J. Symbolic Computation (1993) **15**, 607–640

Synthesis of ML programs in the system Coq

CHRISTINE PAULIN-MOHRING† AND BENJAMIN WERNER‡

† *LIP-IMAG URA CNRS 1398, ENS Lyon, France*
‡ *INRIA Rocquencourt, France*

(Received 11 March 1993)

The system Coq (Dowek et al., 1991) is an environment for proof development based on the Calculus of Constructions (Coquand, 1985) (Coquand and Huet, 1985) enhanced with inductive definitions (Coquand and Paulin-Mohring, 1990). From a constructive proof formalized in Coq, one extracts a functional program which can be compiled and executed in ML. This paper describes how to obtain ML programs from proofs in Coq. The methods are illustrated with the example of a propositional tautology checker. We study the specification of the problem, the development of the proof and the extraction of the executable ML program. Part of the example is the development of a normalization function for IF-expressions, whose termination has been studied in several formalisms (Leszczylowski, 1981) (Paulson, 1986) (Dybjer, 1990). We show that the total program using primitive recursive functionals obtained out of a structural proof of termination leads to an (at first) surprisingly efficient algorithm. We explain also how to introduce a fixpoint and get the usual recursive program. Optimizations which are necessary in order to obtain efficient programs from proofs will be explained. We also justify the properties of the final ML program with respect to the initial specification.

1. Introduction

1.1. Proofs as programs

The paradigm for the development of certified programs in the system Coq comes from Heyting's interpretation of propositions. A proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ should give a method to transform any object i combined with a proof of $P(i)$ into an object o together with a proof of $Q(i,o)$. We can give a concrete realization of this method as a program. In particular an intuitionistic proof formalized in natural deduction style can directly be interpreted in a well-chosen functional programming language.

Following these ideas, Martin-Löf introduced an intuitionistic theory of types (Martin-Löf, 1984). It contains a programming language, a logic to reason about programs, and also the possibility to get the program underlying any proof, as is explained in (Nordström, Petersson, and Smith, 1990). The system Coq follows the same ideas but involves a different theory including a polymorphic programming language and higher-order logic.

† This research was partly supported by ESPRIT Basic Research Action "TYPES" and by the french government Programme de Recherche Coordonnées "Programmation Avancée et Outils pour l'Intelligence Artificielle" .

0747–7171/93/5–6607 + 34 $08.00/0

The main problem with this methodology is obtaining efficient, realistic programs out of proofs. We list a few problems:

To each proof corresponds a particular program. A problem is to control the algorithm from the proof and to be sure that no extra computation is introduced.
From pure proofs, we get strongly terminating programs. We would like to introduce fixpoints in the programs, and still be able to say something about the termination of the program.
We want to efficiently execute the extracted programs and to be able to integrate them in a general-purpose language.

We present program extraction as it is implemented in the system Coq. This is an attempt to get realistic programs using the functional interpretation of intuitionistic proofs. More generally, our aim here is to shed some light on the fine interactions between proofs and programs as they do appear in practice, illustrating our experience on a concrete example and in a real implementation.

We should also say a word about our target language: out of formal Coq proofs, we will generate ML programs. ML is a family of functional typed languages, enjoying limited polymorphism thus enabling type inference. We refer to (Weis et al., 1990) (Milner, Tofte and Harper, 1990) (Gordon, Milner and Wadsworth, 1979). We chose ML because of its clean semantics and the structural similarity between ML programs and Coq proofs. Most ML implementations (SML, CAML) follow *strict* or *eager* evaluation; a lot of progress has however been made in the technology of *lazy* evaluation (LML). Our system allows extraction towards both families; this difference is discussed later on.

1.2. Plan

The paper is organized as follows. The remaining part of this section is devoted to the description of the tautology-checker. In the second section we explain how to specify the problem. In the third section we present the development of the three parts of the tautology-checker. In the fourth part we study the terms extracted from the proofs, show how to execute them and state their properties. The last part before the conclusion is devoted to a brief description of the system Coq.

1.3. A propositional tautology checker

Throughout the paper, we shall study the example of a propositional tautology checker. This example was first introduced by Boyer and Moore (Boyer and Moore, 1979). It contains a function for normalization of IF-expressions, whose termination was also studied in the framework of LCF (Leszczylowski, 1981) and Martin-Löf's intuitionistic theory of types (Dybjer, 1990) (Paulson, 1986).

The problem is to decide whether a propositional formula is a tautology (is true under all truth assignments). Boyer and Moore proposed the following algorithm.

They introduce the notion of IF-expressions. An *IF-expression* is built from the atomic formulæ (`Tr`, `Fa` and propositional variables) with a ternary operator (`If`). The value of an *IF-formula* (`If` M P Q) is the value of P if the value of M is True and the value of Q otherwise. The first action of the algorithm is to replace a propositional

formula (expressed with usual connectors for disjunction, implication, negation,...) by an equivalent IF-expression.

An IF-expression is said to be *normal* if it is either an atomic formula or an `If`-formula (`If` X P Q) with X a propositional variable and P and Q two normal IF-expressions. The second pass of the algorithm is to replace any IF-expression by an equivalent normal IF-expression.

The normalization function is defined by recursion. Atomic formulæ are normal. To normalize an `If`-formula (`If` M P Q), with M an atomic formula, it is sufficient to normalize P (if M = `Tr`) or Q (if M = `Fa`) or both (if M is a propositional variable). To normalize (`If` (`If` M P Q) R S) the algorithm recursively normalizes (`If` M (`If` P R S) (`If` Q R S)). It is easy to check that (`If` (`If` M P Q) R S) and (`If` M (`If` P R S) (`If` Q R S)) are equivalent, and thus the main problem is to prove the termination of this function.

Deciding whether a normal IF-expression is a tautology is easy. Boyer and Moore introduce the notion of partial assignment. A *partial assignment* is a function from propositional variables to truth values with a finite domain. Given a partial assignment α, we introduce the notion of *partial tautology* with respect to α. A formula is a partial tautology with respect to α if it is true for all assignments which coincide with α on its domain. A formula is a tautology if and only if it is a partial tautology with respect to the partial assignment with an empty domain. Now we may recursively decide whether a normal IF-expression M is a partial tautology with respect to a given partial assignment α:

If M is an atomic formula, the only interesting case is when it is a propositional variable X: it is a partial tautology if and only if $\alpha(X)$ is defined and equal to the value True (in the other case we may find α' extending α such that $\alpha'(X)$ is the value False).

If M is a normal `If`-formula (`If` X P Q) then so are P and Q and X is a propositional variable. If $\alpha(X)$ is defined and equal to the value True (resp. the value False) then M is a partial tautology with respect to α if and only if, P (resp. Q) is a partial tautology with respect to α. Let α_T (resp. α_F) be the extension of α with the value true (resp. false) assigned to X. If $\alpha(X)$ is not defined then M is a partial tautology respectively to α if and only if P is a partial tautology respectively to α_T and Q is a partial tautology respectively to α_F.

This example illustrates the main problems of program development in the system Coq. In particular we shall study how to develop partial or recursive programs. We show that using the term extracted from a termination proof for the normalization function leads to an efficient program. This contradicts a frequent belief in the inefficiency of programs extracted from proofs.

2. Specification

The first step in the program development is to state the specification. In general there is not one good specification but several possibilities. A very poor language is theoretically sufficient for most the computing problems, but our experience is that some methods lead to smoother proofs in Coq, and we would like to illustrate this point in this paper.

An important question is the representation of data. We have to choose the mathematical structure (for instance propositional variables are coded with numbers, formulæ are seen as a free algebra, ...). These choices are sometimes arbitrary in this example. We would like to emphasize that Coq provides a natural way to represent most of the mathematical structures involved in program development using functionality and inductive

definitions. The general method for specification will be to use this power in order to get high-level representations of specifications and avoid tedious encodings.

2.1. The Calculus of Constructions with Inductive Definitions

2.1.1. Description

Formally the Calculus of Constructions is a typed lambda-calculus. It defines a language of types and of terms; its rules simply describe how one can derive the judgement that a term t is correctly typed of type T in an environment which assigns types to the free variables of t (we shall write $t : T$).

Types may of course represent data structures like natural numbers or functions spaces. The point is, however, that the type algebra is rich enough to represent also propositions via the Curry-Howard isomorphism. Thus, a typing judgement "t has type T" can be interpreted both as "t is a program of type T" or "t is a proof of the proposition T".

Furthermore types themselves may be constructed using a small typed programming language. There is a typing judgement $T : A$ which says that T is a well-formed type (proposition) or type scheme with A describing its arity. To say that T is a type is to assert the judgement $T : \mathit{Set}$. For each type T we can, for instance, build the type of lists of elements of T. To do so, we build a type-scheme *list* which is a function from types to types and so is formally a term of type $\mathit{Set} \rightarrow \mathit{Set}$. Further basic examples involving the pure Calculus of Constructions can be found for instance in (Coquand and Huet, 1985) (Coquand and Huet, 1987).

In Coq the original Calculus of Constructions is extended with a mechanism for primitive inductive definitions. We can declare a positive inductive type like the type of natural numbers by giving the expected types of its constructors. The precise mechanisms are described in (Coquand and Paulin-Mohring, 1990) (Paulin-Mohring, 1992) but the examples given in this paper should be understandable by themselves and illustrate the various uses of this powerful scheme.

In Coq there are two possible sorts for the type of propositions namely *Set* and *Prop*. Using these two sorts, the user can specify in the development of the proof, which part has to be interpreted as a program and which is only a comment asserting that some conditions are satisfied by the program. More explanation about the distinction between *Prop* and *Set* can be found in section 2.4.2.

We should also mention that the meta-theory (especially strong normalization and hence consistency) of the entire formal system corresponding to Coq is still under development (Werner, 1992).

2.1.2. Notations

We briefly give the notations for terms in the system Coq. The typewriter font will be used for commands which can be interpreted by the system, comments are enclosed in brackets: `(* .. *)`.

`[x:A]B`	Abstraction of `B` w.r.t. the variable `x` of type `A` (usually written as $\lambda x^A.B$).
`(A B)`	Application of `A` to `B`.
`(x:A)B`	Product of `B` w.r.t. the variable `x` of type `A` (i.e. $\forall x : A.B$).
`A->B`	Type of functions from `A` to `B` (corresponds to `(x:A)B` when `x` does not occur in `B`).
`Prop`	Constant corresponding to the type of propositions.
`Set`	Constant corresponding to the type of specifications and data types.

Application associates to the left and arrow to the right. Other notations will be introduced and explained in the following.

2.2. Using the underlying programming language

The arithmetic is a framework to reason about properties of integers. Theoretically we only need a few primitive functions and predicates on natural numbers to express properties of programs. But often for convenience, we assume that we have function symbols for all primitive recursive functions together with axioms corresponding to their properties.

In the system Coq we have a more powerful language to express the set of objects we are reasoning about, almost a general programming language.

2.2.1. Defining data types

First of all we may reason about natural numbers, but also about other algebras of terms, like lists or trees. Following the ML-style of programming we do not encode an algebra using (for instance) lists but use a direct representation with concrete types and constructors. Examples of such concrete definitions of data types are the type of boolean values, of propositional formulæ and of IF-expressions:

```
Inductive Set bool = true : bool | false : bool.

Inductive Set PropForm =
          Fvar : nat -> PropForm
        | Or   : PropForm -> PropForm -> PropForm
        | And  : PropForm -> PropForm -> PropForm
        | Impl : PropForm -> PropForm -> PropForm
        | Neg  : PropForm -> PropForm
        | Bot  : PropForm.

Inductive Set IFExpr =
          Var  : nat -> IFExpr
        | Tr   : IFExpr
        | Fa   : IFExpr
        | If   : IFExpr -> IFExpr -> IFExpr -> IFExpr.
```

2.2.2. Functionality and Polymorphism

To represent an assignment we have to associate a boolean value with each propositional variable; this can be done using the type of functions from natural numbers to

boolean values. We shall introduce a constant `Assign`, the type of assignments, defined as `nat->bool`. This illustrates a feature of the language: the possibility to reason about functions or to parameterize a problem with respect to a functional value. In the same way, types or programs can be abstracted with respect to type variables. This feature allows the definition of parameterized types (lists of elements of type α) and the development of algorithms working uniformly regardless of the type of their input elements (e.g. sorting functions).

Here, a partial assignment is a partial function with finite domain from variables to boolean values. It will be represented by a pair of integer lists. Each list corresponds to a finite set of variables. One list corresponds to variables assigned to true and the other one to variables assigned to false. We shall need an extra logical hypothesis (the two lists do not contain the same elements) to make sure that it corresponds in an unambiguous way to a function. To emphasize the fact that these two lists are constraints for variables, we introduce the constant `Constraints` for the type of lists of integers.

2.2.3. Defining primitive functions

We may define on these new concrete types all the primitive recursive functions but also more complicated ones (following a primitive recursive scheme but on functionals). The properties corresponding to the definition of these functions are automatically known by the system. For instance on natural numbers, given two functions g and h we may define a new function ϕ following the primitive recursive scheme:

$$\phi(0) \equiv g \qquad \phi(n+1) \equiv h(n, \phi(n))$$

The terms $\phi(0)$ and g (resp. $\phi(n+1)$ and $h(n, \phi(n))$) will be intentionally equal in the system (we may replace one by the other everywhere). Assume that `g` and `h` are two terms of Coq which represent g and h. The type of `g` will be `C` and the type of `h` will be `nat->C->C`. The syntax to designate ϕ in the system Coq is:

```
Definition phi : nat -> C = [n:nat](<C>Match n with g h)
```

More generally let `t` be a term which has type an inductive type `I` and `C` be a type, the expression `<C>Match t with` is a well-formed term of the system Coq. Its type depends on the structure of the inductive type `I`. It expects as many arguments as there are constructors of `I`, the type of each argument depends on the type of the corresponding constructor of the inductive definition. In case the inductive definition is not recursive it corresponds to a matching operation like in ML. In the case of a recursive inductive definition like the natural numbers, it implements the analogue of the operator R for primitive recursive definitions in Gödel's system T (Girard, Lafont and Taylor, 1989).

We give examples of definitions using the primitive recursive scheme. We indicate using comments `(* .. *)`, for each argument of the `<C>Match t with` term, the corresponding constructor. We may define a ternary function `ifb` on booleans such that `(ifb b1 b2 b3)` is `b2` (resp. `b3`) if `b1` is `true` (resp. `false`):

```
Definition ifb : bool -> bool -> bool -> bool
  = [b1,b2,b3:bool](<bool>Match b1 with (* true *) b2 (* false *) b3).
```

Following the same scheme, we may define the boolean functions corresponding to negation (**negb**), conjunction (**andb**), disjunction (**orb**) and implication (**implb**). The definition of the semantics of a propositional formula under some assignment is then defined as a primitive recursive function on the structure of formulæ.

```
Definition Prop_sem : Assign -> PropForm -> bool
 = [A:Assign][F:PropForm]
  (<bool> Match F with
    (* Fvar n *) [n:nat](A n)
    (* Or F G *) [F:PropForm][SF:bool][G:PropForm][SG:bool](orb SF SG)
   (* And F G *) [F:PropForm][SF:bool][G:PropForm][SG:bool](andb SF SG)
  (* Impl F G *) [F:PropForm][SF:bool][G:PropForm][SG:bool](implb SF SG)
     (* Neg F *) [F:PropForm][SF:bool](negb SF)
       (* Bot *) false).
```

Behind this admittedly obscure syntax, one should recognize the definition of a function such that the following equalities hold internally:

```
(Prop_sem A (Fvar n))  =  (A n)
(Prop_sem A (Or F G))  =  (orb (Prop_sem A F) (Prop_sem A G))
                  ...     ...
 (Prop_sem A (Neg F))  =  (negb (Prop_sem A F))
     (Prop_sem A Bot)  =  false
```

Following the same scheme, we define a function `IF_sem` of type `Assign->IFForm->bool` which computes the semantics of an IF-expression under some assignment.

2.3. Defining properties of programs

In the system Coq we may express properties of objects. We show standard ways to define such properties and also how to prove them.

2.3.1. Using predicate calculus

We may define properties just using usual predicate calculus (propositional connectors, quantifiers, equality, natural numbers,...) which are predefined in the system. We shall use the following notations:

`False`	the falsum proposition which admits no closed proof
`True`	a true proposition with exactly one closed proof
`<A>x=y`	equality of x and y of type A
`~A`	negation of A
`A/\B`	conjunction of the properties A and B
`<A>Ex(P)`	existence of x of type A such that (P x) is provable

A propositional formula and an IF-expression are equivalent if their semantics are equal boolean values under any assignment. This equivalence relation is defined in Coq as a term of type `Formula->IFForm->Prop` with:

```
Definition Prop_IF_Equiv : PropForm -> IFExpr -> Prop
  = [F:PropForm][I:IFExpr](A:Assign)<bool>(Prop_sem A F)=(IF_sem A I).
```

An IF-expression is a tautology if it is true under all assignments. We may define this property by:

```
Definition Tautology :  IFForm -> Prop
     = [IF:IFForm](A:Assign)<bool>true=(IF_sem A IF).
```

We may define the predicate *not to be a tautology* as the negation of the previous formula: `[G:IFForm]~(Tautology G)` or more positively as the existence of a refutation:

```
Definition Refutable :  IFForm -> Prop
     = [IF:IFForm]<Assign>Ex([A:Assign]<bool>false=(IF_sem A IF)).
```

In order to prove a property involving primitive recursive functions, one can use logical properties (introduction and elimination of connectors), but in general one needs to reason on the values of the functions. For this, one can use the computation rules. If the argument of the function starts with a constructor then one uses one of the definitional equalities of the function. Otherwise, reasoning by induction on the argument introduces the various cases corresponding to each constructor for which the computation rule applies. For instance, to prove for an arbitrary assignment `A` and a formula `F`:

```
<bool>true=(Prop_sem A (Or F (Neg F)))
```

we may first compute in the goal the value of `(Prop_sem A (Or F (Neg F)))`, leading to an equivalent goal:

```
<bool>true=(orb (Prop_sem A F) (negb (Prop_sem A F)))
```

No more computation can be done, but we may reason by cases on the boolean value of `(Prop_sem A F)`; this gives us two subgoals:

```
<bool>true=(orb true (negb true))
<bool>true=(orb false (negb false))
```

Now the computation rule applied to the boolean functions `orb` and `negb` reduces both goals to the trivial logical one: `<bool>true=true`.

2.3.2. Using inductive definitions

Another powerful way to define a property of objects is to use inductive definitions of predicates or relations. It corresponds to the mathematical view of defining the smallest relation which satisfies some closure properties and to the Prolog definition of a relation with definite clauses.

For instance we may define the predicate *to be normal* for an IF-expression as the smallest predicate which is true for all atomic formulæ and true for (`If` X P Q) provided that X is a propositional variable and P and Q are normal. This is done in the system Coq by the declaration:

```
Inductive Definition Normal : IFExpr -> Prop
 = NVar : (n:nat)(Normal (Var n))
 | NTr  : (Normal Tr)
 | NFa  : (Normal Fa)
 | NIf  : (n:nat)(F,G:IFExpr)
           (Normal F)->(Normal G)->(Normal (If (Var n) F G)).
```

With this definition we get automatically proofs corresponding to the constructors, stating that the predicate `Normal` satisfies the closure properties.

We also can use the fact that it is the smallest relation in order to derive smooth proofs of logical propositions like: `(F:IFExpr)(Normal F) -> (P F)`, where `P` is an arbitrary predicate (of type `IFExpr->Prop`). Instead of doing a proof by induction on `F` we may use an induction on the structure of the proof of `(Normal F)`. This will generate exactly the four interesting cases with the right induction hypotheses, namely:

```
(n:nat)(P (Var n))
(P Tr)
(P Fa)
(n:nat)(F,G:IFExpr)
  (Normal F)->(P F)->(Normal G)->(P G)->(P (If (Var n) F G))
```

A difficulty appears when we want to prove a property `(Normal T)->Q` where `T` is not a variable and the property is true because of the very structure of `T`. Such an example is the theorem `(Normal (If Tr F G))->False`.

In that case we have to find a property `P` of type `IFExpr->Prop` such that `(P T)->Q` is provable as well as `(F:IFExpr)(Normal F)->(P F)`. One possibility is to define `P` as the term: `[F:IFExpr](<IFExpr>T=F)->G`. We may also use the powerful feature of definition of a property by cases on the structure of an IF-expression to construct a property `P` such that `(P T)` and `Q` are two convertible propositions. The syntax of such a definition follows the pattern of a primitive recursive definition of an object. Let us give an example.

We may define a primitive recursive property `Norminv` such that `(Norminv T)` will be the proposition `True` if `T` is atomic, will be the conjunction of the two propositions `(Normal F)` and `(Normal G)` if `T` is `(If (Var n) F G)` and the proposition `False` otherwise:

```
Definition Norminv = [F:IFExpr]
 (<Prop> Match F with
    (* Var n *) [n:nat]True
       (* Tr *) True
       (* Fa *) True
  (* If G H I*) [G:IFExpr][PG:Prop][H:IFExpr][PH:Prop][I:IFExpr][PI:Prop]
              (<Prop> Match G with
              (* Var m *) [m:nat]((Normal H)/\(Normal I))
                 (* Tr *) False
                 (* Fa *) False
       (* If G' H' I' *) [G':IFExpr][PG':Prop][H':IFExpr][PH':Prop]
                         [I':IFExpr][PI':Prop]False)).
```

It is a simple induction to prove: `(F:IFExpr)(Normal F)->(Norminv F)`, and particular instantiations (because `(Norminv (If Tr F G))` is convertible with `False` for instance) of this theorem give the expected properties:

```
(Normal (If (Var n) F G))->((Normal F)/\(Normal G))
(F,G:IFExpr)~(Normal (If Tr F G))
...
```

The definition of a proposition by cases on the values of an element of an inductive type is called *strong elimination*. It corresponds to admit the term `<A>Match t with` with `A` an arity (`Prop`, `nat->Prop`,...). Its use admits some restrictions for consistency reasons. The inductive type should be "small". This means that the arguments of the constructors may be proofs or programs but not propositions or types. Allowing strong elimination in the case of a non-small inductive definition leads to paradoxes. The other restriction is that we may use strong elimination to define a proposition or a logical relation but not the type of a program (typically `A` cannot be `Set`) in which case we could have the type of the output of the program depending on the values of input, thus leading to non ML-typable programs. This more problematic extension is under study (Werner, 1992).

The relation `Assigned` of type (`nat->Constraints->Prop`) is also inductively defined. The property `(Assigned n l)` is true if the number `n` appears in the list `l`. It is defined by:

```
Inductive Definition Assigned [n:nat] : Constraints -> Prop
 = Assign_hd : (PA:Constraints)(Assigned n (Cons n PA))
 | Assign_tl : (m:nat)(PA:Constraints)
               (Assigned n PA)->(Assigned n (Cons m PA)).
```

We remark that, in this definition, the two clauses do not correspond to different constructors of `Constraints`. Actually the proof of `(Assign n l)`, if it exists, is not uniquely determined by the structure of `l` and `n` (for instance if `n` appears twice in `l`); also in order to exhibit a proof of `(Assign n l)`, we shall need an extra assumption, namely the decidability of equality over natural numbers.

2.4. Programs specification

In the previous sections we have briefly discussed how to write programs in Coq and how to define properties over these programs. Another way to combine proofs and programs is to introduce logical informations inside the programs. That way, Coq could be considered as a programming language in which comments are analyzed and checked.

2.4.1. Specification of functions and predicates

A specification will be a type of program which contains a logical part. For instance we want to specify a program which associates to a propositional formula `F`, an equivalent IF-expression. The type of IF-expressions which are equivalent to a propositional formula F will be represented as:

```
{I:IFExpr|(Prop_IF_Equiv F I)}.
```

This type is defined as an inductive definition with only one constructor of type :

```
(I:IFExpr)(Prop_IF_Equiv F I)->{I:IFExpr|(Prop_IF_Equiv F I)}
```

A closed normal term of this type is a pair consisting of a term `I` of type `IFExpr` as a first component and a proof of `(Prop_IF_Equiv F I)` as a second component. It is the analogue of an existential definition except that the second component is marked to be without computational meaning (just a comment) and will be discarded during the program extraction process.

The specification of the program itself will be:

```
(F:PropForm){I:IFExpr|(Prop_IF_Equiv F I)}
```

Finding an element in this type will give us a certified program.

We are interested in programs for computing recursive functions and predicates. A recursive predicate can be represented by its characteristic function but we have in Coq a more direct specification. Given two properties `A` and `B`, we want the program to answer `true` (resp. `false`) if some property `A` (resp. `B`) is satisfied. We use the specification: `{A}+{B}`. An element of this type is either the value `true` combined with a proof of `A` or the value `false` combined with a proof of `B`.

A program may be valid only on a subset of the inputs; and thus, in general, the specification will contain a precondition. Typically the tautology-checker for normal IF-expressions will be specified by:

```
(F:IFExpr)(Normal F)->{(Tautology F)}+{(Refutable F)}
```

2.4.2. Informative contents

A specification is a type of program with some extra logical information in it. This logical information is like a comment and will not appear in the extracted program. In Coq, the duality between `Set` and `Prop` embodies this distinction between types of program development and types of logical proofs in comments. If A is of type `Prop` then A is a logical proposition and elements of A are proofs which are ignored from the computational point of view. If A is of type `Set` then A is a specification and elements of A are program developments from which Coq extracts certified programs with respect to A.

An alternative specification of a tautology-checker would be that it answers "It is a tautology" or "It is refutable" and in that second case also exhibit an assignment under which the proposition is false. The specification of such a program would be:

```
(I:IFExpr)(Normal I)->{(Tautology I)}+{A:Assign|<bool>false=(IF_sem A I)}
```

Note that even if the final specification of the problem is just to answer "yes" or "no", the more informative specification which explicitly builds the assignment in the program could be necessary as an intermediate specification for a particular algorithm.

When writing a specification, the choice of the connectives determines the input and outputs of the underlying program. For instance, we do not want the proof of `(Normal I)` to be an input of the program, so the predicate `Normal` will be declared as a logical one

(of type `IFExpr->Prop`). The property "I is refutable" can be either defined as a logical proposition `<Assign>Ex([A:Assign]<bool>false=(IF_sem A I)):Prop` i.e. there exists an assignment but we do not care about its value, or as a specification written `{A:Assign|<bool>false=(IF_sem A I)}:Set` which explicitly builds the assignment.

3. Development of programs

We present in this part the development of certified programs in Coq. The structure of the proof is related to the structure of the underlying program, so we use the program as a guide for conducting the proof.

3.1. A PRIMITIVE RECURSIVE PROOF

A very simple example of program development is the one which transforms a propositional formula into an equivalent IF-expression. The transformation is a simple induction over the structure of propositional formulæ. Its specification is:

```
(F:PropForm){I:IFExpr|(Prop_IF_Equiv F I)}.
```

Doing a proof by induction on `F` generates six subgoals corresponding to the various cases for `F`.

```
{I:IFExpr|(Prop_IF_Equiv (Fvar n) I)}
{I:IFExpr|(Prop_IF_Equiv (Or G H) I)}
{I:IFExpr|(Prop_IF_Equiv (And G H) I)}
{I:IFExpr|(Prop_IF_Equiv (Neg G) I)}
{I:IFExpr|(Prop_IF_Equiv Bot I)}
```

For the first subgoal, we have just to provide the solution `(Var n)`. It generates a subgoal `(Prop_IF_Equiv (Fvar n) (Var n))`, which is solved using computation rules.

For the second subgoal, we get as hypotheses:

```
{I:IFExpr|(Prop_IF_Equiv G I)}
{I:IFExpr|(Prop_IF_Equiv H I)}
```

These two induction hypotheses assert the existence of IF-expressions equivalent to the propositional formulae `G` and `H`. Performing eliminations on these two hypotheses gives us access to the IF-expressions `G'` and `H'` plus the proofs of `(Prop_IF_Equiv H H')` and `(Prop_IF_Equiv G G')`. Now we provide the solution for `(Or G H)`, namely `(If G' Tr H')`. This generates the subgoal:

```
(Prop_IF_Equiv (Or G H) (If G' Tr H'))
```

which is solved using computation rules plus the hypotheses on `G'` and `H'`.

The other cases are analogous. In this example, the relationship between the proof and the program is trivial.

3.2. A PARTIAL PROOF : THE TAUTOLOGY CHECKER FOR NORMAL IF-EXPRESSIONS

We now study the program which decides whether a normal IF-expression is a tautology.

3.2.1. USING SUBPROGRAMS

This program is not really difficult to justify as soon as we identify the specifications of each important subpart. The main loop decides, given a partial assignment (two coherent sets of constraints), if a formula is a tautology under these constraints. The specification of this part is:

```
(F:IFExpr)(Normal F)->
     (ltrue,lfalse:Constraints)(Coherent ltrue lfalse)
     ->({Part_Tauto F ltrue lfalse}+{Part_Refut F ltrue lfalse}).
```

The program uses also a subroutine which decides whether a variable is assigned in a constraint. From the algorithmic point of view this just amounts to check whether the variable appears in the list; from the specification point of view this corresponds to a proof of:

```
(n:nat)(PA:Constraints)({(Assigned n PA)}+{~(Assigned n PA)})
```

A conditional expression "if n appears in l then ... else ..." in the program corresponds to the elimination of the property `{(Assigned n l)}+{~(Assigned n l)}` in the proof.

3.2.2. DOING THE RIGHT INDUCTION

The program corresponds to a restricted case of structural induction: because the IF-expression is assumed to be normal, we would like to build a solution only for atomic formulæ and IF-formulæ (`If` X G H) assuming X is a variable and G and H are normal formulæ for which we know the result. We have seen that this kind of proof actually corresponds to an induction on the structure of the proof of `(Normal F)`. But in our interpretation we do not want the proof of `(Normal F)` to be an input of the program but only a comment, and thus it is not possible to build a program by cases depending on this object. More technically, the system will fail to do an elimination of a proof of `(Normal F)` which has type `Prop` to prove a goal which has type `Set`.

In this particular case however, the structure of a proof of `(Normal F)` depends only on the structure of `F`. We recognize this point because the conclusion of each of the clauses defining `Normal` corresponds to exactly one different constructor for IF-expressions and each hypothesis of these clauses can recursively be found from `F`. This makes it possible to build a proof of the following induction principle for normal IF-expressions:

```
(P:IFExpr->Set)
   ((n:nat)(P (Var n)))
     ->(P Tr)
     ->(P Fa)
     ->((n:nat)(F,G:IFExpr)
           (Normal F)->(P F)->(Normal G)->(P G)->(P (If (Var n) F G)))
```

```
->(I:IFExpr)(Normal I)->(P I).
```

This proof is done using structural induction on I, which is possible since I of type IFExpr is an input of the program. We also use negative properties of the predicate Normal like: ~(Normal (If Tr F G)). From the computational point of view, what we are doing is justifying a partial pattern matching:

```
let rec Fnormal = function
    (Var n)          -> ...
  | Tr               -> ...
  | Fa               ->
  | (If (Var n) F G) -> ... (Fnormal F) ... (Fnormal G) ;;
```

by a compilation of this structure into a language with only total pattern matching. In the unreachable cases, we are in a context where the absurd proposition False is provable. In that case any specification can be fulfilled. This corresponds to the hypothesis except whose type is (C:Set)False->C. From the computational point of view this proof should generate an arbitrary element in any type, which will never be evaluated if the program is used according to its specification. We see this object as an exception corresponding to a failure "out of specification" of the program.

In this precise case, in order to avoid the problem of justifying partial pattern matching, we could also introduce a concrete type of normal IF-expressions and use this type for the output of the normalization function and the input of the tautology-checker. Our partial match on IF-expression would become a total match on the type of normal IF-expression. This corresponds to an alternative design solution for this algorithm.

3.3. A RECURSIVE PROOF

The specification of the normalization function is:

```
(F:IFExpr){G:IFExpr | (Normal G) & (Equiv F G)}
```

The expected program as given in Boyer and Moore's book is the following one:

```
let rec norm = function
          (Var n)              -> Var n
        | Tr                   -> Tr
        | Fa                   -> Fa
        | (If Tr F G)          -> norm F
        | (If Fa F G)          -> norm G
        | (If (Var n) F G)     -> If (Var n) (norm F) (norm G)
        | (If (If X Y Z) F G)  -> norm (If X (If Y F G) (If Z F G))
```

The main problem is to justify the inductive structure of the program. The matching part is just the combination of two embedded basic matchings; the problem comes from the recursive call with argument (If X (If Y F G) (If Z F G)) whose termination is not trivial.

We can reflect the structure of the program in the following induction principle which will be called *(IP)*:

```
(P:IFExpr->Set)
 (P Tr) -> (P Fa) -> ((n:nat)(P (Var n)))
 -> ((Y,Z:IFExpr)(P Y)->(P (If Tr Y Z)))
 -> ((Y,Z:IFExpr)(P Z)->(P (If Fa Y Z)))
 -> ((n:nat)(Y,Z:IFExpr)(P Y)->(P Z)->(P (If (Var n) Y Z)))
 -> ((X,Y,Z,T,U:IFExpr)
     (P (If X (If T Y Z) (If U Y Z)))->(P (If (If X T U) Y Z)))
 -> (X:IFExpr)(P X).
```

In order to prove the specification, we have first to prove *(IP)*; then we apply it to the specification. This generates the seven cases corresponding to the different branches of the function with exactly the right hypotheses. The only problem is the justification of the induction principle. Various proofs of this principle have been proposed in the literature. We shall study two of them.

3.3.1. Structural induction

The shortest and most elegant proof goes back to structural induction over IF-expressions. We assume:

```
(P Tr)
(P Fa)
(n:nat)(P (Var n))
(Y,Z:IFExpr)(P Y)->(P (If Tr Y Z))
(Y,Z:IFExpr)(P Z)->(P (If Fa Y Z))
(n:nat)(Y,Z:IFExpr)(P Y)->(P Z)->(P (If (Var n) Y Z))
(X,Y,Z,T,U:IFExpr)
   (P (If X (If T Y Z) (If U Y Z)))->(P (If (If X T U) Y Z))
```

We have to prove `(X:IFExpr)(P X)`. After one step of structural induction, the three cases where X is an atomic formula are trivially solved and it remains to prove:

```
(X:IFExpr)(P X)->(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If X Y Z))
```

The trick is to prove by induction on `X` the property:

```
(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If X Y Z))
```

Once more the three cases where X is an atomic formula are trivial, the interesting one is when `X` is `(If F G H)` in which case the induction hypotheses holds for `F`, `G` and `H`.

```
(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If F Y Z))
(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If G Y Z))
(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If H Y Z))
```

and `(Y:IFExpr)(P Y)->(Z:IFExpr)(P Z)->(P (If (If F G H) Y Z))` is to be proved. Using the last clause of the induction principle we wanted to prove, the conclusion `(P (If (If F G H) Y Z))` reduces to `(P (If F (If G Y Z) (If H Y Z)))`. Now using the induction hypothesis for `F`, where `Y` is instantiated by `(If G Y Z)` and `Z` is instantiated by `(If H Y Z)`, we have to prove `(P (If G Y Z))` and `(P (If H Y Z))`. These properties are easy consequences of the hypotheses for `G` and `H`.

This gives a fine proof of our induction principle, however the resulting program which is explicited in 4.2.2 does not involve any general fixpoint, but only structural induction. Thus it does not correspond to the original algorithm. Actually, the program extracted from this proof is more efficient than the original general recursive program of Boyer and Moore. We shall analyze the different programs in the next section. Usually one complains that constructive proofs give only primitive recursive algorithms which internally contain the termination information and that we need frameworks where we may directly derive general recursive algorithms. In this example, we "discovered" an interesting algorithm for the problem by an analysis of a constructive proof of termination for the recursive program. It should be noticed that this nice proof of termination appears already in the paper by Leszczylowski (Leszczylowski, 1981), but the algorithmic meaning of this proof, although it was mentioned in Paulson's paper (Paulson, 1986), was not recognized as a real optimization of the computational problem.

3.3.2. Well-founded induction

We do not pretend, however, to generalize the example above. Even if in this very case a proof of termination gives a good algorithm, general recursion in the extracted program is still very often needed in practice. This cannot be done in the pure system, but we can add an axiom for well-founded induction which will be realized using a fixpoint. That the resulting program is safe with respect to its specification will be explained in the next section.

To introduce a fixpoint in a program, we use the fact that the principle of well-founded induction is realizable, as it is explained in (Paulin-Mohring, 1989b) (Paulin-Mohring, 1989c). The property:

```
(A:Set)(R:A->A->Prop)(well_founded A R)->
     (P:A->Set)((x:A)((y:A)(R y x)->(P y))->(P x))->(a:A)(P a).
```

in which `(well_founded A R)` is a logical property †, is interpreted by the following ML program:

```
let WF_rec F = wrec where rec wrec x = F x wrec;;
```

We shall give an idea of the proof that `WF_rec` is a correct program with respect to the well-founded induction principle. A precise proof requires the complete definition of the realizability interpretation and is presented in (Paulin-Mohring, 1989b). Let `F` be a correct program for the specification `(x:A)((y:A)(R y x)->(P y))->(P x)` and `a` be a correct program for `A`; we have to check that `(WF_rec F a)` is correct with respect to `(P a)`.

† The property `well_founded` can be defined like Nordström's *Acc* type (Nordström, 1988) using the scheme of inductive definitions or an impredicative encoding.

This proof is done using well-founded induction on a. We can assume that (WF_rec F b) is correct with respect to (P b) for all b such that (R b a). Now (WF_rec F a) is equal to (F a (WF_rec F)). The correctness of F insures that (F a (WF_rec F)) is correct for (P a) if a is correct for A and for any y such that (R y a) we have (WF_rec F y) correct for (P y) but it is true by the induction hypothesis.

Whatever the proof of termination for the relation R is, the extracted program will be a fixpoint for F. From the logical point of view, each time we invoke (F y) in the definition of (F x), we have to justify as a comment that y is less than x w.r.t. R.

If we want to justify the recursive program, we have to find a relation R, which is well-founded and such that:

```
(R Y (If Tr Y Z))                    (R Y (If (Var n) Y Z))
(R Z (If Fa Y Z))                    (R Z (If (Var n) Y Z))
(R (If X (If T Y Z) (If U Y Z)) (If (If X T U) Y Z))
```

In the literature, we find a measure credited to R. Shostak:

$$m(\texttt{Tr}) = m(\texttt{Fa}) = m(\texttt{Var } n) = 1 \qquad m(\texttt{If } X\ Y\ Z) = m(X) \times (1 + m(Y) + m(Z))$$

We may take for the definition of (R Y Z) the fact that m(Y) is strictly less than m(Z), use the well-foundness of the order on natural numbers and check the expected properties for R. But, as mentioned by L. Paulson, it is more convenient to define in the system a relation which only satisfies the expected properties. This is especially easy in Coq using either impredicativity, or inductive definitions, or the definition of a proposition by cases on the structure of an element in an inductive type. In this example we choose the last alternative:

```
Definition norm_order : IFExpr->IFExpr->Prop =
 [X,Y:IFExpr]
 (<Prop>Match Y with
   (* Var n *) [n:nat]False
      (* Tr *) False
      (* Fa *) False
(* If T U V *) [T:IFExpr][PT:Prop][U:IFExpr][PU:Prop][V:IFExpr][PV:Prop]
        (<Prop>Match T with
    (* Var n *) [n:nat](<IFExpr>U=X) \/ (<IFExpr>V=X)
       (* Tr *) <IFExpr>U=X
       (* Fa *) <IFExpr>V=X
 (* If P Q R *) [P:IFExpr][PP:Prop][Q:IFExpr][PQ:Prop][R:IFExpr][PR:Prop]
                  <IFExpr>(If P (If Q U V) (If R U V))=X)).
```

In order to prove that this relation is well-founded we can use the induction principle *(IP)* but for logical propositions. Then we may justify the induction *(IP)* on specifications used in the program just by combining the well-founded induction and matching on IF-expressions.

As will be explained later, it gives us exactly the original recursive program. Such a development can be done systematically from the program.

3.4. Discussion

This study points out a methodology for program development. First of all, the recursive structure of the program can be easily represented by a higher-order proposition. Even if the second order quantification is not necessary (we could have proven the induction principle instantiated with the specification P we are interested in) it helps for an abstract view of the proof.

After that, we have to provide a justification of this principle which will provide the control structure of the program. We shall generally use for this a combination of pattern-matching and well-founded induction. But sometimes, like in this example, there exists a direct proof of this principle and it gives a better algorithmical solution to the problem. A very simple example where this phenomenon already appears is the development of the program computing Fibonnacci's numbers. We can naively write the function:

```
let rec fib = function
    O -> (S O) | (S O) -> (S O) | (S (S x)) -> (fib x)+(fib (S x));;
```

The corresponding induction principle is:

```
(P:nat->Set)
(P O)->(P (S O))->((n:nat)(P n)->(P (S n))->(P (S (S n))))->(n:nat)(P n)
```

The natural way to justify this induction is to prove with structural induction the property: `(n:nat)((P n)*(P (S n)))`. This proof will exactly correspond to a program which stores the two last computed values of the function. So in this case also it corresponds to a truly better program.

In the paradigm of development of programs as proofs, we have generally to keep in mind the program as a guide for the proof. But we have seen that proofs can also suggest interesting algorithms. Our system provides both possibilities, doing original constructive proofs or introducing fixpoints using well-founded relations. Executing the extracted programs allows an easy comparison of the performances of the different solutions.

4. Proof Interpretation and Execution

This section is devoted to the (automatic) synthesis of actual ML programs out of proofs, and the justification of their correctness.

Rather than relating directly Coq proofs to ML programs, we will start by interpreting these proofs by programs typable in a *fragment* of Coq called F_ω^{idt}. This will make it easier to relate the behavior of the programs to the original specification. From there we will see how to generate the ML code. We will then show how well-founded recursion and partial functions correspond to extensions of the programming language. Finally we will state the correctness properties of the extracted programs, according to whether these extensions are used or not.

4.1. An Interpretation in F_ω^{idt}

The key to program extraction from proofs is *realizability*: this formal notion relates the behavior of some functional program to a given proposition of the considered logical

system. For example, a function f is said to realize the proposition $\forall x.I(x) \Rightarrow \exists y.P(x,y)$ if given any x_0 verifying $I(x_0)$, we have a proof of $P(x_0, f(x_0))$. The reader of earlier sections will have understood that what we need here is a notion of realizability for Coq, and an algorithm to automatically synthesize a function realizing a proposition P out of any proof of P. Such a notion of realizability and the corresponding extraction algorithm is precisely described in (Paulin-Mohring, 1989a) (Paulin-Mohring, 1989b) for the pure Calculus of Constructions which corresponds to Coq without inductive definitions.

4.1.1. Extraction

The extraction operation consists in taking a proof of a specification (i.e. an object whose type is of type *Set*) and erasing its non-computational part, thus producing the *underlying program*. The essential ideas of the extraction operation are intuitively simple:

> All proof terms occurring in types or propositions are erased (we get rid of the so-called *dependent types*).
> The proof terms corresponding to non-computational parts are erased.

This is reflected by the types of the terms before (resp. after) the extraction is performed. Let $S : Set$ be a specification and $t : S$ a proof of it, we may call $\overline{t}$ the extracted program and $\mathcal{E}(S)$ its type: $\overline{t} : \mathcal{E}(S)$. Moreover, to express that $\overline{t}$ actually meets its specification S, we may define the realizability predicate $\mathcal{R}(S,x)$ such that $\mathcal{R}(S,\overline{t})$ is verified.

Let us try to illustrate this by defining the extraction function over t:

t	$\overline{t}$
$[x : S]t_1$	$[\overline{x} : \mathcal{E}(S)]\overline{t_1}$
$[x : P]t_1$	$\overline{t_1}$
$t_1\ t_2$	$\overline{t_1}\ \overline{t_2}$
$t_1\ p$	$\overline{t_1}$
x	$\overline{x}$
$Constr\{i,S\}$	$Constr\{i,\mathcal{E}(S)\}$
$< S > Match\ t\ with\ t_1 \ldots t_n$	$< \mathcal{E}(S) > Match\ \overline{t}\ with\ \overline{t_1} \ldots \overline{t_n}$

t and t_i denote computational (proof) terms, p denotes non-computational ones. $\overline{x}$ is a "fresh" variable, i.e. which has not been used before. S stands for any computational type (of type Set) and P for a non-computational type (of type Prop). $Constr\{i,S\}$ just stands for the i^{th} constructor of the inductive type S.

4.1.2. Realisability

Defining the $\mathcal{E}$ and $\mathcal{R}$ functions in full detail requires case analysis over the whole algebra of terms. This is not our aim here. However, intuitive understanding may be given by considering the quantification, which is the key case.

S	$\mathcal{E}(S)$	$\mathcal{R}(S,t)$
$P \rightarrow S_1$	$\mathcal{E}(S_1)$	$P \rightarrow \mathcal{R}(S_1,t)$
$(x : S_1)S_2$	$\mathcal{E}(S_1) \rightarrow \mathcal{E}(S_2)$	$(\overline{x} : \mathcal{E}(S_1))\mathcal{R}(S_1,\overline{x}) \rightarrow \mathcal{R}(S_2, t\ \overline{x})$

As expected, the inductive types are mapped to inductive types, the extraction function

being applied to each constructor's type. For example, the existence property {x:A|P(x)} is primitively defined inductively:

```
Inductive Set sig [A:Set;P:A->Prop]  =  exist : (a:A)(P a)->(sig A P).
```

And the notation {x:A|(P x)} is just smoother syntax for (sig A P). The single constructor, which precisely corresponds to the existential quantifier introduction scheme, has a computational argument (a) and a non-computational one (the proof of (P a)). Therefore the type $\mathcal{E}$(sig) resulting from the extraction is defined as:

```
Inductive Set sig' [A:Set] =  exist' : A -> (sig' A).
```

On the other hand, an object of type sig' is said to realize its specification if:

- It reduces to a constructor of sig' (in this case reduces to some (exist' a)).
- The arguments of this constructor do realize their specification (here P).

So $\mathcal{R}$(sig) would be defined as:

```
Inductive Definition Rsig [EA:Set;RA:EA->Prop;P:EA->Prop]:(sig' EA)->Prop
    =  Rexist : (a:EA)(RA a)->(P a)->(Rsig EA RA P (exist' EA a)).
```

We can remark that (sig' A) is the type of singletons of objects of type A. Therefore it is obviously isomorphic to A. More generally, each time $\mathcal{E}$(T) is an inductive type with only one constructor of arity one, it may be identified with the type of the argument of the constructor. This corresponds to a simplification often done internally by ML compilers. It is performed in Coq during the optimization phase in order to gain readability in the extracted code.

Another example is the informative disjunction. We previously used the syntax {A}+{B}. This is just an abbreviation for (sumbool A B) which is defined as:

```
Inductive Set sumbool [A,B:Prop] = left  : A -> (sumbool A B)
                                 | right : B -> (sumbool A B).
```

the result of the extraction is just isomorphic to ML booleans:

```
Inductive Set sumbool' = left' : sumbool' | right' : sumbool'.
```

and the realizability predicate states that the two cases actually correspond to A and B:

```
Inductive Definition Rsumbool [A,B:Prop] : sumbool' -> Prop =
   Rleft  : A -> (Rsumbool A B left')
 | Rright : B -> (Rsumbool A B right').
```

The case of pure data types like nat is simple. The definition being:

```
Inductive Set nat = O : nat | S : nat -> nat
```

the type itself remains unchanged by the extraction function and a term is said to realize nat if it satisfies the predicate :

```
Inductive Definition Rnat : nat-> Prop =
  R0 : (Rnat 0)
| RS : (n:nat)(Rnat n) -> (Rnat (S n))
```

As we shall explain later on, the fact that there exists a closed proof of `(Rnat n)` implies that `n` reduces to some `(S (S ...(S 0)...))`.

These notions of extraction and realizability interpret proofs of specifications done in Coq as programs typable in the fragment of Coq where dependent types and non-computational terms are prohibited. This fragment is precisely Girard's system F_ω extended with inductive types, also called F_ω^{idt}. In this fragment, we can still define the same algorithms as in Coq, but we lose the ability to state properties of these programs.

The internal structure of the extracted terms which are a mixture of λ-terms, constructors and elimination schemes for inductive types, suggests the next step, which is execution in ML.

4.2. ML as an Execution Model

The extraction operation described above produces two kinds of objects: the terms extracted from proofs, and types extracted from specifications. The terms are very similar to ML programs, since they are combinations of λ-calculus (abstraction, variable, application) and inductive types (constructors, recursion schemes). Hence, it is almost immediate to translate the obtained terms into ML syntax with concrete types and pattern-matching. Again, examples illustrate the transformation. Let us just outline a few points:

In F_ω^{idt}, recursion appears only in the recursion operators associated with each inductive type; it is combined with pattern-matching. So a little work has to be done to translate these operators into ML, combining pattern-matching and the let rec operator.

A restriction however has to be done over the (computational) inductive types used during the proof/program development: some types have no counterpart in ML. This happens when a quantification over types (explicit polymorphism) occurs in the type of a constructor. For example:

```
Inductive Set anything = dummy : (A:Set) A -> anything.
```

It is well-known that the type system of ML is weaker than systems like F_ω which enjoy explicit polymorphism. Therefore it might be necessary to "switch off" the type inference process when compiling extracted programs. This is not dangerous, in the sense that these programs have already been type-checked. Most of the "realistic" programs we have built in Coq were however ML-typable.

Let us scan through the Coq definitions given above, and see what ML code does arise out of them. Coq allows extraction towards different ML dialects (see sect. 5); here we give the resulting CAML concrete syntax. To obtain the code below some simple optimizations were performed after the extraction. They are described in sect. 4.5.

4.2.1. A simple type

Data types like the ones defined in sect. 2.2.1 are straightforward. For example:

```
type IFExpr =
        Var of nat
      | Tr
      | Fa
      | If of IFExpr * IFExpr * IFExpr;;
```

Note that, since CAML constructors admit at most one argument, the type of `If` had to be un-curryfied. In the case of LML for instance, this is not necessary, as LML admits curryfied constructors.

4.2.2. A SIMPLE FUNCTION

Each time one proves a lemma or a theorem in Coq and names it, one actually defines a constant which is added to the Coq environment. Justifying the general structure of this environment is kept during the program extraction; in other words, a theorem or Coq constant is translated to an ML constant. This is quite natural, as both structures correspond to *closed terms* appearing during the development. Also, the result looks quite natural. Of course, non-computational propositions or proofs like `Normal`, `Norminv` or `Tautology` have no-counterpart in the ML program.

In 3.1, we described the translation of propositional formulas to IF-expressions. It corresponded to a primitive recursive proof of the following statement:
`(F:PropForm){I:IFExpr|(Prop_IF_Equiv F I)}`.
Applying the $\mathcal{E}$ function to it, we get the type of the function extracted from the proof:
`PropForm -> IFExpr`

The proof consisted in a case analysis over the formula `F` with authorized use of the induction hypothesis over the subformulas of `F`. This structure appears clearly in the extracted program:

```
let rec PropForm p =
  match p with
     Fvar v        -> Var v
   | Or (p1,p2)    -> If (PropForm p1,Tr,PropForm p2)
   | And (p1,p2)   -> If (PropForm p1,PropForm p2,Fa)
   | Impl (p1,p2)  -> If (PropForm p1,PropForm p2,Tr)
   | Neg p1        -> If (PropForm p1,Fa,Tr)
   | Bot           -> Fa;;
```

A more subtle and insightful case is the normalization function extracted from the "clever" proof, which does not use general recursion:

```
let rec Norm_prog1 F = match F with
     Var v         -> Var v
   | Tr            -> Tr
   | Fa            -> Fa
   | If (p,p1,p2) ->
       (let rec Norm_if F = match F with
              Var v            -> (fun p1 N1 p2 N2 -> If (Var v,N1,N2))
            | Tr               -> (fun p1 N1 p2 N2 -> N1)
```

```
    | Fa                -> (fun p1 N1 p2 N2 -> N2)
    | If (p',p1',p2')   ->
        (fun p1 N1 p2 N2 ->
           Norm_if p' (If (p1',p1,p2)) (Norm_if p1' p1 N1 p2 N2)
                      (If (p2',p1,p2)) (Norm_if p2' p1 N1 p2 N2))
in Norm_if p p1 (Norm_prog1 p1) p2 (Norm_prog1 p2));;
```

It appears clearly that only structural recursion is needed in both uses of `let rec`, thus implying termination. It is not just as simple to understand why this function is more efficient than the original and straightforward normalization function of Boyer and Moore (sect. 3.3). Thinking in terms of evaluation strategies, it appears that Boyer and Moore's function does *head reduction*: it looks for the first (outermost) redex, reduces it, and iterates. Thus it duplicates some of the expressions, to normalize which makes the complexity exponential. Consider for instance: `(If (If X Y Z) P Q)` where `X`, `Y` and `Z` are variables and `P` and `Q` are some big expressions. The simple `norm` starts by rewriting it to `(If X (If Y P Q) (If Z P Q))`, and therefore it afterwards has to normalize `P` and `Q` twice later on.

In comparison, `Norm_prog1` will start by normalizing `P` and `Q` to `NP` and `NQ` and then evaluates
`Norm_if X (If Y P Q) (Norm_if Y P NP Q NQ) (If Y P Q) (Norm_if Z P NP Q NQ)`
which reduces to `(If X (If Y NP NQ) (If Z NP NQ))`.

There are two relevant remarks to make here:

> The `Norm_prog1` function scans through its argument in such a way that it terminates in linear time, although the resulting normal expression may be exponentially larger than the argument.
> It appears that the second and fourth arguments of the `Norm_if` function are superfluous. So this is also a good example to illustrate that in some cases inefficiencies may remain in the extracted program. One may hope to find smart program optimizations which would get rid of more and more of these redundancies.

About the first remark and the global complexity of the program, note that checking if a normal expression is a tautology is done in linear time w.r.t. the size of the normal expression (and therefore in exponential time w.r.t. the original IF-expression). So the global checking algorithm is of course still exponential, but only during the second phase†.

4.3. Generalized Recursion and Except

Describing program development, we saw that it was possible to build partial functions and to use general well-founded recursion. To obtain the expected programs, we have to change slightly the extraction definition. This section focuses on these features.

4.3.1. A partial function: Except

Let us take a look at the function checking if normal IF-expressions are tautologies. It is a partial function in the sense that it is only supposed to be applied to *normal*

† The problem of tautology-checking is co-NP-complete (all known algorithms are exponential).

IF-expressions. In sect. 3.2 we saw how this appeared in a specific induction principle which corresponded to partial pattern matching. However, and for obvious reasons, only complete pattern matching is primitively available in Coq. Therefore, in order to prove our partial induction principle for normal IF-expressions, we have to argue that some branches of the matching-tree will never be explored. From the logical point of view, we are able to prove the absurd Proposition `False` inside these branches. Therefore, we will assume the following axiom: `except : (C:Set)False -> C`. It says that if we are in an incoherent situation where falsum can be proved, we can also realize any specification.

It is easy to prove that this axiom is consistent with the theory. However, when we perform the extraction, we have to translate it to some piece of ML code. As this code should never be executed and ought to inhabit any type, it is natural to use the feature of ML exceptions. For instance, here is the actual code extracted from the "partial" induction principle proved in sect. 3.2:

```
let Normal_rec Fv Ctr Cfl Cif p =
 let rec Norm_aux p = match p with
     Var v          -> Fv v
   | Tr             -> Ctr
   | Fa             -> Cfl
   | If (p1,p2,p3) -> match p1 with
         Var v              -> Cif v p2 p3 (Norm_aux p2) (Norm_aux p3)
       | Tr                 -> fail
       | Fa                 -> fail
       | If (p1',p2',p3') -> fail
  in  Norm_aux p;;
```

Of course, to get satisfactory code, the definition of `Normal_rec` will have to be expanded. This happens in the optimization phase described in 4.5.

Realizing the `except` axiom with an ML exception or failure is particularly well-suited in this example. The general case is more problematic since it involves a structure (exception) which is not part of the functional language we used to define the realizability. For example the behavior of a program with exceptions depends on the execution strategy. We discuss and justify this in sect. 4.4.

4.3.2. Extracting general recursion

If a recursive function can be proved terminating in Coq, the corresponding ML function can be extracted in the way described in 3.3.2: the axiom `WF_rec` is actually mapped to the let rec construction. The termination proof, being entirely in the non-computational fragment of Coq, does not appear in the extracted program. Hence, a little program transformation expanding the corresponding piece of code produces the expected program. For instance the original normalization function of Boyer and Moore is obtained exactly as given in 3.3.2. Again, this corresponds to an extension of the realizability interpretation and we will have to justify it.

4.4. Correctness and Properties of the ML Program

What is finally expected is that the computational behavior of the ML program respects its specification as stated in Coq. For reasons of space, it is impossible to go into a full description of the syntax and operational semantics of the extraction language. We will however see that the realizability property allows us to obtain most of the expected results without getting too involved with the very details of ML.

Coq allows extraction towards strict and lazy dialects of ML. If we use general recursion or partial functions, this may change the properties of the extracted programs. Let us discuss the different cases. For the moment we do not consider program transformation and optimizations, as they can be justified separately.

4.4.1. Pure Programs

By *pure* we mean programs using neither partial functions nor general recursion as described above. In other words, programs extracted from proofs which do not assume the `WF_rec` and `except` axioms. In this precise case, the obtained ML code is therefore entirely derived from F_ω^{idt} terms. Therefore, we may use the essential property of F_ω^{idt}, *strong normalization*, in order to ensure termination of the programs:

Theorem 4.1. *Any program extracted from a closed Coq proof (i.e. assuming no axioms) terminates under Call-by-Value and Call-by-Need evaluation strategies. Moreover the obtained value meets the specification corresponding to the original proof.*

Proof. For programs without exceptions and other side effects, ML evaluation can be interpreted as a sequence of reduction (rewritings) over the program. Each of these reductions can be mapped back to a reduction in the corresponding F_ω^{idt} term, except the let rec expansion. Yet, as the original proof did not use well-founded recursion, all the let rec f = T1 in T2 constructions come from a recursive elimination scheme over some inductive type and therefore T1 is a pattern matching. A little analysis of the programs shows that (in both evaluation strategies) the let rec expansion is always followed by the matching resolution, and the sequence of these two rewritings exactly corresponds to the reduction of the elimination scheme in F_ω^{idt}. Therefore strong normalization of F_ω^{idt} implies termination of the evaluation strategy. The fact that the evaluation follows F_ω^{idt} reductions also justifies the soundness w.r.t. the original specification. □

4.4.2. Impure Programs and Call-by-Need Evaluation – Lazy ML

Neither exceptions, nor the fixpoint are typable in F_ω^{idt}. Therefore, when extracting programs containing exceptions or the well-founded recursion operator, we cannot use the strong normalization argument to ensure termination. However the realizability still holds; i.e. for any proof M of any specification S, the property $\mathcal{R}(S, \overline{M})$ is verified even if $\overline{M}$ is not typable in F_ω^{idt} anymore. So the idea is now that the realizability property will contain the termination information.

We will now show that the notion of realizability defined above is sufficient to prove that $\overline{M}$ terminates under lazy evaluation, provided some conditions on S.

Definition 4.1. *We will call* CBN evaluation, *any deterministic evaluation strategy*

which terminates for any program M, provided that $M =_\beta V$ for some value V. We say that M admits a value.

It is well-known that compilers like LML (Augustsson and Johnsson, 1989) fit this definition. In this case, as most of the time, a value is either an abstraction $\lambda x.t$ or term $(C\ V_1\ \ldots\ V_n)$ where C is a constructor and the V_i's are values.

There is no general property in Coq asserting that a program admits a value. However saying that a program realizes some type of a well-chosen form implies that it admits a "lazy" value, without changing the definition of $\mathcal{R}$. We shall explain this phenomenon.

REMARK. Usually logicians introduce two different notions of realizability. One called *modified realizability* in which all programs terminate because they are strongly typed. The other one called recursive realizability in which one explicitely proves that each manipulated program admits a value in the current context. To our knowledge Krivine with the system AF_2 (Krivine, 1990) (Krivine and Parigot, 1987) was the first to manipulate a modified notion of realizability in a context of possibly non-terminating programs. The termination condition was included in the specification itself via a proposition which expresses that a program is in a data type†.

The ideas developed in this part are greatly inspired by this work.

THEOREM 4.2. *Let I be an inductive definition of a type. Let p be a program such that there exists a proof of the property "p realizes I" (or $\mathcal{R}(I,p)$) with no hypothesis. Then p is β equivalent to a term $v = (c_i\ t_1\ \ldots t_n)$ where $c_i = Constr\{i, \mathcal{E}(I)\}$ is one of the constructors of the type $\mathcal{E}(I)$.*

PROOF. $\mathcal{R}(I)$, the predicate describing the realisers of I, is an inductive definition. A proof of the property $\mathcal{R}(I,p) \equiv (\mathcal{R}(I)\ p)$ with no logical hypothesis is necessarily of the shape : $(Constr\{i, \mathcal{R}(I)\}\ u_1\ \ldots u_p)$ and has type $\mathcal{R}(I, (Constr\{i, \mathcal{E}(I)\}\ t_1\ \ldots t_n))$ for some terms $t_1 \ldots t_n$. The types $\mathcal{R}(I,p)$ and $\mathcal{R}(I, (Constr\{i, \mathcal{E}(I)\}\ t_1\ \ldots t_n))$ are consequently convertible. We can conclude that p and $(Constr\{i, \mathcal{E}(I)\}\ t_1\ \ldots t_n)$ are convertible terms. □

This allows us to distinguish a first class of specifications whose realizations admit values:

DEFINITION 4.2. *A computational type S is said to have a* non-void quantification domain *if it is of the form $(x : D_1) \ldots (x_n : D_n)I$ where $D_1, \ldots, D_n$ are realizable formulas, I is an inductive type and at least one of the D_i is a computational type.*

We may remark that these conditions are satisfied by the usual program specifications, like $\forall x : D.P(x) \Rightarrow \exists y : D'.Q(x,y)$.

THEOREM 4.3. *If a type S has a non-void quantification domain, then any term M, such that $\mathcal{R}(S,M)$ is provable with no hypothesis admits a value $\lambda x.M'$ under CBN evaluation.*

PROOF. The proof is done by induction on the structure of S. If S is $(x : D)S'$ with D

† The same idea to get programs out of termination proofs was developed independently in (Leivant, 1983) (Leivant, 1990)

realizable, then either D is non computational and there exists a proof with no hypothesis of $\mathcal{R}(S', M)$ so we can apply the induction hypothesis to S' since it admits a non-void quantification domain, or D is computational and there exists a term d and a proof with no hypothesis of $\mathcal{R}(D, d)$. This gives us a proof with no hypothesis of $\mathcal{R}(S', (M\ d))$. Hence, a simple case analysis shows that $(M\ d)$ either reduces to some constructor form $(C\ t_1\ \ldots\ t_n)$ (by the previous lemma) or to some abstraction $\lambda y.M'$ (because of the induction hypothesis for S'). Both cases obviously imply that the term M can be reduced to $\lambda x.M'$ and thus admits a value. □

Finally we have to characterize a class of inductive types admitting values:

DEFINITION 4.3. *We say that a type S is* valuable *if it either admits a non-void quantification domain, or if it is an inductive type such that all the arguments of all its constructors are either of a valuable type, or recursive arguments of type S.*

THEOREM 4.4. *Any term M realizing any valuable type S admits a value.*

PROOF. The case where S has a non-void domain has been treated above. If S is inductive, we reason by induction over the proof of S being valuable. The theorem 4.2 ensures that M reduces to some $(C\ t_1\ \ldots\ t_n)$ and the induction hypothesis says that all the non-recursive arguments of C admit a value. A second induction over the proof of $\mathcal{R}(S, M)$ allows to state that so do the recursive arguments of C, since they realize S. Therefore M reduces to some $(C\ V_1\ \ldots\ V_n)$ with all the V_i's being values. □

To apply this result to the termination of programs containing general recursion and `fail`,we now simply need the following result:

THEOREM 4.5. *The axioms `except` and `WF_rec` are respectively realized by the lazy ML terms `fail` and `WF_real` where `WF_real` is defined as:*

```
let WF_real F =  wrec where rec wrec x = F x wrec.
```

PROOF. The case of `except` is obvious: as `False` is not inhabited, we can ensure for any specification C:

$$\mathcal{R}(\texttt{False} \to C, \texttt{fail}) = \texttt{False} \to \mathcal{R}(C, \texttt{fail}).$$

For `WF_rec` the proof was sketched in 3.3.2. □

4.4.3. IMPURE PROGRAMS AND CALL-BY-VALUE EVALUATION

This case is more problematic, as the notion of realizability defined above is not sufficient to ensure termination, even for trivially inhabited specification. For instance, the following program is extracted from a Coq term of type `nat`, and yet, obviously fails under CBV:

```
let failed_nat = ((fun f -> 0) fail);;
```

In the same way, one may also construct non-terminating terms. Of course, in practice, such programs rarely occur, and when they do, optimizations as described in 4.5 will

generaly make them safe. Yet this is unsatisfactory, as we want formally proved programs. This section is devoted to a solution of this problem.

4.4.4. Recursive Realizability

Non-termination may appear because closed pieces of extracted code do not necessarily come from closed proofs; for instance the `fail` in the term above may correspond to some proof of `False -> nat`, but the fact that we locally assume `False` is not reflected in the program. The right way to solve this problem is therefore to freeze the evaluation of any part of the program which we "are not sure about" by putting an abstraction over it: Consider `[f:False]p`; we should take care of the fact that a logical assumption is made (`False`) which may have consequences on the computational behavior. If we translate it by the program `fun () -> p`, we prevent any looping or undesired exception-raising in the code corresponding to `p`. The crucial point is of course that all ML evaluation strategies exclusively perform *weak reductions*, i.e. never evaluate under an abstraction. In the same sense, `except` should be translated by `fun () -> fail`. Following that idea, we also translate proofs of non-computational propositions to the ML void construct (or `()`). In the literature, the resulting proof/program relation is called "recursive realizability"; it is described for the Calculus of Constructions in (Paulin-Mohring, 1989c). This realizability precisely carries the information necessary for termination in CBV. Here is the new extraction function for terms:

t	$\overline{t}$
$[x:S]t_1$	$[\overline{x}:\mathcal{E}(S)]\overline{t_1}$
$[x:P]t_1$	$[x:\mathbb{I}]\overline{t_1}$
$t_1\ t_2$	$\overline{t_1}\ \overline{t_2}$
$t_1\ p$	$\overline{t_1}\ \Diamond$
x	$\overline{x}$
$Constr\{i,S\}$	$Constr\{i,\mathcal{E}(S)\}$

Where $\mathbb{I}$ is a type of the extraction language whose only inhabitant is $\Diamond$ (thus respectively corresponding to ML's `unit` and `()`).

The aim is to deal with terms of a language which are not always terminating. Therefore we make use of a predicate $t\downarrow$ to state that a term t of the language has a value. We then may reformulate extraction and realizability over types:

S	$\mathcal{E}(S)$	$\mathcal{R}(S,t)$
$P \to S_1$	$\mathbb{I} \to \mathcal{E}(S_1)$	$(t\downarrow) \wedge (\mathcal{R}(P) \to \mathcal{R}(S_1, t\ \Diamond))$
$(x:S_1)S_2$	$\mathcal{E}(S_1) \to \mathcal{E}(S_2)$	$(t\downarrow) \wedge ((\overline{x}:\mathcal{E}(S_1))(\mathcal{R}(S_1,\overline{x}) \to \mathcal{R}(S_2, t\ \overline{x})))$

Of course, the resulting programs are a little longer than with modified realizability. However, a very few optimizations are enough to get reasonable decent code.

We are now in a framework very well-suited for general recursion or partial functions, even with CBV evaluation. However, in order to do a precise correctness proof, we would need to define precisely the property $t\downarrow$ in our logic and relates precisely its semantics to ML CBV operational semantics. Therefore we only conjecture the result and postpone a detailed study for future work.

CONJECTURE 4.1. *Given a Coq proof of any specification S, the program p extracted using recursive realizability terminates and verifies its specifications $\mathcal{R}(S,p)$.*

We shall have to find recursive realizations for the `except` and `WF_rec` axioms. This can easily be done. `except` will be realized by the program `fun () -> fail` and `WF_rec` will be realized by:

```
let WF_real F = wrec where rec wrec x = F x (fun y () -> wrec y);;
```

The extraction algorithm corresponding to recursive realizability is not implemented in the currently distributed implementation of Coq. A new release should however soon include this feature, with the possibility to choose between the two extraction algorithms.

4.5. SOME OPTIMIZATIONS

We mentioned previously that some optimizations were performed on the extracted code. It would certainly be interesting to study how the information given by the proof and the specification could be used to transform the program. Here we are much less ambitious, concentrating simply on the extracted code. Hence, the simple optimizations we perform do not change the algorithm in any way, but help to get rid of some superficial redundancies. In any case, it seems fair to say a word about what code transformations are actually performed in the implementation.

4.5.1. CONSTANT EXPANSION

The main point which makes optimization compulsory, is the presence of recursors over data types in the "raw" extracted code. For example in the case of lists, this corresponds to:

```
let rec list_rec a f l =
  match l with
     nil -> a
   | cons (x,l') -> let l'' = list_rec a f l' in f x l' ll'';;
```

Considering that all pattern matching over lists uses the `list_rec` function, the resulting inefficiencies clearly appear: each time, a recursive exploration of the list is performed, even if only the head of the list is to be calculated. Similarly, in the case of strict evaluation, both branches of an if ...then ...else ... construct would be evaluated. As a consequence, a non-optimized program may have its complexity exponentially increased under CBV evaluation. More generally, even if important theorems almost always correspond to important functions, the presence in the code of some of the constants created during the proof process may be useless or even burdensome.

Therefore, a certain number of constants are expanded: if the definition corresponds to a recursion scheme of course, but also if the definition is very small or obviously corresponds to a non-strict function. If, for the reason above, optimizations are vital for CBV evaluation, and apart from the fact that readable and natural code is morally comforting, these transformations are also useful in the case of lazy ML where in practice they provide a speed improvement of up to a factor of two. Also, having more readable code is often useful for gaining understanding.

4.5.2. Reductions

Expanding constants creates new β-redexes. To be effective the expansion has to be combined with some reduction. This includes β-reduction, but also simplification of pattern-matching in the case where the head of the matched term is a constructor. We have however to check that performing partial evaluation on ML programs does not jeopardize their termination. For CBN this is an immediate consequence of the Church-Rosser property. For CBV, the result is not trivial, even if not surprising. We propose a proof, based on a result of Plotkin's (Plotkin, 1975). For means of place, we here restrict ourselves to pure λ-calculus, since adding constructors, pattern-matching and a `fail` construct does not deeply change the proof. A short review of the basic material: $M \rhd_{ML} V$ means that the λ-term M evaluates to the value V under the Call-by-Value evaluation strategy (corresponding to strict ML operational semantics). We will also write $M \rightarrow_{ML} M'$ to express that M rewrites to M' in one ML evaluation step. Plotkin introduces a restricted form of the β-reduction, named β_v-reduction:

Definition 4.4. *One defines the* β_v-reduction *by:*

$$(\lambda x.M\ N) \rightarrow_{\beta_v} M[x \setminus N] \Leftrightarrow \ N \text{ is a value i.e. of the form } y \text{ or } \lambda y.N'$$

As usual we write $M \rhd_{\beta_v} M'$ *(resp.* $M \rhd_{\beta} M'$*) to express that* M *rewrites to* M' *by* β_v *(resp. usual* β*) reduction of some subterm.*

The essential result we will use here is the following:

Theorem 4.6. *(Plotkin). Let* M *be any lambda-term. The following holds:*

$$\exists V.M \rhd_{ML} V \Leftrightarrow \exists v.M \rhd_{\beta_v} v$$

where v *is either a variable* y *or an abstraction* $\lambda x.N$*.*

This will allow us to prove the soundness of the β-reduction as a program transformation, formally stated as follows:

Theorem 4.7. *Let* M *be any* λ*-term. If* $M \rhd_{ML} V$ *and* $M \rhd_{\beta} M'$*, then* $M' \rhd_{ML} V'$ *(i.e. the evaluation of* M' *terminates), with* $V =_{\beta} V'$*.*

We first remark that if V' exists, the proposition $V =_{\beta} V'$ is an obvious consequence of the Church-Rosser property. So all we have to prove is the existence of V'.

What follows is quite similar to Tait's well-known proof of the Church-Rosser property.

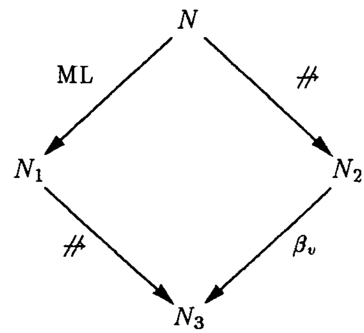
Definition 4.5. *We define the* parallel reduction $\not\Rightarrow$ *by:*

$$\begin{array}{ll}
(\lambda x.M\ N) \not\Rightarrow M'[x \setminus N'] & \text{if } M \not\Rightarrow M' \text{ and } N \not\Rightarrow N' \\
\lambda x.M \not\Rightarrow \lambda x.M' & \text{if } M \not\Rightarrow M' \\
(M\ N) \not\Rightarrow (M'\ N') & \text{if } M \not\Rightarrow M' \text{ and } N \not\Rightarrow N' \\
M \not\Rightarrow M &
\end{array}$$

It is obvious that $M \rhd_{\beta} M'$ implies that $M \not\Rightarrow M'$. Therefore, and applying theorem 4.6, we may reduce the theorem to the following:

LEMMA 4.1. *For any term N, if $N \rightarrow_{ML} N_1$ and $N \not\!\!\!\!\rightarrow\!\!\!\!\!/\!/ N_2$, then there exists a term N_3 such that $N_1 \not\!\!\!\!\rightarrow\!\!\!\!\!/\!/ N_3$ and $N_2 \rhd_{\beta_v} N_3$.*

This may be illustrated by the following diagram:



PROOF. The proof is quite straightforward by induction on N. It uses the substitutivity property (if M and N reduces to M' and N' then $M[x \setminus N]$ reduces to $M'[x \setminus N']$ for both parallel reduction $\not\!\!\!\!\rightarrow\!\!\!\!\!/\!/$ and $\rhd_{\beta_v}$. □

4.5.3. OTHER SIMPLIFICATIONS

Another consequence of the presence of recursors in the originally extracted code, is that each pattern-matching over some recursive type is coupled with a let rec, even if the corresponding code is in fact not recursive. Such unused let rec's are detected and erased.

In the same way, let `A`, `B`, `A'`, `B'` be some non-computational propositions such that we have proofs of `A -> B` and `A' -> B'`. We may easily prove the computational proposition (i.e. specification): `{A}+{B} -> {A'}+{B'}`. However, the resulting program would not be the identity but something like:

```
let trans u = match u with true -> true | false -> false;;
```

so these kinds of trivial matchings are simplified (replaced by the matched element). Another possible case is that all the right branches of the matching are equal, which can be simplified similarly.

Let us also recall that, as already mentioned, inductive types with one constructor of arity one (as the one extracted from `{x:A|P}`) can be suppressed by identification with the type of the constructor's argument (in this case `A`).

All these transformation are obviously sound.

4.5.4. MORE ADVANCED TRANSFORMATIONS

Finally it may happen that the way pattern matchings are embedded within each other prevents from natural simplifications. We propose the following two rewrite rules:

> Code of the form (Match t with C1 → t1 | C2 → t2 | . . . | Cn → tn) u is transformed into: Match t with C1 → (t1 u) | C2 → (t1 u) | . . . Cn → (tn u).

Code of the form
Match (Match t with C1 → t1 | ... | Cn → tn) with D1 → u1 | ... | Dm → um
is transformed into:
Match t with C1 → (Match t1 with D1 → u1 | ...) | ... | Cn → (Match tn with D1 → u1 | ...).

The reader may notice that these transformations are quite similar to the reduction process of IF-expressions as described above. More seriously, they correspond to commutative cuts in natural deduction. The idea is that applying these rules can make redexes appear, thus leading to new simplifications. In practice this actually happened. Care is however due, as in some cases these rewritings may increase tremendously the size of the code. We did not study these transformations in full detail, which would imply for instance stating a Church-Rosser property. Their soundness w.r.t. ML evaluation seems however quite clear.

5. The Implementation

At last, we should say a word about the implementation used to carry out this experiment. Coq (french for rooster) was developed at project FORMEL of INRIA and ENS–Lyon. It is itself implemented in CAML, an ML dialect, also from INRIA. The system is an evolution of a former implementation of the pure Calculus of Constructions. It enjoys top-down proof development in the style of LCF, the user being able to modify the current goal through the use of tactics.

The implementation makes use of various parsing and pretty printing facilities of CAML, especially to produce extracted code for various target languages:

CAML itself, which follows CBV evaluation.
LML, the lazy ML implementation of Göteborg University.
GAML, a more experimental but similar lazy ML developed at INRIA.

The implementation is now available by anonymous ftp at INRIA †. The distribution includes a manual and various examples. Among them is the complete file of the tautology checker, so interested readers may study it in more detail.

All the given examples of this paper are machine-checked and printed out as they appear on the screen. We just did some variable renaming by hand in order to achieve better readability.

6. Conclusion

This paper describes the development and extraction of programs in the system Coq. We emphasize the use of inductive definitions for the representation of data or properties. Our mechanism for inductive definitions allows a natural formalization of computing notions and avoids tedious encoding. Higher-order quantification is used to express and justify recursive structures which appear in programs. Even if not needed theoretically in our examples, this possibility is convenient in a computer assisted development system.

† On ftp.inria.fr (absolute addressing: 128.93.2.54) in directory INRIA/coq.

We may represent as a single proposition what would be a meta-level parameterized inference rule in a system like Martin-Löf's Type theory.

The underlying programming language is close to the ML-language and our system provides the possibility to execute extracted programs using ML compilers. The system PX of S. Hayashi and H. Nakano (Hayashi and Nakano, 1988) is to our knowledge the only other system to provide extraction of programs into a real programming language. A lot of work on this topic has also been done at Cornell University in the group of R. Constable (Constable et al., 1986)· the NuPRL system allows the extraction of programs, but these are interpreted and not compiled‡. One can judge the efficiency of algorithms which came out of proofs. Obviously the direct interpretation of proofs as programs introduces useless computations, but after an optimization step we get reasonably readable and efficient code.

We developed in this paper the example of a tautology-checker for propositional logic following Boyer and Moore. We found that the program extracted out of a constructive proof of termination gave a good algorithm for the normalization of IF-expressions. This is slightly in contradiction with the point of view that we have to get exactly the recursive program we start with. In this paper we showed that proofs may suggest interesting alternative solutions.

In Coq, the user represents a specification and provides a proof which determines the algorithm. So the specification part and the development part are disjoint. We are free to specify the program in a general mathematical language and prove properties of the specifications. But for some programs (for instance primitive recursive functions) the specification is really simple (equations of a certain shape) and we would like the system to automatically provide the corresponding proof. To develop a program, we need to understand the computational meaning of proofs rather than the one of specifications (like in a language like Prolog for instance). The algorithm we try to develop should not determine the specification but gives a skeleton of the proof we have to find. A tactic based on this idea (the program is a guide for the proof of a specification) is under development (Parent, 1992) and a prototype of this tactic is available in Coq V5.8. More generally more work has to be done in order to partially automatize the methodology proposed in this paper.

ACKNOWLEDGMENTS

We thank Xavier Leroy and Chet Murthy for their help in analysing the behavior of the optimized normalization function.

References

Augustsson, L. and Johnsson, T. (1989). Lazy ml user's manual, preliminary draft. Chalmers University, Sweden.

Boyer, R. S. and Moore, J. S. (1979). *A computational logic.* ACM Monograph. Academic Press.

Constable et al., R. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall.

Coquand, T. (1985). *Une Théorie des Constructions.* PhD thesis, Université Paris 7.

Coquand, T. and Huet, G. (1985). Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, Linz. Springer-Verlag. LNCS 203.

‡ Very large programs have been proved correct in NuPRL, using a slightly more traditional methodology than actual program extraction.

Coquand, T. and Huet, G. (1987). Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In T. P. L. Group, editor, *Logic Colloquium'85*. North-Holland.

Coquand, T. and Paulin-Mohring, C. (1990). Inductively defined types. In Martin-Löf, P. and Mints, G., editors, *Proceedings of Colog'88*. Springer-Verlag. LNCS 417.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Paulin-Mohring, C., and Werner, B. (1991). The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134, INRIA.

Dybjer, P. (1990). Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79.

Girard, J.-Y., Lafont, Y., and Taylor, P. (1989). *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press.

Gordon, M., Milner, R., and Wadsworth, C. (1979). *Edinburgh LCF*. LNCS 78. Springer-Verlag.

Hayashi, S. and Nakano, H. (1988). *PX, a Computational Logic*. Foundations of Computing. MIT Press.

Krivine, J.-L. (1990). *Lambda-calcul types et modèles*. études et recherche en informatique. Masson.

Krivine, J.-L. and Parigot, M. (1987). Programming with proofs. Preprint, presented at the 6th symposium on Computation Theory, Wendish-Rietz, Germany.

Leivant, D. (1983). Reasoning about functional programs and complexity classes associated with types disciplines. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science*, pages 460–469, Washington DC. IEEE Computer Society.

Leivant, D. (1990). Contracting proofs to programs. In Odifreddi, P., editor, *Logic and Computer Science,*, volume 31 of *APIC Series*, pages 279–327. Academic Press.

Leszczylowski, J. (1981). An experiment with "Edinburgh LCF". In Bibel, W. and Kowalski, R., editors, *Fifth Conference on Automated Deduction*, volume 87 of *LNCS*, pages 170–181. Springer-Verlag.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis.

Mauny, M. and Leroy, X. (1992). The Caml Light system release 0.5, Documentation and user's manual. INRIA.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of standard ML*. The MIT press.

Nordström, B. (1988). Terminating General Recursion. Report 46 of the Programming Methodology Group, University of Göteborg and Chalmers University of Technology.

Nordström, B., Petersson, K., and Smith, J. (1990). *Programming in Martin-Löf's Type Theory*. International Series of Monographs on Computer Science. Oxford Science Publications.

Parent, C. (1992). Automatisation partielle du développement de programmes dans le système COQ. Master's thesis, ENS Lyon.

Paulin-Mohring, C. (1989a). Extracting F_ω's programs from proofs in the Calculus of Constructions. In for Computing Machinery, A., editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin.

Paulin-Mohring, C. (1989b). *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7.

Paulin-Mohring, C. (1989c). Recursive programs in the Calculus of Constructions. In *France-Japan Artificial Intelligence and Computer Science Symposium*.

Paulin-Mohring, C. (1992). Inductive Definitions in the System Coq - Rules and Properties. research report 92-49, LIP-ENS Lyon. To appear in the proceedings of the conference Typed Lambda Calculi and Applications (March 93).

Paulson, L. (1986). Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74.

Plotkin, G. (1976). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1 (2):125–159.

Weis, P. et al. (1990). The CAML reference manual. Rapports Techniques 121, INRIA.

Werner, B. (1992). A normalization proof for an impredicative type system with large elimination over integers. In Nordström, Petersson and Plotkin, G. eds (1992). *Proceedings of the 1992 Workshop on Types for Proofs and Programs*.