Formal Aspects of Computing (1997) 3: 1–000

Verifying programs in the Calculus of Inductive Constructions

Catherine Parent-Vigouroux [1]
VERIMAG
Centre Equation, 2 avenue de Vignate, 38610 Gières, France
e-mail : Catherine.Parent@imag.fr

Keywords: λ-calculus, proofs of programs, extraction, Calculus of Constructions

Abstract. This paper deals with a particular approach to the verification of functional programs. A specification of a program can be represented by a logical formula [Con86, NPS90]. In a constructive framework, developing a program then corresponds to proving this formula. Given a specification and a program, we focus on reconstructing a proof of the specification whose algorithmic contents corresponds to the given program. The best we can hope is to generate proof obligations on atomic parts of the program corresponding to logical properties to be verified. First, this paper studies a weak extraction of a program from a proof that keeps track of intermediate specifications. From such a program, we prove the determinism of retrieving proof obligations. Then, heuristic methods are proposed for retrieving the proof from a natural program containing only partial annotations. Finally, the implementation of this method as a tactic of the *Coq* proof assistant is presented.

1. Introduction

A large part of the programming task is devoted to verifying that a program execution indeed satisfies the programmer's intentions. While testing the programs

Correspondence and offprint requests to: Catherine Parent-Vigouroux - VERIMAG - Centre Equation - 2 avenue de Vignate - 38610 Gières - France

[1] 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".

on various inputs can help to detect errors and to increase our confidence, only formal mathematical methods can guarantee correctness.

Program specifications

Specifying a program consists of explicitly stating its desired behavior, in the sense of "what" it does, but not "how" (see [Hoa69]). Input and output properties have to be clearly stated. As an example, consider the specification of a division algorithm on natural numbers. The algorithm takes two natural numbers a (*dividend*) and b (*divisor*) as inputs, and computes two other natural numbers q (*quotient*) and r (*remainder*). The quotient and the remainder satisfy properties with respect to a and b: $a = b * q + r$ and $b > r$. The input b has to be strictly positive (we talk about natural numbers). Furthermore, a specification can be met by different algorithms since it describes *why* a program is valid but not *how*. Similarly, a sorting algorithm can be specified as, given an input list, the output is an ordered permutation of the input list, regardless of the specific sorting algorithm used.

Proofs of programs

The interest of verifying programs lies in proving that each part of a program satisfies a precise specification. A program is then documented and more easily portable. A proof of the above mentioned specification of the division, which contains an explicit method for constructing q and r such that $(a = b*q+r) \wedge (b > r)$, is called a *constructive* proof.

Proofs in type theory

We focus here on the particular framework of the Calculus of Inductive Constructions [Coq85, Coq89]. In such a scheme, a specification is a logical formula linking the input and the output. In the division example, our informal specification now corresponds to the following logical formula: $\forall a, b(b > 0) \rightarrow \exists q, r(a = b*q+r) \wedge (b > r)$. The Curry-Howard isomorphism [How80] identifies specification and type, proof and well-typed term. It is the origin of systems like Automath [dB80], the intuitionistic type theory of Martin-Löf [ML84] or the Calculus of Constructions. Proofs and programs can then be uniformly manipulated. Proving a formula such as the specification of the division is finding a well-typed term whose type is this formula.

A specification has the general form $\forall x P(x) \rightarrow \exists y Q(x, y)$, that can be read as, given an input x verifying the property $P(x)$, there exists an output y verifying $Q(x, y)$. In the case of the division, $P(a, b)$ is $b > 0$ and $Q(a, b, q, r)$ is $(a = b * q + r) \wedge (b > r)$. For a specification $\forall x P(x) \rightarrow \exists y Q(x, y)$, the proof explicitly contains a witness y and a method for building this witness. Moreover, the proof takes as an argument a proof of $P(x)$ and returns a proof of $Q(x, y)$.

Program extraction

In this framework, a proof can be seen as a program containing its correctness proof. Consider again the division example. The corresponding mathematical proof skeleton of $\forall a, b(b > 0) \rightarrow \exists q, r(a = b * q + r) \wedge (b > r)$ is the following (by induction on a). If a is zero, then $q = r = 0$ and we have $0 = b * 0 + 0$ and

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

Table 1. Euclidean division ML program

$b > 0$. Otherwise, let us suppose q and r to be known for the predecessor of a, that is, $a - 1 = b * q + r$ and $b > r$, and let us build q' and r' for a. Two cases have to be distinguished: either $b \leq (r + 1)$, or $b > (r + 1)$. In the first case, the quotient q' is $q + 1$, the remainder r is 0 and we have $a = b * (q + 1) + r$ and $b > 0$. In the second case, the quotient q' is q, the remainder r' is $r + 1$ and we have $a = b * q + r + 1$ and $b > (r + 1)$.

The proof contains an explicit construction of q and r but takes also as an argument a proof of $b > 0$ and returns a proof of $(a = b * q + r) \wedge (b > r)$ for each possible case of a. Proving the specification corresponds to building a correct program with respect to this specification. But, the generated program is inefficient since it contains its correctness proof (in the program, the justification of $a = b * q + r$ is not interesting).

To obtain more realistic programs, the real constructive part of the proof has to be distinguished from the correctness proof part. The first part is the computational part of the proof, the second the logical. Obviously, only the computational part needs to be kept as a program. In [PM89b], this operation is called "program extraction". This approach validates a program by proving a specification and synthesizing a program from this proof. A logical language called the Calculus of Inductive Constructions supports proof development and the programming language is the ${F_\omega}^{Ind}$ system. The proof language is more powerful than the programming language since it allows writing properties of elements of the programming language.

In this paper, we consider the inverse problem. Indeed, proving a specification is not a trivial exercise and is not automatic. A more natural idea is to directly prove the correctness of a program, in the same sense as in Hoare's logic. We want to consider a specification and a program and build step by step a proof skeleton of the program. The program could be proved correct when only axioms remain. Hoare's logic is a framework for imperative programs. We place ourselves in a framework for typed functional languages.

In Hoare's logic, rules transform a specification into a simpler specification following the structure of a program until axioms are obtained. Here, we want to retrieve a proof skeleton term from two inputs: a program and a specification which is the type of the proof term. We will also use the structure of the program to derive the proof skeleton. Indeed, even if a program is only a method for computation, it contains information about the underlying proof. A conditional or a test in a program corresponds to a case analysis in a proof. A recursive call in a program corresponds to an invocation of an induction hypothesis in a proof. If a program uses sub-programs, the proof uses lemmas. In the Euclidean division example, the ML program is given by Table 1.

The specification of this program is $\forall a, b(b > 0) \rightarrow \exists q, r(a = b*q+r) \wedge (b > r)$. Let us analyze the structure of the program with respect to the structure of the proof. The case analysis on a corresponds to an induction on a in the proof. Two specifications can be deduced $\exists q, r(0 = b * q + r) \wedge (b > r)$ and $\forall n \exists q, r(n - 1 =$

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

Table 2. Annotated euclidean division program

$b*q+r) \wedge (b>r) \rightarrow \exists q,r(n=b*q+r) \wedge (b>r)$. Then, the case analysis on b corresponds to a case analysis in the proof and two sub-specifications can be deduced $(n=b*(q+1)+0) \wedge (b>0)$ and $(n=b*q+r+1) \wedge (b>r+1)$. It is clear with this example that a program contains much information about its correctness proof. This can be illustrated by annotating the program with sub-specifications that can be retrieved, as in Table 2.

Proof skeleton synthesis

In this paper, we develop a method for synthesizing proof skeletons from programs. Programs are typed functional programs in the ${F_\omega}^{Ind}$ system. The difficulties lie in reconstructing the entire proof term only from the program and the specification. The basic principle consists of analyzing the structure of the program; then, some simpler sub-specifications associated to some simpler sub-programs are generated from the specification. With each generation of a sub-specification, a part of the proof term is built. It can then be proved that a partial proof term with "holes" can be generated, where holes correspond to logical properties to be satisfied by the program.

Nevertheless, this is not trivial. A recursive call corresponds to an invocation of an induction hypothesis. In practice, we do not know which specification the induction has to verify. In the division example, the case analysis on a corresponds to an induction on a. The current specification is $b>0 \rightarrow \exists q,r(a=b*q+r) \wedge (b>r)$. Should the induction be applied to this specification or to the specification $\exists q,r(a=b*q+r) \wedge (b>r)$ in the context $b>0$? In this example, both solutions work. In the previous annotated program, the second solution has been chosen. But one can easily imagine that, in more complicated examples, the first solution could generate more complicated proofs than the second (or even impossible proof obligations), or conversely. This problem corresponds to the problem of finding the good invariant in Hoare's logic, where the programmer has to intervene.

In our case, the intervention of the programmer will be necessary too. Indeed, even if the program contains some logical information about its correctness proof, it may not contain enough. Therefore, we need to define a new language where programs can be annotated by logical information concerning the underlying proof. This new language should be rich enough to allow retrieving the necessary information. This language has to be placed between a logical language such as the Calculus of Inductive Constructions and a programming language such as the system ${F_\omega}^{Ind}$. This requires a method to retrieve and reuse such annotations in a program.

Inversion of the extraction

It is then clear that we should look for a process that inverts program extraction [PM89b]. The extraction omits logical information from proofs to obtain programs and we want to retrieve logical information from programs and specifications to build proofs. Then, as we said before, we look for a language richer than ${F_\omega}^{Ind}$.

In this paper, we modify the extraction procedure of [PM89b] in order to obtain proof traces containing sufficient information on proofs. In the division example, the method of [PM89b] extracts a program without any annotation as the program of Table 1. A weaker extraction could give the annotated program of Table 2. We then define a new extraction function using a new language of annotated programs. From such programs and their specification, we prove that we can deterministically retrieve partial proof terms. Programs are then sufficiently explicit and allow to automatically retrieve the underlying proof (modulo some properties corresponding to the remaining holes in the partial proof term). Then, we prove that this new extraction function is invertible. From this proof, we can deduce an algorithm for reconstructing proofs skeletons from programs in this new language and show this method to be valid and complete (in a certain sense of completeness).

The problem of this new language is that it does not allow to write sufficiently natural programs. Programs are not human-friendly since they contain too many annotations. We propose then a heuristic method, based on the previous method on annotated programs, to retrieve a proof skeleton from a program in ${F_\omega}^{Ind}$. Due to the heuristics, this method is no longer complete. We then allow to add annotations in ${F_\omega}^{Ind}$ similar to the annotations of the previous language of annotated programs. This heuristic method can use annotations and then use the deterministic method. Then, as a direct consequence, the heuristic method is complete for *sufficiently annotated* programs.

Finally, the heuristic algorithm has been implemented in the *Coq* system (which is itself an implementation of the Calculus of Inductive Constructions) as a tactic. This tactic builds a partial proof term from a program (possibly with annotations) and a specification. The user has to prove remaining logical lemmas with the usual *Coq* tactics. An important library of examples has been developed.

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 is extended with primitive inductive types. Then, we explain the motivations for extracting programs and present the extraction method of [PM89a].

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

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

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

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

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

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

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

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

Table 3. Inference rules of the pure Calculus of Constructions

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 and those marked with $Prop$ are logical terms.

Definition 1. Syntax of terms in the pure Calculus of Constructions:

The terms and types of the Calculus of Constructions are given by the following syntax rules with t standing for terms, U for types, x for term variables and X for type variables.

$$\begin{array}{rcl} t & ::= & x \mid (t\ t) \mid [x : U]t \mid (t\ U) \\ U & ::= & S \mid X \mid [X : U]U \mid (U\ U) \mid (X : U)U \mid (U\ t) \\ S & ::= & Prop \mid Set \mid Type \mid TypeSet \end{array}$$

The calculus itself is defined by a set of inference rules (see Table 3). Types are generally written in upper case, terms in lower case. The judgment $\Gamma \vdash t : U$ states that the term t has type U in the context Γ. A context is a list of typed variables given between square brackets and separated by semicolons. Adding a variable x of type A to a context Γ is denoted by $\Gamma[x : A]$. Well formed contexts are defined by the first two rules. The sixth, seventh and eighth rules allow to derive types for products, abstractions and applications. The last rule says that two types which are equal modulo β-reduction (if their normal forms are equal) can be identified.

2.2. Inductive Types

We consider the Calculus of Constructions without universes. Since data types cannot be easily coded in this calculus, we add primitive inductive types which allow defining 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 (successor). This section presents the syntax and rules for inductive types. Our presentation is much inspired by [PM93, PC89], but we treat the parameters of inductive types explicitly. In [PM93], inductive types are presented without parameters which are treated as λ-abstractions.

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

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

Definition 3. Strictly positive type:
A type is strictly positive with respect to a type variable X if it has the following syntax:

$$Pos ::= X \mid (Pos\ t) \mid (x : U)Pos \quad \mathit{with}\ X \notin FV(U,t)$$

Definition 4. Constructor types (X necessarily appears in C):

$$C ::= X \mid (C\ t) \mid (P \rightarrow C) \mid (x : U)C$$

with X not appearing in t and U, and P a type which is strictly positive w.r.t. X, where t is a term or a type and U a type.
We write *constructor*(C,X) for C *is a constructor type w.r.t.* X.

Definition 5. Inductive Types:
An inductive type is a new type defined as follows:

$$U ::= \mathsf{Ind}[p_1 : P_1| \dots |p_m : P_m](X : Ar)\{c_1 : C_1| \dots |c_n : C_n\}$$

Ar is an arity, p_i are the name of the parameters of the inductive type and P_i their types, m the number of parameters, n the number of constructors and c_i the name of the constructors of the inductive type and C_i their types, respecting *constructor*(C_i,X).

In the following, P_i will always denote parameter types and C_i constructors types.

Definition 6. Constructors:
A constructor is a new term defined as follows:

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

where i is the selector for the i-th constructor of the inductive type U. $\mathsf{Constr}(i,U)$ denotes the i-th constructor of U while c_i of definition 5 is just its name (for instance, the name of $\mathsf{Constr}(1,nat)$ is S).

Example

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

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

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

$$\begin{array}{lll} sumor & := & \mathsf{Ind}[A\!:\!Set; B\!:\!Prop](X\!:\!Set)\{inleft : A \rightarrow X \\ & & |inright : B \rightarrow X\} \end{array}$$

- The type of lists can be defined by:

$$list := \mathsf{Ind}[A : Set](X : Set)\{nil : X|\ cons : A \rightarrow X \rightarrow X\}$$

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

□

Then, a notion of elimination can be defined.

Definition 7. Elimination:
An elimination is a new term defined as follows:

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

where t_0 is the term on which the elimination is done, U the elimination predicate (type of the elimination term) and t_i the different branches of the elimination.

Notations:

- A vectorial notation is used for lists of variables. We write $\vec{a}$ for $a_1 \ldots a_n$ and $(\vec{a} : \vec{A})t$ for $(a_1 : A_1) \ldots (a_n : A_n)t$.
- Eliminations will be presented with the syntax $\mathsf{Elim}(e, Q)\{f_1| \ldots |f_n\}$ in the following. Then e denotes the eliminated term, f_i the elimination branches and Q the elimination predicate. The $\vec{c}$ notation denotes a list of c_i which are just names of constructors but not terms.
- ind_n will denote any inductive type with n constructors. If the number of constructors is meaningless, then the ind notation is used.

There are two kinds of eliminations: the dependent elimination and the non-dependent elimination. 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, $A \equiv (\vec{x} : \vec{A})s$, X a variable of type A, Q a variable of type $(\vec{x} : \vec{A})(X\ \vec{x}) \rightarrow s'$, C a constructor type w.r.t. X and e a term of type C, we define a new type denoted by $C\{X, Q, e\}$ by induction on C:

$$\begin{array}{rcl} (P \rightarrow C)\{X, Q, e\} & = & (p : P)P\{X, Q, p\} \rightarrow C\{X, Q, (e\ p)\} \\ & & \quad X \text{ strictly positive in } P \\ (x : U)C\{X, Q, e\} & = & (x : M)C\{X, Q, (e\ x)\} \quad X \text{ not in } U \\ (X\ \vec{a})\{X, Q, e\} & = & (Q\ \vec{a}\ e) \end{array}$$

Then, three new rules are defined for typing an inductive type, a constructor and an elimination. They are given in Table 4.

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

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

$$\frac{\begin{array}{c}\forall i = 1 \ldots n, \Gamma \vdash f_i : C_i\{(ind_n\ \vec{p}), Q, (\mathsf{Constr}(i, ind_n)\ \vec{p})\}\\ \Gamma \vdash e : (ind_n\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q : (\vec{x}:\vec{A})(ind_n\ \vec{p}\ \vec{x}) \rightarrow s'\end{array}}{\Gamma \vdash \mathsf{Elim}(e, Q)\{f_1 | \ldots | f_n\} : (Q\ \vec{a}\ e)}$$

Table 4. Inductive Types in the Calculus of Constructions

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

We ignore the details of the particular notion of reduction on inductive types, called the ι-reduction, that is used. A definition of this reduction can be found in [PM93].

Example

We show an elimination to illustrate 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 has type $(list\ A) \rightarrow Set$. The elimination has two cases f_1 and f_2 with the following types:

$$\begin{array}{lcl} f_1 & : & (Q\ (nil\ A)) \\ f_2 & : & (x : A)(l : (list\ A))(Q\ l) \rightarrow (Q\ (cons\ A\ x\ l)) \end{array}$$

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

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

□

***Coq* notations:**

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

Example

A function giving the tail of a list can be defined by an elimination on this list. If the list is empty, then the output is the empty list.

```
[l:(list A)]<(list A)>Match l with
             nil
             [a:A][m:(list A)][H:(list A)]m   end
```

□

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 initially been marked by the programmer with $Prop$ for logical parts and Set for computational (or informative) parts. Moreover, dependent types are removed. Then, resulting terms are typable in the system ${F_\omega}^{Ind}$ (F_ω of [Gir72, PM89b] plus inductive types). ${F_\omega}^{Ind}$ corresponds to the Calculus of Inductive Constructions restricted to $Set : TypeSet$ and without the pair $(Set, TypeSet)$ in the (s, s') rule. This corresponds to a calculus without types that depend on terms.

An auxiliary definition of level on terms is needed. 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 $N = TypeSet$ or $\exists s.N : s$ where s is an element of S defined in Table 3. Four levels can then be defined.

Definition 9. 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

- Level 0: if A is a type, $A \rightarrow Set$ has level 0.
- Level 1: inductive types have level 1.
- Level 2: constructors of inductive types 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 10. Informative arity:
An informative arity is a term with the following syntax:

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

Definition 11. 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.

$$\varepsilon_0(Set) = Set$$
$$\varepsilon_0((x:A)B) = \begin{cases} \varepsilon_0(A) \to \varepsilon_0(B) & \text{if A informative of level 0} \\ \varepsilon_0(B) & \text{otherwise} \end{cases}$$
$$\varepsilon_1(x) = \bar{x} \quad \text{where } \bar{x} : \varepsilon_0(T_x) \text{ if } x : T_x$$
$$\varepsilon_1((x:A)B) = \begin{cases} (\bar{x} : \varepsilon_i(A))\varepsilon_1(B) & \text{if A informative of level } i \leq 1 \\ \varepsilon_1(B) & \text{otherwise} \end{cases}$$
$$\varepsilon_1([x:A]B) = \begin{cases} [\bar{x} : \varepsilon_0(A)]\varepsilon_1(B) & \text{if A informative of level 0} \\ \varepsilon_1(B) & \text{otherwise} \end{cases}$$
$$\varepsilon_1(A\ B) = \begin{cases} (\varepsilon_1(A)\ \varepsilon_1(B)) & \text{if B informative of level 1} \\ \varepsilon_1(A) & \text{otherwise} \end{cases}$$
$$\varepsilon_1(\mathsf{Ind}[\vec{p} : \vec{P}](X:A)\{\vec{c} : \vec{C}\}) =$$
$$\mathsf{Ind}[\bar{\vec{p}} : \varepsilon_0(\vec{P})](\bar{X} : \varepsilon_1(A))\{\bar{\vec{c}} : \varepsilon_1(\vec{C})\} \quad \text{for informative } P_i \text{ of level 0}$$
$$\varepsilon_2(x) = \bar{x} \quad \text{where } \bar{x} : \varepsilon_1(T_x) \text{ if } x : T_x$$
$$\varepsilon_2([x:A]B) = \begin{cases} [\bar{x} : \varepsilon_i(A)]\varepsilon_2(B) & \text{if A informative of level } i \leq 1 \\ \varepsilon_2(B) & \text{otherwise} \end{cases}$$
$$\varepsilon_2(A\ B) = \begin{cases} (\varepsilon_2(A)\ \varepsilon_i(B)) & \text{if B informative of level } i \leq 2 \\ \varepsilon_2(A) & \text{otherwise} \end{cases}$$
$$\varepsilon_2(\mathsf{Elim}(e,Q)\{f_1|\ldots|f_n\}) = \mathsf{Elim}(\varepsilon_2(e),\varepsilon_1(Q))\{\varepsilon_2(f_1)|\ldots|\varepsilon_2(f_n)\}$$
$$\varepsilon_2(\mathsf{Constr}(i,ind,\vec{p})) = \mathsf{Constr}(i,\varepsilon_1(ind),\varepsilon_1(\vec{p}))$$

Table 5. Strong extraction on terms

- A term of level 2 is informative if the type of its type is *Set*.

A term is *logical* if it is not informative.

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 regardless of their level. For constructors, only the informative parameters of level 1 have to be kept. Constructors parameters are then not treated as usual application arguments. A constructor applied to its parameters is written $\mathsf{Constr}(i, ind, \vec{p})$ where i is the number of the constructor, ind is the corresponding inductive type and $\vec{p}$ is the vector of parameters. We change the notation for constructors only for presenting the strong extraction. This is useful only for this presentation and not for the rest of the paper where we keep the original notation.

The extraction function of [PM89b] can now be presented (see Table 5). In the following, we refer to this function as the strong extraction function. Intuitively, this function suppresses all the logical information from a proof term. Moreover, it suppresses all the dependencies of types on terms and can be applied only on informative terms.

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

Definition 12. 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 1. (due to C.Paulin-Mohring [PM89b])
If $\Gamma \vdash t : T$ then $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}(T)$.

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

Example

In order to illustrate the definitions, let us consider 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:

```
[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 replacement 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 as:

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

□

2.4. Non-invertibility of the strong extraction

The extraction function allows synthesizing 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 information needed to prove properties is reflected by the program.

Example

For a specification $\{x : A|(P\ x)\}$ and a program a, we need to reconstruct a proof of $(P\ a)$ for an arbitrary P. In general, this is impossible, since it is undecidable whether such a proof even exists. It is impossible to retrieve a proof only from its type.

There is another problem: intermediate specifications disappear with extraction. Consider the 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 even and odd n. 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 the logical information is missing from 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

This new extraction function attempts to keep a sufficient amount of information for retrieving a proof skeleton from a weak extracted term. Since we show the problem is to retrieve intermediate specifications, this new function keeps the specifications and no longer suppresses them. Then, this function only suppresses logical proofs.

3.1. Definitions and properties

In section 2 , arguments of constructors had a special status. In this section, they are treated as ordinary arguments and we want to keep all their informative parameters regardless of their level. Then, we definitely adopt the original notation $(\mathsf{Constr}(i, ind)\ \vec{p})$: constructors are no longer bound to the constructor notation but are usual arguments of an application. The $\mathsf{Constr}(i, ind, \vec{p})$ notation was only useful to present the strong extraction in the previous section.

Definition 13. 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.

Table 6 gives the weak extraction rules on terms. Only the dependencies w.r.t. logical proofs (of level 2) are suppressed unlike Table 5 where informative proofs and logical specifications were suppressed too.

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

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

$$\sigma_2(\mathsf{Elim}(e,Q)\{f_1|\ldots|f_n\}) = \mathsf{Elim}(\sigma_2(e),\sigma_1(Q))\{\sigma_2(f_1)|\ldots|\sigma_2(f_n)\}$$

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

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

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

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

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

$$\sigma_i(s) = s \quad s \in S$$

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

Table 6. Weak extraction on terms

In order for terms and types to be coherent, logical types of level 1 are suppressed in λ-expressions. Note the rule for products of level 1 where nothing, and in particular specifications, is suppressed.

Weak extraction is invertible, as can be seen from the example showing the non-invertibility of the strong extraction. There, it was impossible to keep information on *even* and *odd*, which now becomes possible due to the rule (*) in Table 6.

Example

Let us show the weak extraction on the predecessor example. Note that weak extraction does not change the inductive types *sig* and *sumor*. The weak extracted term is then the following (let us call it *weak_pred*):

```
[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 as the strong extracted program. The only difference is in the annotations corresponding to intermediate specifications. □

The terms obtained by applying this function to 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 information allowing the inversion of the weak extraction (information on *even* and *odd* in our previous example for non-invertibility). We need a new notion of typing for these particular terms since they are neither typable in the Calculus of Inductive Constructions nor in ${F_\omega}^{Ind}$.

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

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

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

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

Table 7. Typing on weak extracted terms

Table 7 contains the rules that differ from the corresponding rules of Table 3. The judgment $\Gamma \vdash t \in U$ states that the term t has the weak type U in the context Γ. The program t is said to be coherent with the specification U. It allows, given a program and a specification, to check whether they are coherent.

The rules of Table 7 are similar to those of the Calculus of Inductive Constructions (see Table 3) but the difference is in rules 1, 2, 3 and in the application formation, which 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 maintained kept in types (due to the rule on the product of level 1). Rules 2 and 3 are direct consequences of this fact. The rules 2 and 3 deal with logical products in types that have no corresponding argument in an application and allow to add or suppress any logical products in a type.

Remark.

1. In 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.
2. The definition of $C\{X,Q,e\}$ is modified into $C\{X,Q,e\}_\sigma$ since proof variables cannot appear in constructors. Indeed, if a constructor is $e:(n:nat)(n > 0) \rightarrow C$, the elimination principle should be $(n:nat)(n>0) \rightarrow (P\ (e\ n))$ and not $(n:nat)(h:(n>0))(P\ (e\ n\ h))$ which is calculated by $C\{X,Q,e\}$. The new definition of $C\{X,Q,e\}_\sigma$ avoids this problem.

Definition 14. Given two sorts s and s', a variable X of type $A \equiv (\vec{x}:\vec{A})s$, a variable Q of type $(\vec{x}:\vec{A})(X\ \vec{x}) \rightarrow s'$, a constructor type C w.r.t. X and a term e of type C, we define $C\{X,Q,e\}_\sigma$ by induction on C:

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

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

$$(X\ \vec{a})\{X,Q,e\}_\sigma = (Q\ \vec{a}\ e)$$

Remark.

1. In the first case of definition 14, P cannot have the type $Prop$. An inductive type in an elimination is always informative. Moreover, P is strictly positive w.r.t. X, then, $P:Set$.

$$\dfrac{\vdash 0=0 \in Prop \quad \dfrac{\vdash 0=0 \in Prop \quad \dfrac{\vdash \{m:nat|(S\ m)=0\} \in Set \quad \vdash \overline{inright} \in (A:Set)(B:Prop)(B \rightarrow A+\{B\})}{\begin{array}{l}\vdash (\overline{inright}\ \{m:nat|(S\ m)=0\}) \\ \quad \in (B:Prop)(B \rightarrow \{m:nat|(S\ m)=0\}+\{B\})\end{array}}}{\vdash \sigma(T_1) \in (0=0) \rightarrow \{m:nat|(S\ m)=0\}+\{0=0\}}}{\vdash \sigma(T_1) \in \{m:nat|(S\ m)=0\}+\{0=0\}}$$

Table 8. A typing derivation tree example

2. In the following, σ applies to all terms except logical proofs and ε applies only on informative terms.

Example Typing the weak extraction of the *weak_pred* example.

We will only detail the typing of the term $\sigma(T_1)$ where $T_1 \equiv (inright\ \{m : nat|(Sm)=0\}\ (0=0)\ (refl\_equal\ nat\ 0))$ is the first argument of the elimination. We have the following type information:

- $\overline{sig}\ :\ (A:Set)(A \rightarrow Prop) \rightarrow Set$;
- $nat\ :\ Set$;
- $[m\ :\ nat]((S\ m)\ =\ n0)\ \ :\ \ nat \rightarrow Prop$ because the equality on natural numbers has type $nat \rightarrow nat \rightarrow Prop$;
- $\overline{inright}\ :\ (A:Set)(B:Prop)(B \rightarrow (sumor\ A\ B))$.

The typing derivation tree appears in Table 8. With the previous points, the leaves of this tree (at the top of the table) are trivial. Note the use of rule 2, without which the term could not be typed. The same calculation can be done to obtain the type of the global term *weak_pred*:

$$weak\_pred \in (n:nat)\{m:nat|(S\ m)=n\}+\{0=n\}$$

□

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

Proposition 2. 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 3. First substitutivity lemma:

For all terms 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 is the level of t and x. Moreover, information of x and t has to be coherent (one is logical if and only if the other is).

Proof. By induction on the structure of T. □

Proposition 4. Second substitutivity lemma:

For all logical terms t of level 2:

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

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

Proof. By induction on the structure of T. □

Proposition 5. Coherence of 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 weak extraction

Given a weak extracted program and a specification, we show that we can reconstruct a partial proof term with "holes" corresponding to the logical proofs left to the user. Then, all the logical proofs of a specification have to be identified. The proof of invertibility gives such an algorithm. Some restrictions on the framework have to be stated in order to be able to invert the function.

Let us take an example to illustrate why logical proofs have to be identified. Consider 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. There exist many possibilities for h_1. Then, we want to identify all the proofs of a same logical proposition: we need an *irrelevance* of logical proofs.

Let us take another example to point out which restrictions are useful. Consider a term T whose type is $(x : nat)(P : (y : nat)(x > y) \rightarrow Set)(P\ 1\ h_1) \rightarrow (P\ 2\ h_2)$. Here, logical proofs appear in an arity and not in the type itself. Retrieving a proof term from a weak extracted term containing T is impossible since logical proofs to be retrieved are too deeply hidden in the term. We do not want to accept such programs.

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). Hence we consider the Calculus of Inductive Constructions without the $(Prop, Type)$ and $(Prop, TypeSet)$ pairs in the (s, s') rule.

Remark. In such a framework, 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 framework implies its validity 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 $A = [x : B']C'$.

Definition 15. Judgments restricted to the Calculus of Inductive Constructions without the $(Prop, Type)$ and $(Prop, TypeSet)$ pairs in the (s, s') rule are denoted by $\vdash_r$.

We now want to state the irrelevance of logical proofs in such a framework. We need to extend the equality notion between two terms to an equality between two typed terms.

Definition 16. Extended equality to two typed terms of $\vdash_r$ (denoted by $=_e$):
Given two types B and B', if $B =_{\beta\iota} B'$ and if l and l' are two logical terms with level 2 then $B[x \leftarrow l] =_e B'[x \leftarrow l']$.

The following proposition can be proved:

Proposition 6. Irrelevance of logical proofs:
Given two terms A and A' of $\vdash_r$ such that $type(A) =_e type(A')$ (where $type(A)$ is the type of A) and A and A' are not logical proofs, then $\sigma_i(A) = \sigma_i(A') \Rightarrow A = A'$.

Proof. By induction on the η-long normal forms of A and A'. General forms of A and A' are:
$A = [x_1 : A_1] \dots [x_n : A_n](y_1 : B_1) \dots (y_m : B_m)(c\ t_1 \dots t_p)$.
$A' = [x'_1 : A'_1] \dots [x'_n : A'_n](y'_1 : B'_1) \dots (y'_m : B'_m)(c'\ t'_1 \dots t'_q)$.
Since $type(A) =_e type(A')$ and A and A' are in η-long form, there is exactly the same number of x_i and x'_i, and $\sigma(A_i) = \sigma(A'_i)$. Then, $A_i = A'_i$ by the induction hypothesis.
Since $\sigma((x : A)B) = (x : \sigma(A))\sigma(B)$ (see the remark on the previous page) and $type(A) =_e type(A')$, there is exactly the same number of y_i and y'_i , and $\sigma(B_i) = \sigma(B'_i)$. Then, $B_i = B'_i$ by the induction hypothesis.
Since $\sigma(A) = \sigma(A')$, $\sigma(c\ t_1 \dots t_p) = \sigma(c'\ t'_1 \dots t'_q)$. Then $\sigma(c) = \sigma(c')$.
Then, three cases are possible:

- c and c' are variables or constructors and then $c = c'$.
- $c = ind$ and $c' = ind'$. Then, $\sigma(ind) = \sigma(ind')$ and σ applied to an inductive type can be reduced into the application of σ on each subpart of the inductive type. The induction hypothesis can be applied on each subpart, then $ind = ind'$ and $c = c'$.
- $c = \mathsf{Elim}(t, Q)\{f_i\}$ and $c' = \mathsf{Elim}(t', Q')\{f'_i\}$.
  In this case, $\mathsf{Elim}(\sigma(t), \sigma(Q))\{\sigma(f_i)\} = \mathsf{Elim}(\sigma(t'), \sigma(Q'))\{\sigma(f'_i)\}$. We know $\sigma(t) : \sigma(ind)$ and $\sigma(t') : \sigma(ind')$. The induction hypothesis can be applied on ind and ind', then $ind = ind'$. It can be applied on t and t', Q and Q', f_i and f'_i as well. Then, $t = t'$, $Q = Q'$ and $f_i = f'_i$ and consequently, $c = c'$.

In all the three cases, $c = c'$ can be deduced. Two terms $(c\ t_1 \dots t_p)$ and $(c\ t'_1 \dots t'_q)$ with the same type and the same σ have now to be compared. The type of their head is identical by the previous explanation. Let us suppose $p \leq q$ (the problem is symmetrical). There is an exact correspondence between t_i and t'_i for $i \leq p$. Since terms are in η-long form, there are no more products after the applications of all t_i. Then, $p = q$, and finally, $A = A'$. □

With such restrictions to $\vdash_r$ which give the irrelevance of logical proofs, the invertibility theorem can be stated. 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 skeleton for p can be reconstructed with holes corresponding to logical proofs.

Theorem 1. Invertibility 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 there exist a term t', a well formed logical context $\mathcal{L}$ and a type S' such that $\Gamma, \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$, $\sigma(S') = T$.

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

We only give the two most significant cases, namely the application and rule 2.

- 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, condition 1 imposes arities that do not depend 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 6). $A' = A''$ can be deduced from $\sigma(A') = \sigma(A'')$. Then, $\exists \mathcal{L}''$, union of $\mathcal{L}$ and $\mathcal{L}'$, such that $\Gamma, \mathcal{L}'' \vdash_r (t'\ u') : B'[x \leftarrow u']$, and $(t'\ u')$ is the proof term with the following properties:

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

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

- Rule 2:

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

By induction hypothesis, $\exists t', \mathcal{L}$ and S' such that $\Gamma, \mathcal{L} \vdash_r t' : S'$ with $\sigma(t') = t$ and $\sigma(S') = A \rightarrow U$. 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 hypothesis are used but no new one is introduced.

□

This proof is constructive. It explicitly gives a method to construct a proof term skeleton 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 invertibility, we identify all of them). Then, given a proof $t : S$, applying the proof synthesis method to $\sigma(t)$ (by theorem 1) gives $\Gamma', \mathcal{L} \vdash_r t' : S'$ where $\sigma(t) = \sigma(t')$, $\sigma(S) = \sigma(S')$ and $\mathcal{L}$ is a well formed logical context. The proof term contains holes corresponding to logical proofs to be proved by hand. These holes are called existential variables in the logical context.

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

Let us explain this result. The inversion method defined by the proof of theorem 1 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: For φ a derivation $\Gamma \vdash_r t : S$, let $\sigma(\varphi)$ denote the corresponding derivation for σ, that is $\sigma(\Gamma) \vdash \sigma(t) \in \sigma(S)$.

Proposition 7. 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 the one of the inversion theorem. Each step shows that the constructed logical lemmas are provable. □

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

Example

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

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

An unprovable lemma $(0 < 0)$ is then generated. A program that would generate only provable 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 satisfy to validate its specification. This approach is both a method of verifying programs and a method of synthetically describing proofs.

4. A heuristic method

Weak extracted programs are not very natural. When trying to use them in practice we find that they still require too many type annotations, so we develop heuristics to deal with programs that have even less information than our intermediate language. However, the programmer knows that if he puts in enough types, he can always obtain a program in this intermediate language, which therefore plays a critical role in the definition of the language. It would be nice to consider more natural programs, that is, programs with fewer 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, Par95b, Par95a].

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 keeping a certain notion of completeness. We describe some heuristics and optimizations. We refer the reader to [Par93, Par95a] for further details.

A notion of typing is always necessary since we want to be able to check if the program and its specification are coherent (for this, we need to compare the type of the program and the specification). This typing is a little different from $\in$. We define in definition 17 $\in_2$ which replaces (2) and (3) by (4) in order to consider only pure ${F_\omega}^{Ind}$ programs (not containing logical informations). This new rule allows to go back to a typing corresponding to F_ω.

Definition 17. Typing on ${F_\omega}^{Ind}$ programs:

$$\frac{\Gamma \vdash T' \in_2 s \quad \Gamma \vdash t \in_2 T \quad \varepsilon(T) =_\beta \varepsilon(T')}{\Gamma \vdash t \in_2 T'} \quad s \in S \tag{4}$$

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 do not give all the details in this paper, rather we give an idea of the problems 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 and we choose the unifier which abstracts most variables. In practice, it seems to be a good choice.

4.1.1. Application case

Let us give an example for a heuristic in an application.
Consider the specification $\forall n.\exists m.(S\ m) = n \vee n = m = 0$ where $x = y = z$ is a notation for $x = y \wedge y = z$. 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 $(P : nat \rightarrow Set)(P\ 0) \rightarrow ((n : nat)(P\ n) \rightarrow (P\ (S\ n))) \rightarrow (n : nat)(P\ n)$. Note that in the previous program, `nat_rec` is a program variable whose type is $(P : Set)P \rightarrow (nat \rightarrow P \rightarrow P) \rightarrow nat \rightarrow P$. The proof term to be retrieved is (the `\/` notation is the *Coq* notation for $\vee$ that is *sumor*):

```
[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 the application case (the abstraction which precedes the application is trivial). We look for the predicate whose extraction is *nat*. `nat_rec` and its proof type are known. The predicate P to be determined has type $nat \rightarrow Set$. 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 not 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 binds the greatest number of variables, in this case the first one. Two subgoals are generated:

$$\left\{\begin{array}{l} \exists m.(S\ m) = 0 \vee 0 = m = 0 \\ \forall n.\exists m.((S\ m) = n \vee n = m = 0) \\ \quad \rightarrow \exists m.(S\ m) = (S\ n) \vee (S\ n) = m = 0 \end{array}\right.$$

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)
```

The method can be iterated and the complete proof term can be retrieved except the parts `P1` and `P2`. Two logical lemmas corresponding to `P1` and `P2` remain:

$$\left\{\begin{array}{l} (S\ 0) = 0 \vee 0 = 0 = 0 \\ (S\ y) = (S\ y) \vee (S\ y) = y = 0 \end{array}\right.$$

The principal heuristic of the applicative case consists in looking for the good predicates which are arguments of the application. Let us suppose the specification of the applicative part of the example is a little bit more complicated than in the previous example. If it has the form $(\vec{l} : \vec{L})S$, the problem is to decide what these logical products $(\vec{l})$ correspond to: either they are integrated in the specification (the proof type of the head of the application begins with logical products) or they are logical hypotheses that have to be added in the context (the proof term corresponding to the application begins with logical λ-abstractions).

The heuristic is to introduce all the logical hypotheses that cannot be used in the following. Two cases are distinguished: either the head of the type of the application is bound or not. In the first case, the proof term is generally a predicate. No decision is uniformly better. The heuristic which seems to be the best in practice consists in introducing the hypotheses on which the head of the type of the application does not depend. This is exactly the case of an elimination (we show an example in the following). In the second case, all the logical hypotheses are introduced.

4.1.2. Elimination case

Consider again the proof of a division algorithm. The global specification is $\forall b, a.(b > 0) \to \exists q.\exists r.((S\ n) = b * q + r) \land (b > r)$. The corresponding program is:

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

Let us see the sub-specification $(b > r) \to (n = b * q + r) \to \exists q.\exists r.((S\ n) = b * q + r) \land (b > r)$ and the sub-program:

```
<nat*nat>Match (inf b (S r)) with
          ((S q),O)
          (q,(S r))
end
```

To look for the elimination predicate, we use the same principle as in the previous case. We look for a predicate on booleans, since we want to apply an elimination on booleans *bool* of type $(P : bool \to Set)(P\ true) \to (P\ false) \to (x : bool)(P\ x)$.

But, the specification does not depend on $(inf\ b\ (S\ r))$ on which the elimination is applied. The unique unifier between $(P\ b)$ and the specification is
$P = [x : bool](b > r) \to (n = b * q + r) \to \exists q.\exists r.((S\ n) = b * q + r) \land (b > r)$
It is meaningless since x does not appear anywhere in the body of P.

We have then to introduce an artificial dependence. Then, if Sp is the specification, t the term on which the elimination is applied and T_t its type, Sp is transformed into $(t = t) \to Sp$ by eliminating only the second occurrence of t. The chosen unifier is then $P = [x : T_t](t = x) \to Sp$, and in our example:

$$\begin{aligned} P \quad = \quad & [x : bool]((b < (S\ r)) = x) \to (b > r) \to (n = b * q + r) \\ & \to \exists q.\exists r.((S\ n) = b * q + r) \land (b > r) \end{aligned}$$

Two subgoals are generated and associated to two subprograms `((S q),0)` and `(q,(S r))`:

$$\left\{\begin{array}{l} ((inf\ b\ (S\ r)) = true) \rightarrow (b > r) \rightarrow (n = b * q + r) \\ \quad \rightarrow \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r) \\ ((inf\ b\ (S\ r)) = false) \rightarrow (b > r) \rightarrow (n = b * q + r) \\ \quad \rightarrow \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r) \end{array}\right.$$

In this case too, we have to decide what to do with logical products at the head of the elimination specification. Indeed, do they belong to the elimination predicate or not ? Identical decisions concerning applications are taken. We introduce hypotheses which should not be modified, that is, hypotheses that do not depend on the term on which the elimination is applied. Once more, this heuristic is only motivated by the practical experience.

4.1.3. Elimination at the head of an application

We just give here an example of a particular problematic case. When the head of the application is an elimination, it is not possible to retrieve the elimination predicate (there are too many possibilities). Then, for this particular case, we hope to have a special treatment in order to go back to the previous case of elimination.

For this, we use generalization. This corresponds to applying the following inference rule:

$$\frac{\Gamma \vdash \mathsf{Elim}(e, Q)\{f_1| \ldots |f_n\} : (x : A)B \quad \Gamma \vdash a : A}{\Gamma \vdash (\mathsf{Elim}(e, Q)\{f_1| \ldots |f_n\}\ a) : B[x \leftarrow a]}$$

Given the program $(\mathsf{Elim}(e, Q)\{f_1| \ldots |f_n\}\ a)$ and the specification $B[x \leftarrow a]$, we generalize the specification with respect to the argument a and the elimination (and only the elimination) is then associated to it. Then, it becomes the previous elimination case. This method is a heuristic since there are different methods for generalizing a term.

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 information to the program. These annotations have to be taken into account by the method in order to make good decisions.

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

Definition 18. Annotations:

$$\frac{\Gamma \vdash p \in_2 S \quad \varepsilon(S) = \varepsilon(S')}{\Gamma \vdash (p :: S) \in_2 S'}$$

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 more precise specifications. Their role is 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 allow logical λ-abstractions. This corresponds to the already existing rule of $\in_2$:

$$\frac{\Gamma[x:L] \vdash p \in_2 S}{\Gamma \vdash [x:L]p \in_2 (x:L)S} \text{ if } L : \mathit{Type}$$

These logical abstractions can also be used ad-hoc. We saw previously that, in some cases, logical hypotheses have to be introduced. The decision is heuristic. It is then possible to force some hypotheses to be introduced by using logical abstractions: if introductions are not done and if the user wishes to introduce them (for any reason), it is possible to use the logical abstractions.

Examples

1. Consider again the division algorithm. The intermediate specification is $(b > r) \to (n = b * q + r) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$ and the subprogram:

```
<nat*nat>Match (inf b (S r)) with
           ((S q),O)
           (q,(S r))
end
```

   In this program, it is assumed that the specification of `inf` is known. If not, `inf` is just a boolean function. Neither the proof term of `inf` nor the elimination predicate can be retrieved.
   An annotation can then be used to give the specification of `inf` explicitly inside the program, which becomes:

```
<nat*nat>Match (inf b (S r)) :: {(b<=(S r))} + {(b>(S r))} with
           ((S q),O)
           (q,(S r))
end
```

2. To give a more complicated example, let us take the complete division program and annotate it too much (just to see exactly what an annotation can be, even if in this case this is not necessary at all).
   The notation `{x:A & (P x)}` is similar to `{x:A | (P x)}` but with $P : A \to Set$.

```
[b,a:nat]<nat*nat>Match a with
   (O,O) :: {q:nat & {r:nat | (O=b*q+r)/\(b>r)}}
   ([n:nat][h:nat*nat]<nat*nat>Match h with
        [q,r:nat]
          <nat*nat>Match (inf b (S r))
                          :: {(b<=(S r))} + {(b>(S r))} with
              ((S q),O)
              (q,(S r))
       end end) :: (n:nat){q:nat & {r:nat | (n=b*q+r)/\(b>r)}}
                   ->{q:nat & {r:nat | ((S n)=b*q+r)/\(b>r)}}
end :: {q:nat & {r:nat | (a=b*q+r)/\(b>r)}} .
```

4.3. Validity and Completeness

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

1. The method fails.
2. The method generates a set of logical lemmas that are not provable.
3. The method generates a set of provable logical lemmas.

In the first two cases, either the program is incorrect, or there is not enough logical information in the program. It might be necessary to add annotations in the program.

In the third case, theorem 1 ensures that the inversion of the weak extraction generates a partial proof of the initial specification, and 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 information. 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.

Observation 1. 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 7. □

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, corresponding to recursive structures in a program. This is related to the problem of loop invariants search in the Hoare's logic [Hoa69]. We compare these notions in section 5.

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 further and further from the proof. If we take this criterion into account, the heuristics to retrieve a proof term skeleton become much harder. A possible optimization of the strong extraction procedure consists in distinguishing particular types that we call singleton types. Such an optimization suppresses constructors 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, based on a well-founded induction principle. Extracted programs often contain trivial expressions such as `if b then true else false`, which can be replaced by the more natural program `b`. This implies that heuristics have to be able to retrieve the underlying structure of proof even if this is not the same in the program.

We do not elaborate this part but one can refer to [Par93, Par95a] for an explanation of such optimizations.

5. Related Work

Our method can be compared with methods based on Hoare's logic. Indeed, Hoare's logic proves that programs meet specifications. The structure of the program is analyzed and sub-specifications corresponding to subprograms are generated until axioms are reached. Our idea is exactly the same, with Hoare's axioms corresponding 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 give invariants explicitly. In our method, it is necessary to add annotations. The two methods differ in the framework in which they are defined. Hoare's logic considers imperative programs and specifications are expressed in first order predicate calculus. We consider functional programs and specifications are expressed 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, by analyzing the program and the postcondition and building the weakest precondition. Our method is similar since we build a proof term skeleton, not exactly for the initial specification but for a specification that contains less logical information. This can be considered as a construction of the "weakest specification".

Our work is motivated by the observation that users generally know the algorithm they want to prove correct. Similar motivations underly the work of [BM92] and [Pol94] where a proof and a program are constructed 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, using methods different from ours.

The work of [Pol94] consists in building the program and its correctness proof at the same time, using distinct languages for programs ($\lambda\omega_s$) and proofs ($\lambda\omega_p$). The main difference with our work is that these systems are weaker than the Calculus of Inductive Constructions (they are close to the F_ω system).

Similarly, in [BM92], a notion of deliverable is introduced. A deliverable is a pair (program, correctness proof). The programming language is the simply-typed λ-calculus and the proof language is the Extended Calculus of Constructions [Luo90, LP92]. The idea is the same that [Pol94] to construct proof and program at the same time but the framework different.

In Nuprl [Con86], one can prove directly that a program realizes a specification. The logic allows the user to hide the logical information and include as much computational information as needed (see [How93] for examples).

6. 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 invertibility gives an algorithm for reconstructing a proof skeleton 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 reconstructed deterministically. 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 based on the same idea. The considered programs are ${F_\omega}^{Ind}$ programs. Heuristics are needed to retrieve intermediate sub-specifications by unification. Annotations can be added in ${F_\omega}^{Ind}$ programs to explicitly give sub-specifications. Then, the heuristic method is complete for sufficiently annotated programs. Optimizations can be introduced to consider more natural programs. The method presented in this paper is both a method of verifying programs and a method of synthetically describing proofs. Indeed, weak extracted programs can be seen as proof descriptions.

This method has been implemented as a tactic in the *Coq* system. The description of this tactic can be found in [Par93, Par95a] and a simple proof example using this tactic can be found in [Par97]. A library of examples has been developed with optimizations on the input language for programs. These examples are not trivial: they include algorithms on graphs, trees and lists including different sorting algorithms. We can use the ProPre tactic of [MS92] which has been integrated into *Coq* to define functions via equations. Such definitions are transformed into primitive recursive definitions which can be used as input for our tactic. This allows writing input programs for our tactic more easily. Currently, this is possible but only for a restricted number of functions. We hope the number of functions easily expressible with ProPre will increase and will allow us to use it in a more general way. Moreover, we hope to be able to write programs in a more natural formalism than ${F_\omega}^{Ind}$.

Finally, other methods of proof synthesis from programs could perhaps be developed. It should be possible to work with the program f and with a required proof of the form $\forall x.(P\ x) \rightarrow (Q\ x\ (f\ x))$. This formula means that f realizes $\forall x.(P\ x) \rightarrow \exists y.(Q\ x\ y)$. The corresponding technique should not be very different from ours but one can hope to prove more things since a proof of "f realizes B" can sometimes be done even 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, and to the anonymous referees for many useful comments.

References

[Bar] H. Barendregt. Lambda Calculi with Types. Handbook of Logic in Computer Science, Volume 2.

[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. INRIA technical report 0177, july 1995.

[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.

[Coq89] T. Coquand. Meta-mathematical investigations of a Calculus of Constructions. in [Hue89], 1989.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.R. Hindley, editor, *To H.B.Curry: Essays on Combinatory Logic , lambda-calculus and formalism.* Seldin, J.P., 1980.

[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 formulae-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.

[How93] D. Howe. Reasoning About Functional Programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, 1993.

[Hue89] G. Huet. The Calculus of Constructions, Documentation and users guide, Version 4.10. Technical report, INRIA, 1989.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System : User's Manual. Technical Report 92-211, LFCS, May 1992.

[Luo90] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

[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.

[Par95a] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* PhD thesis, Ecole Normale Supérieure de Lyon, January 1995.

[Par95b] C. Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics for Programs Constructions'95*, volume 947 of *LNCS*, 1995.

[Par97] C. Parent-Vigouroux. A proof example of a division algorithm with the Program tactic in Coq. In *Formal Aspects of Computing* 9(E) , ppabc-xyz (1997). Also available via the journal web site.

[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.