p'} : \boldsymbol{P'}; X' : A'; \boldsymbol{l} : \boldsymbol{L}] \vdash_r t'_i : S'_i$ where $[\boldsymbol{l} : \boldsymbol{L}]$ is the union of $\mathcal{L}$ and $\mathcal{L}'$. As $\sigma(t'') = \sigma(A')$, thanks to the irrelevance of logical proofs, $t'' = A'$. Then, as for precedent cases, $\boldsymbol{l}$ have to be commuted and the standard typing rule for inductive types can be applied. A proof term is obtained and it verifies the good properties.

– Constructor formation :

$$\frac{\Delta \vdash Ind \equiv \mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A)\{\boldsymbol{c} : \boldsymbol{C}\} \in B \quad 1 \leq i \leq m}{\Delta \vdash \mathsf{Constr}(i, Ind) \in (\boldsymbol{p} : \boldsymbol{P})C_i[X \leftarrow (Ind\ \boldsymbol{p})]}$$

By induction hypothesis, $\exists ind', \mathcal{L}$ and S' such that $\Gamma, \mathcal{L} \vdash_r ind' : S'$ with $\sigma(ind') = Ind$ and $\sigma(S') = B$.
$B = (\boldsymbol{p} : \boldsymbol{P})A$, then $\sigma(S') = (\boldsymbol{p'} : \boldsymbol{P'})A'$ with $\sigma(p'_i) = p_i$ and $\sigma(A') = A$.
Then, $\Gamma, \mathcal{L} \vdash_r ind' : (\boldsymbol{p'} : \boldsymbol{P'})A'$. Then, $ind' = \mathsf{Ind}[\boldsymbol{p} : \boldsymbol{P}](X : A')\{\boldsymbol{c'} : \boldsymbol{C'}\}$.
Then, $\Gamma, \mathcal{L} \vdash_r \mathsf{Constr}(i, ind') : (\boldsymbol{p'} : \boldsymbol{P'})C'_i[X \leftarrow (ind'\ \boldsymbol{p'})]$ with $\sigma(\mathsf{Constr}(i, ind')) = \mathsf{Constr}(i, Ind)$ and $\sigma((\boldsymbol{p'} : \boldsymbol{P'})C'_i[X \leftarrow (ind'\ \boldsymbol{p'})]) = (\boldsymbol{p} : \boldsymbol{P})C_i[X \leftarrow (Ind\ \boldsymbol{p})]$.

– Elimination formation :

$$\frac{\begin{array}{c}\forall i = 1 \ldots m, \Gamma \vdash f_i \in C_i\{(I\ \boldsymbol{p}), Q, (\mathsf{Constr}(i, I)\ \boldsymbol{p})\}_\sigma\\ \Gamma \vdash c \in (I\ \boldsymbol{p}\ \boldsymbol{a}) \quad \Gamma \vdash Q \in (\boldsymbol{x} : \boldsymbol{A})(I\ \boldsymbol{p}\ \boldsymbol{x}) \to s'\end{array}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1 | \ldots | f_m\} \in (Q\ \boldsymbol{a}\ c)}$$

By induction hypothesis, $\exists c_1, \mathcal{L}_1$ and S_c such that $\Gamma, \mathcal{L}_1 \vdash_r c_1 : S_c$ with $\sigma(c_1) = c$ and $\sigma(S_c) = (I\ \boldsymbol{p}\ \boldsymbol{a})$; $\exists Q_1, \mathcal{L}_2$ and S_Q such that $\Gamma, \mathcal{L}_2 \vdash_r Q_1 : S_Q$ with $\sigma(Q_1) = Q$ and $\sigma(S_Q) = (\boldsymbol{x} : \boldsymbol{A})(I\ \boldsymbol{p}\ \boldsymbol{x}) \to s'$; $\exists f'_i, \mathcal{L}^i$ and S_i such that $\Gamma, \mathcal{L}^i \vdash_r f'_i : S_i$ with $\sigma(f'_i) = f_i$ and $\sigma(S_i) = C_i\{(I\ \boldsymbol{p}), Q, (\mathsf{Constr}(i, I)\ \boldsymbol{p})\}_\sigma$.
Then, $S_Q = (\boldsymbol{x_1} : \boldsymbol{A'})(I_1\ \boldsymbol{p_1}\ \boldsymbol{x_1}) \to s'$ with $\sigma(A'_i) = A_i$, $\sigma(I_1) = I$, $\sigma(\boldsymbol{p_1}) = \boldsymbol{p}$ and $\sigma(\boldsymbol{x_1}) = \boldsymbol{x}$; $S_c = (I_2\ \boldsymbol{p_2}\ \boldsymbol{a_1})$ with $\sigma(I_2) = I$, $\sigma(\boldsymbol{p_2}) = \boldsymbol{p}$ and $\sigma(\boldsymbol{a_1}) = \boldsymbol{a}$; $S_i = C'_i\{(I_3\ \boldsymbol{p_3}), Q_2, (\mathsf{Constr}(i, I_4)\ \boldsymbol{p_4})\}_\sigma$ with $\sigma(C'_i) = C_i$, $\sigma(I_3) = I$, $\sigma(\boldsymbol{p_3}) = \boldsymbol{p}$, $\sigma(Q_2) = Q$, $\sigma(I_4) = I$ and $\sigma(\boldsymbol{p_4}) = \boldsymbol{p}$.
Then, thanks to the proof irrelevance, $I_1 = I_2 = I_3 = I_4$, $\boldsymbol{p_1} = \boldsymbol{p_2} = \boldsymbol{p_3} = \boldsymbol{p_4}$ and $Q_1 = Q_2$.
Then, $\exists \mathcal{L}_3$, union of $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}^i$, such that $\Gamma, \mathcal{L}_3 \vdash_r \mathsf{Elim}(c_1, Q_1)\{f'_i\} : (Q_1\ \boldsymbol{a_1}\ c_1)$ with the good properties.

$\square$

This proof is constructive. It explicitly gives a method to construct a proof term from a weak extracted program and a specification. Note that the proof term is partial. Indeed, in practice, logical proofs cannot be retrieved (in the proof of inversibility, we identify all of them). However, this proof is based on an analysis of the typing of the program. This typing is not deterministic due to the two rules 2 and 3. We need to replace these two rules with a new one (4) to have a new equivalent typing system. This system will be deterministic. In fact, we say it is deterministic in the sense that the application of the rules 2 and 3 are postponed and used via an equivalence relation only when typing an application. The rules 2 and 3 can then no more be used whenever we want but only when absolutely necessary.

Definition 28. Logical equivalence :
Two types A and B are logically equivalent ($A \equiv_L B$) if and only if there exist $\boldsymbol{L}$ and $\boldsymbol{M}$ logical of level 1, and C such that $A = \boldsymbol{L} \rightarrow C$ and $B = \boldsymbol{M} \rightarrow C$.

Remark.

1. This property is decidable.
2. If $A \equiv_L B$ and $\Gamma \vdash M \in A$, then $\Gamma \vdash M \in B$. Trivial by a suit of applications of the rules 2 and 3.
3. $\equiv_L$ is reflexive, symmetric and transitive.
4. $L \rightarrow A \equiv_L A$ if L is logical of level 1.

We can now define a new typing system called $\in_2$. It is the same as $\in$ but 2 and 3 are replaced by the following 4.

Definition 29. Typing system $\in_2$:

$$\frac{\Gamma \vdash t \in_2 A \quad \Gamma \vdash u \in_2 B \quad A \equiv_L (x:M)N \quad B \equiv_L M}{\Gamma \vdash (t\ u) \in_2 N[x \leftarrow u]} \tag{4}$$

We can now prove the equivalence between $\in$ and $\in_2$. The proof deals with the fact that 4 can be deduced from 2 and 3 and conversely.

Proposition 30. *Equivalence of $\in$ and $\in_2$:*
$\Gamma \vdash t \in S$ *if and only if* $\exists S'$ *such that* $\Gamma \vdash t \in_2 S'$ *with* $S \equiv_L S'$.

Proof.

– Necessary condition : $\in \Rightarrow \in_2$.
By induction on the typing $\in$. We consider only th rules 2 and 3.
 • Rule 2 :
$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \rightarrow U}{\Gamma \vdash t \in U}$$
By induction hypothesis, $\exists T$ such that $\Gamma \vdash t \in_2 T$ and $T \equiv_L A \rightarrow U$. Remark 3 on $\equiv_L$ gives $A \rightarrow U \equiv_L U$. Transitivity of $\equiv_L$ gives $T \equiv_L U$. Then, we have what we are looking for.

- Rule 3 : exactly the same principle.

– Sufficient condition : $\in_2 \Rightarrow \in$.
  By induction on the typing $\in_2$. The problematic rule is the rule 4.

$$\frac{\Gamma \vdash t \in_2 A \quad \Gamma \vdash u \in_2 B \quad A \equiv_L (x:M)N \quad B \equiv_L M}{\Gamma \vdash (t\ u) \in_2 N[x \leftarrow u]}$$

  By induction hypothesis, $\exists A_1, B_1$ such that $\Gamma \vdash t \in A_1$, $\Gamma \vdash u \in B_1$, $A_1 \equiv_L A$ and $B_1 \equiv_L B$. By transitivity of $\equiv_L$, $A_1 \equiv_L (x:M)N$ and $B_1 \equiv_L M$. Finally, remark 1 on $\equiv_L$ gives $\Gamma \vdash (t\ u) \in N[x \leftarrow u]$.

□

By this proposition and the theorem 27, we have a deterministic method to reconstruct a partial proof term from a specification and a weak extracted program. Given a proof $t : S$, the proof synthesis method applied to $\sigma(t)$, by the theorem 27, gives $\Gamma', \mathcal{L} \vdash_r t' : S'$ with $\sigma(t) = \sigma(t')$, $\sigma(S) = \sigma(S')$ and $\mathcal{L}$ a well formed logical context. The proof term contents holes corresponding to logical proofs to be retrieved by hand. These holes are existential variables in the logical context.

We can give some results directly from this theorem. First of all, what happens if the proof synthesis method is applied to a term who is exactly the weak extraction of a proof ? It can be proved the logical generated lemmas are provable. This gives a weak completeness notion for the method.

Let us explain this result. The inversion method defined by the proof of the theorem 27 can be seen as a functor F which, from a derivation of $\Gamma \vdash t \in S$ builds Γ', t', S' such that $\Gamma' \vdash_r t' : S'$.

Notation : let φ be a derivation $\Gamma \vdash_r t : S$, we note $\sigma(\varphi)$ the corresponding derivation for σ, that is $\sigma(\Gamma) \vdash \sigma(t) \in \sigma(S)$. We consider $\in$ and not $\in_2$, because the proof of the theorem 27 uses the rules of $\in$ and the completeness proof will follow the same structure than this proof. Moreover, the two typing judgments are equivalent.

Proposition 31. *Weak completeness of the inversion of the weak extraction :*
If $\varphi \equiv \Gamma \vdash_r t : S$ then $F(\sigma(\varphi)) \equiv \Gamma', \mathcal{L} \vdash_r t' : S'$ with t and t', S and S' identical modulo the identification of all the logical proofs of a same proposition, Γ' provable in Γ and with $\mathcal{L}$ containing logical lemmas which have a proof in Γ.

Proof. The proof follows exactly the same structure as for the inversion theorem. Each step shows that the constructed logical lemmas are provable. □

Then, let us see what happens if the method is applied to a provable specification and a term which is only coherent with it (it is not necessarily a trace of a proof). Naturally, if the program is not correct, the logical lemmas are not provable. But, it cannot be ensured that, if the program is correct, the logical lemmas are provable. We give examples where the program is correct w.r.t. to the specification but can generate non provable logical lemmas.

Example 7.

Let us take the specification $(x : nat)(x > 0) \rightarrow \{y : nat|y < x\}$ with a *Coq* syntax. An program coherent with this specification can be :

```
[x:nat]<{y:nat|y<x}>Match x with
              0
              [n:nat]n
       end
```

A non provable lemma is then generated : $(0 < 0)$.

A program that would generate only logical lemmas could be :

```
[x:nat]<(x>0)->{y:nat|y<x}>Match x with
              0
              [n:nat]n
       end
```

The method is then deterministic. It reconstructs a partial proof term from a weak extracted program and its specification. Proof obligations are left to the user. They represent logical properties that the program has to verify to valid its specification. This implies that we have a canonical representation for proofs. This approach is both a method of proving programs and a method of synthetically describing proofs.

4 A heuristic method

Weak extracted programs are not very natural. It would be nice to consider more natural programs, that is, programs with less specifications. In fact, we would like the programmer to write ${F_\omega}^{Ind}$ programs and the method to use unification to retrieve subspecifications. This is the goal of a tactic implemented in *Coq* and presented in [Par93, Par95].

This heuristic approach should follow the same method as the deterministic method, but the use of unification introduces non-determinism. Nevertheless, we introduce annotations in ${F_\omega}^{Ind}$ programs that the heuristic method could use and that allow to keep a certain notion of completeness. We briefly describe some heuristics and optimizations. We refer the reader to [Par93, Par95] for further details.

A notion of typing is always necessary. This typing is a little different from $\in$. We define $\in_3$ which replaces 2 and 3 by 5. Programs are now ${F_\omega}^{Ind}$ programs and this new rule only allows to go back to a typing corresponding to F_ω. It just introduces a notion of correction for programs.

Definition 32. Typing on natural programs :

$$\frac{\Gamma \vdash T' \in_3 s \quad \Gamma \vdash t \in_3 T \quad \varepsilon(T) = \varepsilon(T')}{\Gamma \vdash t \in_3 T'} \tag{5}$$

4.1 Heuristics

The problem is to retrieve specifications. For this, we use higher-order unification between the type of the program and the specification. Since unification is not deterministic, we have to make some choices. We prefer not to give all the details in this paper, but we can give an idea of where the problems are and how they can be solved. The main problematic cases are applications and eliminations. In an application, the problem is to retrieve all the logical arguments. Moreover, if one argument is a predicate, it is not deterministic to retrieve it by unification. For the elimination, the problem is to retrieve the elimination predicate. Heuristics deal with unification between the type of the program and the specification. The problem is to choose the good unifier. The choice is to always keep the most general unifier (in the sense of the one which bounds the biggest number of variables). In practice, it seems to be a good choice.

Example 8.

Let us give an example for a heuristic in an application.

Let us take the specification $\forall n.\exists m.(S\ m) = n \vee n = m = 0$. A strong extracted program coherent with this specification is :

```
[n:nat](nat_rec (sig nat) (exist nat 0)
                [y:nat][H:(sig nat)](exist nat y))
```

where `nat_rec` is the usual induction principle on natural numbers whose type is $(P : nat \to Set)(P\ 0) \to ((n : nat)(P\ n) \to (P\ (S\ n))) \to (n : nat)(P\ n)$. Note that in the previous program, `nat_rec` is a program variable and then its type is $(P : Set)P \to (nat \to P \to P) \to nat \to P$.

The proof term to be retrieved is :

```
[n:nat](nat_rec [n0:nat]{m:nat|(S m)=n0 \/ n0=m=0}
                (exist nat [m:nat]((S m)=0 \/ 0=m=0) 0 P1)
                [y:nat][H:{m:nat|(S m)=y \/ y=m=0}]
                  (exist nat [m:nat]((S m)=(S y) \/ (S y)=m=0)
                         y P2))
```

where `P1` and `P2` are proofs for $((S\ 0) = 0) \vee (0 = 0 = 0)$ and $((Sy) = (Sy)) \vee ((Sy) = y = 0)$.

Let us consider only the application case (the previous abstraction is trivial). We look for the predicate whose extraction is nat. `nat_rec` and its proof type are known. The searched predicate P has type $nat \to Set$. Let us use the specification to instantiate P. The head of the type of `nat_rec` (i.e. $(P\ n)$) can be unified with the specification (i.e. $\exists m.(S\ m) = n \vee n = m = 0$).

This is non deterministic and there are many possible unifiers :

1. $P = [n0]\exists m.(S\ m) = n0 \vee n0 = m = 0$.
2. $P = [n0]\exists m.(S\ m) = n0 \vee n = m = 0$.
3. $P = [n0]\exists m.(S\ m) = n \vee n0 = m = 0$.
4. $P = [n0]\exists m.(S\ m) = n \vee n = m = 0$.

The heuristic consists in keeping the unifier which bounds the biggest number of variables, thus the first one in this case. Two subgoals are generated :

$$\begin{cases} \exists m.(S\ m) = 0 \vee 0 = m = 0 \\ \forall n.\exists m.((S\ m) = n \vee n = m = 0) \to \exists m.(S\ m) = (S\ n) \vee (S\ n) = m = 0 \end{cases}$$

associated to two subprograms `(exist nat 0)` and

```
[y:nat][H:{m:nat|(S m)=y\/y=m=0}]
  (exist nat [m:nat]((S m)=(S y)\/(S y)=m=0) y)
```

4.2 Annotations

The heuristic method deals with cases where the higher-order unification succeeds. In practice, this seems to be quite often the case. However, to ensure the determinism of this method, we must be able to direct the choices and then to allow the programmer to add logical informations in the program. These annotations have to be taken into account by the method to take good decisions.

Syntax : S is an annotation for the program p is noted $p :: S$.

Definition 33. Annotations :

$$\frac{\Gamma \vdash p \in_3 S \quad \varepsilon(S) = \varepsilon(S')}{\Gamma \vdash (p :: S) \in_3 S'}$$

The intuition is the following. S is an annotation for p if p is coherent with S. Moreover, if p is coherent with S' then $p :: S$ is coherent with S' and $\varepsilon(S) = \varepsilon(S')$. Annotations are logical specifications that mark programs and allow to precise their specification. The goal is to use them to give explicitly the specification of a program when the heuristic method fails.

An annotation can contain free variables. These variables can be program variables or logical variables. It is necessary to be able to refer to logical variables inside a program and then to authorize logical λ-abstractions. This corresponds to the already existing rule of $\in_3$:

$$\frac{\Gamma[x : L] \vdash p \in_3 S}{\Gamma \vdash [x : L]p \in_3 (x : L)S} \text{ if } L : Type$$

4.3 Validity and Completeness

The heuristic method can either succeed or fail. There are three possible behaviors.

1. The method fails. Then, either the program is false, or there are not enough logical informations in the program. It can be necessary to add annotations in the program.

2. The method generates a set of logical lemmas that are not provable. Then, either the program is false, or there are not enough logical informations in the program. It can be necessary to add annotations in the program.
3. The method generates a set of provable logical lemmas. The theorem 27 ensures that the inversion of the weak extraction generates a partial proof of the initial specification. Then, the validity of the method is ensured.

We come now to the problem of the completeness of the heuristic method. The heuristic method fails on a correct program when there are not enough logical informations. The method on weak extracted terms is deterministic. As a consequence, the heuristic method succeeds on sufficiently annotated programs that is if the annotated program is the weak extracted program. The completeness of the heuristic method can then be stated.

Proposition 34. *If S is provable and if the associated program p is sufficiently annotated to be a trace of a proof of S, then the logical lemmas generated by the heuristic method are provable.*

Proof. Comes directly from the proposition 31. □

Then, if programs are weak extracted programs, the heuristic method is complete. The notion of sufficiently annotated programs corresponds in the worst case to a weak extracted term. In practice, annotations are typically elimination predicates. It corresponds to recursive structures in a program. One retrieves the problem of loop invariants search in the Hoare's logic [Hoa69]. We discuss this comparison in the conclusion.

4.4 Optimizations

The strong extraction can be optimized in order to generate programs that are closer to a natural form. Such optimizations generate programs more and more far from the proof. Until this section, we never consider such an optimization. By now, if we consider it, the heuristics to retrieve a proof term in such a case are much more harder. Special heuristics have to been developed to deal with these cases. A possible optimization of the strong extraction consists in distinguishing particular types that we call singleton types. Such an optimization suppresses constructors on singleton types and eliminations on singleton types. Then, for instance, heuristics have to be found to retrieve an elimination on a singleton type which has disappeared by extraction or a constructor on a singleton type.

Other optimizations can be introduced to consider more natural programs. An operator of well-founded recursion can be introduced. It is based on a well-founded induction principle. Extracted programs often contain trivial expressions such as `if b then true else false`. A more natural program is `b`. This implies that heuristics have to deal with such situations. They have to be able to retrieve the underlying structure of proof even if this is not the same in the program.

We do not detail at all this part but one can refer to [Par93, Par95] for an explanation of such optimizations.

5 Conclusion

We have defined a new extraction function for the Calculus of Inductive Constructions called the weak extraction. Weak extracted terms are condensed forms of proofs. They are ${F_\omega}^{Ind}$ programs annotated with specifications. A new notion of typing has been defined for such terms. This weak extraction has an important property : it can be inverted. The proof of inversibility gives an algorithm to reconstruct a proof from a program and its specification. Given a weak extracted program and its specification, a partial proof term, with "holes" corresponding to logical lemmas to be proved, can be deterministically reconstructed. The method is complete in the sense that there exists a proof for these logical lemmas in the original proof term.

A heuristic method can be deduced based on the same idea. The considered programs are ${F_\omega}^{Ind}$ programs. Heuristics are needed to retrieve intermediate subspecifications by unification. Annotations can be added in ${F_\omega}^{Ind}$ programs to explicitly give subspecifications. Then, the heuristic method is complete for sufficiently annotated programs. Optimizations can be introduced to consider more natural programs. This method presents two different aspects : it is both a method of proving programs and a method of synthetically describing proofs. Indeed, weak extracted programs can be seen as proofs descriptions.

This method can be compared with the Hoare's logic. Indeed, Hoare's logic proves that programs meet specifications. The structure of the program is analyzed and subspecifications corresponding to subprograms are generated until arriving at axioms. Our idea is exactly the same. Hoare's axioms corresponds to our logical lemmas. A known problem of Hoare's logic is the problem of retrieving loop invariants. This can be compared to our problem of retrieving elimination predicates in the heuristic method. In Hoare's logic, the user has to explicitly give invariants. In our method, it is necessary to add annotations. The main difference between the two methods is that they are in different frameworks. Hoare's logic considers imperative programs and specifications in first order predicate calculus. We consider functional programs and specifications in the Calculus of Inductive Constructions. Moreover, this work can be compared with the problem of retrieving the weakest precondition in Hoare's Logic. This problem consists in looking for a minimal precondition. This can be done by analyzing the program and the postcondition and building step-by-step the weakest precondition. Our method is close to this one. Indeed, we can reconstruct a proof term for a specification logically equivalent to the initial one. In fact, this specification contains less logical informations than the initial one. It can be considered as a construction of the "weakest specification".

The motivations of this work were that the user knows in general the algorithm he wants to prove. The same motivations are the basis of two works [BM92, Pol94]. These works construct a proof and a program hand in hand. They are not the same method as ours but the motivations are the same : allowing the user to direct the proof with a program.

This method has been implemented as a tactic in the *Coq* system. As we already said, the description of this tactic can be found in [Par93, Par95]. A library of examples has been developed with optimizations on the input language for programs. These examples are non trivial : they consist in algorithms on graphs, trees, lists (different sorts)... We can use the ProPre tactic of [MS92] which has been integrated into *Coq* to define functions via equations. This definition is transformed into a primitive recursive definition which can be used as input for our tactic. This allows to write input programs for our tactic more easily. In the current state, this is possible but only for 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 a more general way. Moreover, we hope to be able to write programs in more natural form than ${F_\omega}^{Ind}$ programs.

Finally, other methods of proof synthesis from programs could perhaps be developed. It should be possible to work with the program f and with the proof to do of $\forall x.(P\ x) \rightarrow (Q\ x\ (f\ x))$. This formula express that f realizes $\forall x.(P\ x) \rightarrow \exists y.(Q\ x\ y)$. The developed technology should not be very different but one can hope to prove more things since a proof of "f realizes B" can sometimes be done when a proof of B does not exist.

Acknowledgments

I am grateful to Christine Paulin-Mohring for supervising this work and for many helpful discussions.

A An example : a division algorithm

We detail here a complete example for a division algorithm. Some notions n the *Coq* syntax have to be known here. The `Program_all` tactic corresponds to the implementation of the proof synthesis method. It automatically generates a proof (with eventually holes corresponding to logical lemmas) from a program (in a *Coq* syntax) associated to a specification (in a *Coq* syntax too) by a command called `Realizer`.
Let us give the specification of the euclidean division algorithm. We want to prove that for a and b ($b > 0$), there exist q and r such that $a = b * q + r$ and $b > r$.
An ML program following this specification can be :

```
let div b a = divrec a where rec divrec = function
        if (b<=a) then let (q,r) = divrec (a-b) in (Sq,r)
                  else (0,a)
```

Suppose we give the following definition in *Coq* which describes what has to be proved, ie, $\exists q \exists r.\ (a = b * q + r \wedge\ b > r)$:

```
Coq < Inductive diveucl [a,b:nat] : Set :=
        divex : (q,r:nat)(a=(plus (mult q b) r))->
                     (gt b r)->(diveucl a b).
```

The decidability of the ordering relation has to be proved first, by giving the associated function of type `nat->nat->bool` :

```
Coq < Recursive Definition le_gt_dec_func : nat->nat->bool :=
Coq <   O m => true
Coq < | (S n) O => false
Coq < | (S n) (S m) => (le_gt_dec_func n m).
Coq <
Coq < Theorem le_gt_dec : (n,m:nat){(le n m)}+{(gt n m)}.
Coq < Realizer le_gt_dec_func.
Coq < Program_all.
Coq < Save.
```

Then the specification is `(b:nat)(gt b O)->(a:nat)(diveucl a b)`. The associated program corresponding to the ML program will be :

```
Coq < Realizer [b:nat](<nat*nat>rec div :: :: { lt }
                 [a:nat]<nat*nat>if (le_gt_dec b a)
                   then <nat*nat>let (q,r:nat) = (div (minus a b))
                        in <nat,nat>((S q),r)
                   else <nat,nat>(O,a)).
```

Where `lt` is the well-founded ordering relation defined by :

```
Coq < Definition lt := [n,m:nat](gt m n).
```

Note the syntax for recursive programs. The `rec` construction needs 4 arguments : the type result of the function (`nat*nat` because it returns two natural numbers) between `<` and `>`, the name of the induction hypothesis (which can be used for recursive calls), the ordering relation `lt` (as an annotation because it is a specification), and the program itself which must begin with a λ-abstraction. The specification of `le_gt_dec` is known because it is a previous lemma. The term `(le_gt_dec b a)` is seen by the `Program_all` tactic as a term of type `bool` which satisfies the specification `{(le a b)}+{(gt a b)}`. The tactic `Program_all` can be used, and the following logical lemmas are obtained :

```
6 subgoals
  b : nat
  H : (gt b O)
  a : nat
  ============================
   (well_founded nat lt)
subgoal 2 is:
 x=(plus (mult (S q) b) r)
subgoal 3 is:
 (gt b r)
subgoal 4 is:
 (lt (minus x b) x)
subgoal 5 is:
 x=(plus (mult O b) x)
subgoal 6 is:
 (gt b x)
```

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].

[CCF+94] C. Cornes, J. Courant, J.C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. Coq V5.10 Reference Manual. Technical report, 1994. Disponible en ftp anonyme sur ftp.inria.fr, à paraître.

[Con86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Coq85] T. Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[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 5.8. Technical Report 154, Projet Formel - INRIA-Rocquencourt-CNRS-ENS Lyon, May 1993.

[Gir72] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[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.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, 1984.

[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. *Proceedings of the 1992 workshop on types for proofs and programs*, June 1992.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory : an introduction.* Oxford Science Publications, 1990.

[Par93] C. Parent. Developing certified programs in the system Coq - The Program tactic. In H. Barendregt and T. Nipkow, editors, *Types For Proofs and Programs*, volume 806 of *LNCS*, pages 291–312, May 1993.

[Par95] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions.* PhD thesis, Ecole Normale Supérieure de Lyon, January 1995.

[PC89] F. Pfenning and Paulin-Mohring C. Inductively Defined Types in the Calculus of Constructions. In *5th International Conference on Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, pages 209–228, 1989.

[PM89a] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *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.

[Pol94] E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Technische Universiteit Eindhoven, 1994.