A User's Friendly Syntax to Define Recursive Functions as Typed λ-terms

Pascal MANOURY[0]
INRIA
Domaine de Voluceau, Rocquencourt
F-78153 Le Chesnay Cedex
FRANCE
`manoury@inria.fr`

November 27, 1998

Abstract

This paper discusses the problem of translating a (recursive) function definition expressed in a ML like syntax into a typed λ-term.
In this paper, we recall how termination proofs can be solutions of the investigated problem. Then we recall how the method of [10] deals with the problem of searching termination proofs. Finally, we discuss the questions raised by the adaptation of our method to the `Coq` proof assistant.

Introduction

Type theory or higher order logic allows to express functions intentionally as typed λ-terms. This means that one can directly write, as formal objects, functional programs computing those functions.
We assume that the reader is more or less acquainted with such a formalism. Let us only recall the way recursive functions can be defined as typed λ-terms. Recursive functions are mainly defined over objects belonging to *inductive sets*. The simplest one is the set of natural numbers. Let us note it `nat`. It is inductively defined by the three following axioms :

(n1) `0`$\in$`nat`*;*
(n2) if `n`$\in$`nat` *then its successor* `(S n)`$\in$`nat`*;*
(n3) `nat` *is the least set satisfying (n1)* and *(n2).*

The axioms *(n1)* and *(n2)* introduce the two symbols `0` and `S` which we call the *constructors* of `nat` . The last axiom *(n3)* raises the existence of a structural induction principle over `nat` which asserts : *for any predicate* `P` *over* `nat`*, if* `0` *statisfies* `P` *and if for any* `m`$\in$`nat` *which satisfies* `P`*, its successor* `(S m)` *satisfies* `P` *then all* `n`$\in$`nat` *satisfies* `P`. An important fact for our purpose is that this induction principle has a computational counterpart : the `Match_rec` operator whose computational rules are

(c1) `(<P>Match_rec T0 TS 0)` *reduces to* `T0`*;*
(c2) `(<P>Match_rec T0 TS (S n))` *reduces to*
`(TS n (<P>Match_rec T0 TS n))`.

As `Match_rec` occurs twice in the *(c2)* computation, it allows to write recursive functions. Those computation rules are called ι-reduction.

But λ-terms expressing functions are rather more difficult to read than an ordinary functional program. Compare for instance the following definition *(d1)* of the addition of natural numbers as it can be given in `Coq` :

[0]This research was partially supported by ESPRIT Basic Research Action "TYPES."

```
Definition ADD : nat -> nat -> nat :=
[n,m:nat](<nat>Match_rec n
               m
               [p:nat][h:nat] (S h)).
```

with the following definition *(d2)* :

```
Recursive Definition add : nat -> nat -> nat :=
  O m => m
|(S n) m => (S (add n m)).
```

Even in this simple example, *(d2)* is closer to our intuition of the function than *(d1)*.

In [10], M. Simonot and I have presented an algorithm for automatically synthetizing termination proofs of recursive functions defined by a set of equations. We implemented this algorithm as the "compiler" of the functional programming language ProPre. In this language, the "compiling" phase constists in extracting a λ-term from the termination proof. Then we can see ProPre as a friendly syntax for writing λ-terms.

This paper investigates the problem of translating a function definition as *(d2)* into a λ-term as the one of *(d1)*. To express it more generally, we propose the following declaration schema :

```
Recursive Definition f : D1-> .. -> Dn -> D :=
 p11 .. p1n => t1
...
|pm1 .. pmn => tm.
```

This declaration must be understood as :

- `f` is the name of the constant we want to define;
- `D1-> ..  -> Dn -> D` is its intended type;
- `p11 ..  p1n => t1 | ...  |pm1 ..  pmn => tm.` describe its behaviour in terms of patterns[1]. We call the *pattern clauses* this part of a `Recursive Definition`.

Then our problem is *(p1)* : *does there exist a λ-term* `F` *of type* `D1-> ..  -> Dn -> D` *such that for all* `i`$\in$`[1..m]`, `(F pi1 ..  pin)` $\simeq_{\beta\iota}$ `ti`?

Our paper is divided into two main parts.

The first part is a recall of the algorithm presented in [10] but in a more intuitive way. Indeed, this first paper was a technical setting of the algorihm with a proof of its completeness for a given class of formal proofs. Because of its formal presentation the simple ideas on which our algorithm is based do not clearly appear. We hope that the new presentation we give here will help the readers to understand those "simple ideas" and give them some intuition about the possible improvements of the algorithm.

In the second part of this paper, we discuss some questions concerning the enhancement of our algorithm in the framework of the Coq proof assistant. Indeed, ProPre was originally designed for the second order type theory TTR (see [12])[2] whereas the Calculus of Inductive Construction (see [13] and [14]), on which the Coq proof assistant is based, is a higher order type theory. In particular, we will discuss the problem of dealing with dependent types. Of course this discussion should be available in any type theory framework strong enough to support dependent types as, for instance, Martin Löf's one (see [7]). But as we experimented the Coq system, we will use it.

But before those two parts, we want to come back to the correspondence between the λ-term given in *(d1)* and the termination proof of the function defined in *(d2)*.

[1] see (2.1) for a precise definition of *patterns*

[2] In fact, ProPre can be more simply described in a first order system *à la Gödel* (see [3]).

1 Significance of termination proofs

As we required in *(d2)* that `add` should be of type `nat -> nat -> nat`, the fact that `add` terminates can be expressed as: ∀`n`∈`nat`, ∀`m`∈`nat`, `(add n m)`∈`nat`. Let us prove it : *suppose* `n` *and* `m` *are two elements of* `nat`, *we prove by structural induction on* `n` *that* `(add n m)` ∈ `nat`*; if* `n=0` *then we have to prove* `(add 0 m)`∈`nat`, *but as we set that* `(add 0 m)` *is equal to* `m`, *we have our result by hypothesis; if* `n=(S n')`, *let us suppose that* `(add n' m)`∈`nat` *(which is our induction hypothesis), then we have to prove that* `(add (S n') m)`∈`nat`, *that is to say that* `(S (add n' m))`∈`nat` *which is true because* `S` *maps* `nat` *to* `nat` *and* `(add n' m)`∈`nat` *by induction hypothesis.*

Let us see now how the term `ADD` of *(d1)* can be interpreted as a trace of the above proof : we read `[n,m:nat]` as *suppose* `n` *and* `m` *are elements of* `nat`; if you keep in mind that `Match_rec` is the computational counterpart of the structural induction schema, then we read `Match_Rec n` as *by structural induction on* `n`; in the first case of the induction, we read `m` as *by hypothesis* and in the second case, we read the application `(S h)` as *apply the lemma* ∀`n`∈`nat`, `(S n)`∈`nat` *to the (induction) hypothesis* `h`. This unformal correspondence between proofs and terms which we only sketched is formally known as the *Curry-Howard correspondence* (see [5]).

The important difference between the proof and the corresponding λ-term comes from the lost of information from the proof to the λ-term. In particular, in the λ-term, the induction hypothesis is reduced to an anonymous inhabitant of `nat` whereas, in the proof, it has a form `(add n' m)`. Then, as proofs and λ-terms are isomorphic, and as proofs are more informative than λ-terms, we reformulate the problem *(p1)* in the following one *(p2)* : *given a* `Recursive Definition`, *does there exist a proof* Π *of* ∀`x1`∈`D1`,..,∀`xn`∈`Dn`,`(f x1..xn)`∈`D` *such that, if the* λ*-term* `F` *is extracted from* Π *then for all* `i`∈`[1..m]`, `(F pi1 .. pin)` $\simeq_{\beta\iota}$ `ti` ?

2 The ProPre method

To simplify the presentation, let us design an *ad hoc* framework[3]. We define a set of logical terms upon a set of variables and functionnal symbols as usual. Then we define the set of formulæ, upon a set of constants $\mathcal{D}$, as : if t is a logical term and D is an element of $\mathcal{D}$ then $t \in D$ is an atomic formula; if x is a variable, if D is an element of $\mathcal{D}$ and if Φ is a formula, then $\forall x \in D, \Phi$ is a formula. We have upon formulæ the usual notion of free variables and the substitution of the free occurrence of x in Φ with t is written $\Phi[t/x]$. We write $\Phi[x]$ for a formula where x occurs free. Let Γ be a sequence of formulæ, we have the basic logical rules :

$$\textit{Axiom}\quad \frac{}{\Gamma, t \in D \vdash t \in D} \qquad\qquad \textit{Eq}\quad \frac{\Gamma \vdash t \in D \quad t = u}{\Gamma \vdash u \in D}$$

$$\forall_{\in}\textit{-Elim}\quad \frac{\Gamma \vdash \forall x \in D, \Phi \quad \Gamma \vdash t \in D}{\Gamma \vdash \Phi[t/x]} \qquad\qquad \forall_{\in}\textit{-Intro}\quad \frac{\Gamma, x \in D \vdash \Phi^{(\star)}}{\Gamma \vdash \forall x \in D, \Phi}$$

$^{(\star)}$ the variable x must not be free in Γ.

Data types, as the set of natural numbers, are defined by means of introduction rules of their constructors and a schema of elimination rule for structural induction. For instance we define the new predicate constant Nat, which intends to represent natural numbers, as :

$$\frac{}{\vdash 0 \in Nat} \qquad \frac{\Gamma \vdash n \in Nat}{\Gamma \vdash (S\ n) \in Nat} \qquad \frac{\Gamma \vdash \Phi[0] \quad \Gamma, m \in Nat, \Phi[m] \vdash \Phi[(S\ m)]}{\Gamma, n \in Nat \vdash \Phi[n]}$$

[3] This framework is nothing original, but only a convenient syntax for the presentation.

Then, given a `Recursive Definition` for a function f, we will try to find a proof of $\forall x_1 \in D_1, .., \forall x_n \in D_n, (f\ x_1..x_n) \in D$ using the described above framework, the equational data and the rules defining the data types $D_1, .., D_n, D, etc...$ The proof is searched within a particular class of proofs which we designed and called the *R-proofs*. An *R-proof* is defined according to a set of equations, and the interesting property of the *R-proofs* is that the extracted λ-term is such that both members of each equation are $\beta\iota$-convertible.

So, the precise problem we discuss now is the restricted formulation of *(p2)* in *(p3)* : *given a* `Recursive Definition`*, does there exist an R-proof of* $\forall x_1 \in D_1, .., \forall x_n \in D_n, (f\ x_1..x_n) \in D$?

2.1 Definitions and Glossary

We call a *data type* any set defined by means of introduction rules for its constructors and an elimination rule for its structural induction schema. The shape of an introduction rule for a constructor c of a data type D is :

$$\frac{\Gamma \vdash x_1 \in D_1 \quad .. \quad \Gamma \vdash x_r \in D_r}{\Gamma \vdash (c\ x_1..x_r) \in D}$$

where for all $i \in [1..r]$, $D_i = D$ or D_i is an already defined data type. When $r = 0$, the rule reduces to $\vdash c \in D$.
We say that for any $i \in [1..r]$, t_i is in *inductive position* in $(c\ t_1..t_r)$ iff $D_i = D$. More generally, we say that a subterm t' of t is in inductive position in t iff there exists a subterm $(c\ ..t'..)$ of t such that t' is in inductive position in $(c\ ..t'..)$. If t' is a variable (say x), we say that x is an *inductive variable* of t.
$D, D_1, D_n, \ldots$ will be constants denoting data types. $c, c_1, c_l, \ldots$ will be constants denoting constructors. Structural induction schemas will be denoted by the generic expression *Rec-rule*.
We call a *pattern* any linear term p in which the only functional symbols are constructors. The symbols $p, p_1, p_n, p_i^j, \ldots$ will denote patterns.

If $(f\ p_1..p_n) = t$ is an equation for a function f, we call *inductive call* any pair of tuples $(p_1, .., p_n)(t_1, .., t_n)$ such that $(f\ t_1..t_n)$ is a subterm of t.

Let t and u be two terms. We say that *t matches u according to σ* iff σ is a variable substitution such that $\sigma(t) = u$. Variable substitutions will be denoted with $\sigma, \sigma', \sigma_i, \sigma_t^i, \ldots$

Let α be a sequence of natural numbers and t a term. We say that *α is a position in t* iff either α is the empty sequence (denoted ε) or $t = (h\ t_1..t_n)$, $\alpha = i.\alpha'$ for some $i \in [1..n]$ and α' is a position in t_i. The subterm of t in position α in t will be denoted by $t|_\alpha$.

2.2 A first remark about the *Rec-rules*

One effect of a *Rec-rule* is to transform the goal to prove $\Gamma, x \in D \vdash \Phi[x]$ into a set of subgoals $\Gamma_1 \vdash \Phi[(c_1\ x_1^1..x_{r_1}^1)/x], .., \Gamma_l \vdash \Phi[(c_l\ x_1^l..x_{r_l}^l)/x]$ where l is the number of constructors of D, and for all $i \in [1..l]$, c_i is the i^{th} constructor of D, r_i its arity and $x_1^i, .., x_{r_i}^i$ are new variables. In others words, a *Rec-rule* acts as a *case analysis* or a *pattern matching* rule.
Formulæ we are interested to prove are written in general : $\forall x_1 \in D_1, .., \forall x_n \in D_n, (f\ x_1..x_n) \in D$. Let us focus on the term $(f\ x_1..x_n)$ and write our formulæ as $\Phi[(f\ x_1..x_n)]$. Using a *Rec-rule* on one of the x_i's produces :

$$\frac{\Gamma_1 \vdash \Phi[(f\ x_1..(c_1\ x_1^1..x_{r_1}^1)..x_n)] \ ..\ \Gamma_l \vdash \Phi[(f\ x_1..(c_l\ x_1^1..x_{r_1}^1)..x_n)]}{\Gamma \vdash \Phi[(f\ x_1..x_i..x_n)]}$$

Iterating the use of *Rec-rules* on each subgoals and so on, we will produce a proof-tree with leaves of the form $\Gamma' \vdash \Phi'[(f\ p_1..p_1)]$ where $p_1, .., p_n$ are patterns. We call such a proof-tree a *Rec-tree* and each branch of it is called a *Rec-path from* $(x_1, .., x_n)$ *to* $(p_1, .., p_n)$, or more simply, a *Rec-path to* $(p_1, .., p_n)$.

2.3 General idea of the method

Remind that we want to prove $\vdash \forall x_1 \in D_1, .., \forall x_n \in D_n, (f\ x_1..x_n) \in D$ and that to reach this goal we have at our disposal a set of equations.
First of all, let us focus our attention on one of these equations, say $(f\ p_1..p_n) = t$. We assumed that $p_1, .., p_n$ are patterns. So, using the above remarked effect of *Rec-rules*, starting from $\vdash \Phi[(f\ x_1..x_n)]$, we can build a *Rec-path* to $(p_1, .., p_n)$.
Then, using our equation, we transform this goal into $\Gamma \vdash \Phi'[t]$. Assume that $\Phi'[t]$ has a form $\forall x_1 \in D_1, .., \forall x_k \in D_k, t \in D$ for some $k \in I\!N$. We now transform our goal into $\Gamma'' \vdash t \in D$ (where $\Gamma'' = \Gamma' \cup \{x_1 \in D_1, .., x_k \in D_k\}$). Let us consider at this point the different possibilities for t :

1. if t is a variable, we whish to use the *Axiom* rule;
2. if t is $(g\ t_1..t_1)$ (for some $k \in I\!N$) with $g \neq f$, we assume that g is an already known constant such that $\forall x_1 \in D_1, .., \forall x_k \in D_k, (g\ x_1..x_1) \in D$, and then we go on proving the subgoals $\Gamma'' \vdash t_1 \in D_1, .., \Gamma'' \vdash t_k \in D_k$;
3. if t is $(f\ t_1..t_1)$, we look in Γ'' for an induction hypothesis (say H) that we assume to be in the form $\forall x_1 \in D_1, .., \forall x_k \in D_k, (f\ p'_1..p'_1) \in D$ (for some $k \in I\!N$), we check that H is such that $(f\ p'_1..p'_1)$ matches $(f\ t_1..t_1)$ according to, say, σ and that for all variable z occuring free in H, $\sigma(z) = z$. Then we go on proving $\Gamma'' \vdash \sigma(x_1) \in D_1, .., \Gamma'' \vdash \sigma(x_k) \in D_k$.

We will say in this case that H *can be applied to* $((f\ t_1..t_1))$ or that H *satisfies the inductive call* $(p_1, .., p_n)(t_1, .., t_n)$.

This short informal presentation leads to the following remarks.

1. to be able to use the *Axiom* rule requires that any variable occuring in t occurs in $p_1, .., p_n$;
2. any (functional) constant (except f) occuring in t must be already known.[4]

We can remark that these two points can be checked on the equational data while the existence of the right induction hypothesis needs some further investigations.

2.4 Precising the method

We presented above our method considering only one equation $(f\ p_1..p_n) = t$ and focused our attention on the *Rec-path* to $(p_1, .., p_n)$. Of course, in general, we have more than one equation. So let $\mathcal{E} = \{(f\ p_1^1..p_n^1) = t_1, .., (f\ p_1^m..p_n^m) = t_m\}$ be an equational data. The aim of our method is then to build a *Rec-tree* containing a *Rec-path* to each $(p_1^i, .., p_n^i)$ for all $i \in [1..m]$. Conversely, as we want f to be totally defined on its domain, we also need each branch of the *Rec-tree* to be a *Rec-path* to one $(p_1^i, .., p_n^i)$. We call such a *Rec-tree* a *complete Rec-tree* for $\mathcal{E}$.
Then, as each leaf of the complete *Rec-tree* is extended by using one rewriting step and then *Axiom* and $\forall_\in$*-Elim* rules, we can restate our problem *(p3)* as *(p4)* : *given an equational data $\mathcal{E}$, does there exist a complete Rec-tree for $\mathcal{E}$ such that for every inductive call $(p_1^i, .., p_n^i)(t_1, .., t_n)$ of $\mathcal{E}$, if $\vdash \Phi_0, .., \Gamma \vdash \Phi$ is the Rec-path to $(p_1^i, .., p_n^i)$ then Γ contains an induction hypothesis which can be applied to $(f\ t_1..t_n)$* ?

This restricted problem is now decidable as there is a finite numbers of complete *Rec-trees* for a given equational data $\mathcal{E}$. A decision procedure should be to enumerate all those *Rec-trees* and to look among them for a good one.
But we can go further on in our analysis and design a more efficient algorithm. These analysis and the algorithm it suggests is the object of the end of this section.

2.5 A second remark about *Rec-rules*

The general shape of *Rec-rules* is:

$$\frac{\Gamma_1 \vdash \Phi[(c_1\ x_1^1..x_{r_1}^1)/x]\ ..\ \Gamma_l \vdash \Phi[(c_l\ x_1^l..x_{r_l}^l)/x]}{\Gamma \vdash \Phi[x]}$$

[4]This second point forbids the use of mutually recursive definitions.

Let $i \in [1..l]$ and suppose that c_i is given such that $\forall x_1 \in D_1, .., \forall x_{r_i} \in D_{r_i}, (c_i\ x_1..x_1) \in D \in D$. We can remark that $\Gamma_i = \Gamma \cup \Gamma'_i$ and for all $j \in [1..r_i]$, if $D_j \neq D$ then the only formula of Γ'_i containing x^i_j is $x^i_j \in D_j$ while if $D_j = D$ then Γ'_i contains exactly two formalæ where x^i_j occurs : $x^i_j \in D$ and $\Phi[x^i_j/x]$. Now the important point is to remark that in the case where $D_j = D$, the x^i_j's are the *inductive variables* of $(c_i\ x^i_1..x^i_{r_i})$).

2.6 Accompanying *Rec-rules*

In 2.3, we assumed that induction hypothesis have form $\forall x_1 \in D_1, .., \forall x_k \in D_k, (f\ p'_1..p'_1) \in D$ (for some $k \in I\!N$). Then we can freely decide that *Rec-rules* will always be applied to formulæ where all inductive variables of $p'_1, .., p'_n$ are free while all other variables are bounded. So each use of a *Rec-rule* will be accompanied with some $\forall_\in$-*Intro* and $\forall_\in$-*Elim* rules before and after its proper use in order to keep the freeness or boundness of our distinguished variables.

Suppose we want to prove

$$\Gamma \vdash \forall x_1 \in D_1, .., \forall x_i \in D_i, .., \forall x_k \in D_k, \to (f\ p'_1..p'_1) \in D$$

by induction on x_i. Then, before using *Rec-rule*, we transform our goal into

$$\Gamma, x_i \in D_i$$
$$\vdash \forall x_1 \in D_1, .., \forall x_{i-1} \in D_{i-1}, \forall x_{i+1} \in D_{i+1}, .., \forall x_k \in D_k, (f\ p'_1..p'_1) \in D$$

Now, suppose we are going to prove $\Gamma \vdash \Phi[x]$ using a *Rec-rule* on x and suppose that

$$\Gamma, a_1 \in D_1, .., a_r \in D_r, y_1 \in D', \Phi[y_1/x], .., y_s \in D', \Phi[y_s/x]$$
$$\vdash \Phi[(c\ a_1..a_r\ y_1..y_s)/x]$$

is one of the subgoals generated by the *Rec-rule*[5]. Then we transform this subgoal into

$$\Gamma, y_1 \in D', \Phi[y_1/x], .., y_s \in D', \Phi[y_s/x]$$
$$\vdash \forall a_1 \in D_1 a_1, .., \forall a_r \in D_r \Phi[(c\ a_1..a_r\ y_1..y_s)/x]$$

2.7 Generalized *successor relation*

As we fixed the form of induction hypothesis, it is possible, for a given inductive call $(p_1, .., p_n)(t_1, .., t_n)$, to forsee some relation between some p_i and t_i without which there cannot exist any induction hypothesis satisfying $(p_1, .., p_n)(t_1, .., t_n)$.

Assume that f is a unary function and $(p)(t)$ an inductive call of its definition. In our method, because of its fixed form $(\forall x_1 \in D_1, .., \forall x_k \in D_k, (f\ p') \in D)$ an induction hypothesis H can be completely characterized by the pattern p' and the set of its variables occuring free in H. As H is introduced by a *Rec-rule*, p' can be written as $q[y/x]$ if $\forall x_1 \in D_1, .., \forall x_k \in D_k, (f\ q) \in D$ was the formula we attempted to prove by induction on x. We know after (2.6) that the set of free variables of H is y plus the set of inductive variables of q omitting x.
Then, if we suppose that H can be applied to $(f\ t)$ then $q[y/x]$ matches t; as q is generated when building the *Rec-path* to p, q matches also p. Then, considering the inductive call $(p)(t)$, we can forsee this q and define the following relation between p and t we call the *successor relation* and write $p \gg t$:
$p \gg t$ iff there exists a most general pattern q and a position α in q such that q matches p and t (say according, respectively, to σ and σ'), $q|_\alpha$ is an inductive variable of p (say y), $p|_\alpha = (c\ ..y..)$, $t|_\alpha = y$ and for all inductive variable $z \neq y$ of q, $\sigma(z) = \sigma'(z) = z$.

If $p = (c\ ..y..)$ and $t = y$, we note $p \gg_\varepsilon t$. The name *successor relation* comes from the fact that $s(x) \gg x$. More generaly we have $s^{n+1}(x) \gg s^n(x)$ for all $n \in I\!N$. On lists of natural numbers, for instance, we have $(Cons\ n\ x) \gg x$ and $(Cons\ n\ (Cons\ m\ x)) \gg (Cons\ (g\ n\ m)\ x)$ for any function g.

In general, let $(p_1, .., p_n), (t_1, .., t_n)$ be an inductive call, we claim that *if in a Rec-tree there exists a Rec-path to $(p_1, .., p_n)$ (say $\vdash \Phi_0, .., \Gamma \vdash \Phi$) such that Γ contains an induction hypothesis which can be applied to*

[5] To simplify, we suppose *wlog* that inductive variables of $(c\ a_1..a_r\ y_1..y_s)$ are $y_1, .., y_s$.

$(f\ t_1..t_1)$ *then there exists* $i \in [1..n]$ *such that* $p_i \gg t_i$.

Note that we can check on the equational data if for any inductive call $(p_1,..,p_n)$,
$(t_1,..,t_n)$ there exists or not some $i \in [1..n]$ such that $p_i \gg t_i$. If there does not exist, there is no use to try our method.

2.8 The choice of induction variables

For an inductive call, the above result ensures that there may exist a *Rec-path* along which is produced an expected induction hypothesis, but it doesn't indicate how to build this *Rec-path*.
For instance, let c be a binary constructor which both arguments are inductive and let $((c\ (c\ x_1\ x_2)\ (c\ y_1\ y_2)))\ ((c\ x_1(c\ y_1\ y_2)))$ be an inductive call. We have $(c\ (c\ x_1\ x_2)\ (c\ y_1\ y_2)) \gg (c\ x_1\ (c\ y_1\ y_2))$. But we have two *Rec-pathes* to $((c\ (c\ x_1\ x_2)\ (c\ y_1\ y_2)))$. Let us sketch them as sequences of patterns :

$$\begin{array}{lll} 1. & (c\ x\ y); \quad (c\ (c\ x_1\ x_2)\ y); \quad (c\ (c\ x_1\ x_2)\ (c\ y_1\ y_2)) \\ 2. & (c\ x\ y); \quad (c\ x\ (c\ y_1\ y_2)); \quad (c\ (c\ x_1\ x_2)\ (c\ y_1\ y_2)) \end{array}$$

Only the second one will produce a useful induction hypothesis (we let the reader check it).

As shown by this example, when building a *Rec-path*, we have to make some choice among the possible variables. So our aim is to collect some informations from the equational data (and especially from inductive calls) in order to guide this choice.

2.9 What has to be bound

The first kind of information we can collect from the equational data is what has to be bound in the expected induction hypothesis: suppose now that f is a binary function over natural numbers and consider the following inductive call $((S\ n)\ 0)(n\ (S\ 0))$. It is obvious that the *Rec-path* to $((S\ n)\ 0)$ must begin with a *Rec-rule* on the first argument to produce $\forall m \in Nat, (f\ (S\ n)\ m) \in Nat$ as induction hypothesis where m can be instanciated with $(S\ 0)$. Then, and only then, we can apply a *Rec-rule* to the second argument to achieve our *Rec-path*.

The information that an argument, and more generaly, that any position in the tuple of arguments, has to correspond to a bound variable of an eventual induction hypothesis can be checked *a priori* on the set of equations.

2.10 What can be free

Let us consider now the following equation $(f\ (S\ n)\ (S\ m)) = (f\ n\ (f\ (S\ n)\ m))$ which contains two inductive calls :

$$\begin{array}{ll} ((S\ n),(S\ m)) \quad (n,(f\ (S\ n)\ m)) & \text{and} \\ ((S\ n),(S\ m)) \quad ((S\ n),m) & \end{array}$$

The point now is that on the same *Rec-path* we have to produce two different induction hypothesis.
Using what we state in the above section, we know that we have to begin the *Rec-path* by applying a *Rec-rule* on the first argument to produce $\forall m \in Nat, (f\ (S\ n)\ m) \in Nat$ where m can be instantiated by $(f\ (S\ n)\ m)$. Moreover this begining doesn't waste the possibility to produce an induction hypothesis for our second inductive call because, in $((S\ n),(S\ m))\ ((S\ n),m))$, the first argument remains unchanged. Then we can end our *Rec-path* by applying a *Rec-rule* to the second argument. This will produce the induction hypothesis $(f\ (S\ n)\ m) \in Nat$ which, of course, can be applied to $(f\ (S\ n)\ m)$.

In general, this notion of *"remaining unchanged"* doesn't coincide with litteral equality between terms as illustrated by the following example on lists of natural numbers : $(f\ (Cons\ n\ x)\ (Cons\ m\ y)) = (f\ x\ (f\ (Cons\ (g\ n\ m)\ x)\ y))$. We can here produce the two required induction hypothesis too when we apply first a *Rec-rule* to the first argument and then a *Rec-rule* to the second. This will give $\forall n \in$

$Nat, (f\ (Cons\ n\ x)\ y) \in D$ as induction hypothesis where n can be instanciated with $(g\ n\ m)$. This is due to the fact that we decided to keep bound all variables which are not in inductive position.

We express formally our *"remaining unchanged"* relation as : *let p be a pattern and t a term, we say that p is equivalent to t and write $p \equiv t$ iff (a) if p is a variable then $p = t$, else (b) if $p = (c\ p_1..p_r)$ then there exists a most general pattern $q = (c\ q_1..q_n)$ such that q matches p and t (say according to σ and σ'), the set of inductive variables of q is included in the one of p and for each inductive variable z of q, $\sigma(z) = \sigma'(z) = z$.*

2.11 Guiding the construction of *Rec-trees*

To construct a *Rec-tree* following our method, we use the above collected informations in the following way. Let $\Gamma \vdash \Phi[(f\ p'_1..p'_n)]$ be a leaf of a partial *Rec-tree*, let $(p_1^{k_1}, .., p_n^{k_1})(t_1^1, .., t_n^1)$, $.., (p_1^{k_j}, .., p_n^{k_j})(t_1^j, .., t_n^j)$ be the set of inductive calls such that for all $i \in [1..j]$, $(p'_1, ..p'_n)$ matches $(p_1^{k_i}, .., p_n^{k_i})$ (say according to σ_p^i); $(p'_1, ..p'_n)$ matches $(t_1^i, .., t_n^i)$ (say according to σ_t^i) and Γ doesn't contain any induction hypothesis which can be applied to $(f\ t_1^i..t_n^i)$. Note that this is true for all inductive call at the beginning when our goal is $\vdash \forall x_1 \in D_1, .., \forall x_n \in D_n, (f\ x_1..x_n) \in D$.
Now, let x be a variable of $(p'_1, .., p'_n)$, we will choose x as an induction variable iff :

(i) for all $i \in [1..j]$, $\sigma_p^i(x)$ and $\sigma_t^i(x)$ are in the $\equiv$ or $\gg$ relation and

(ii) if $\sigma_p^i(x) \gg_\varepsilon \sigma_t^i(x)$ then for all inductive variable $z \neq x$ of $(p'_1, .., p'_n)$, $\sigma_p^i(z) = \sigma_t^i(z) = z$.

The first point mainly means that we will not apply *Rec-rules* to variables which are required to be bound in the induction hypothesis. The second point means that before producing some required induction hypothesis (as indicated by the $\gg_\varepsilon$ labelling) we have to check that the free variables of the current formula (*ie* inductive variables of $(p'_1, .., p'_n)$) which are not equal to x do correspond to the same variables on the right handside of the inductive call.

If x doesn't satisfy *(i)* or *(ii)* we either look for another variable in $(p'_1, .., p'_n)$, or we backtrack to the father of $\Gamma \vdash \Phi[(f\ p'_1..p'_n)]$ when all variables of $(p'_1, .., p'_n)$ have been tried.
Choosing induction variables according to the *(i)* and *(ii)* criteria preserves the following invariant : we eventually produce some required induction hypothesis and we do not waste the possibility of getting a later required induction hypothesis.

Criteria *(i)* and *(ii)* widely restrict the search space. Moreover, as the relations they are based on can be checked *a priori* on the equational data, we can use them as a labeling of the left hand side patterns and then improve the efficiency of the algorithm.

3 Coq & ProPre

The method we described below has been implemented in the V5.10 distribution of the Coq proof assistant. It is activated by the command `Recursive Definition` according to the schema we gave in our introduction. With this distribution, one can find the file `ProPre_test.v` which contains a number of example using the `Recursive Definition` command. One can find also the file `ProPre_sorting.v`. It contains examples of definitions of several sorting algorithm and a proof of correctness of the definition of the insertion sorting. Those files are available in the directory `$COQTOP/THEORIES/PROGRAMS`.

3.1 Behaviour the `Recursive Definition` command

A `Recursive Definition` command either is processed with success or fails.

If it fails, an error message is displayed explaning to the user the reasons of this failure. Those reasons can be syntax errors, type mismatching as well as impossibility for ProPre to build the termination proof.

If it succeeds then a given term `F` is generated and declared as value of the variable `f`. Then, for each pattern clause, we state the corresponding equational lemma and prove it with a simple tactic. For instance the Recursive Definition *(d2)* of our Introduction has on the Coq environment the effect of the following script :

```
Definition add :=
 [x1:nat]
  <nat->nat>Match x1 with
     [x2:nat]x2
     [x1_1:nat][h1_1:nat->nat][x2:nat](S (h1_1 x2))
     end.
Theorem add_eq1 : <nat>(m:nat)(add O m) = m
 Intros; Unfold add; Simpl; Apply refl_equal.
 Save.
Theorem add_eq2 :
 <nat>(n,m:nat)(add (S n) m)=(S (add n m)).
 Intros; Unfold add; Simpl; Apply refl_equal.
 Save.
```

As we announced in our introduction, it is time now to examine the questions raised by the enhancement of our method in Coq.

3.2 From low to high order

The Coq proof assistant is based on the *Calculus of Inductive Constructions* (see [13] and [14]). This higher order logical framework is much more expressive than the first order one we designed in 2 to explain our method. Then one can ask the question : *"how far can be extended the method ?"*.
At the moment, we have no positive answer to this question. Indeed, even at the first order level, the method can be extended and we are presently working on some of those extensions. If we are concerned with higher order, we realized some extensions in the implemention we made for Coq, but they use only a little part of what the Calculus of Inductive Constructions allows.

We introduced in 2 and 2.1 the notion of *data types*. Objects belonging to those data types are computational objects in the sense that any element of a data type has a unique normal form, according to $\beta\iota$-conversion. The data types as we intend them can be defined in Coq as `Inductive Set`. That is to say : inductive types of type `Set` in the Calculus of Inductive Construction.
The method we presented allows to define functions which type, in Coq, has a form `D1 -> .. -> Dn -> D` where `D1`,..,`Dn` and `D` are data types in our basic sense.

3.3 Higher order functions

A first extention which can be easely realized is to allow the definition of higher order functions. For instance, we can allow a functional iterator. The way to do it is to allow the definition of function having type

`T1 -> .. -> Tk -> D1 -> .. -> Dn -> D`

where `T1`, .., `Tk` are not required to be data types. As `T1`, .., `Tk` are not data types, there is no sense to use the corresponding arguments in the *pattern clauses*. We can deal with those arguments as as parameters of the function. Then we introduce in the syntax of the Recursive Definition command a *parameters list*. For instance, the functional iterator can be defined this way :

```
Recursive Definition it [f:nat -> nat; a:nat] : nat -> nat :=
 O => a
|(S n) => (f (it f a n)).
```

The type of the `it` function is `(nat->nat)->nat->nat->nat`.

A second extension of this kind is to allow the type of the aimed function to be dependent using for this the same trick. We can then define a *polymorphical* iterator :

```
Recursive Definition polit [A:Set; f:A -> A; a:A] : nat -> A :=
 O => a
|(S n) => (f (polit A f a n)).
```

The type of `polit` is `(x1:Set)(x1->x1)->x1->nat->x1`.

Note that doing this, we weaken the constraints on `D`. At the moment, our implementation allows `D` to be only a parameter of the function or the constant `Prop`. This last possibility allows to define propositional functions as :

```
 Recursive Definition sorted_list : list -> Prop :=
  Nil => True
|(Cons n Nil) => True
|(Cons n (Cons m x)) => (le n m)/\(sorted_list (Cons m x)).
```

Extension of this idea of weakening constraints on `D` will lead to give the possibility of defining functions which compute functions or predicates. But we have not implement this feature yet.

To summarize those kinds of extension we can say that, without deeply modifying the method, one should be able to define functions of the type :

`(X1:T1)..(Xk:Tk)D1 -> .. -> Dn -> (Y1:U1)..(Yl:Ul)U`

where the `(X1:T1)`, .., `(Xk:Tk)`, `(Y1:U1)`, .., `(Yl:Ul)` can be dependent or not and where `D1`, .., `Dn` remain data types.

The central position of data types is keept to allow both pattern matching definitions and inductive reasonning. The question of improving data types is discussed in the following section.

3.4 Inductive sets and patterns

In this section, we will investigate with the help of some examples the question of the type of the arguments used in the pattern clauses of the `Recursive Definition`.

Parametrized data types As a first example, we will consider paramatrized data types as the polymorphical lists :

```
Inductive p_list [A:Set] : Set :=
 p_lNil : (p_list A)
|p_lCons : A -> (p_list A) -> (p_list A).
```

Here, we cannot consider an expression such as `(p_Cons A a x)` as a pattern in the usual sense. Indeed, if we know that any object of `nat` is either `O` or `S`, what can we say about the objects of `Set` and hence about the objects of `p_list` ? In particular, what can be a complete matching for `p_list` ?
The answer we give to this question is to fix the dependent arguments of the constructors of `p_list` to be a constant (as `nat`, `bool`, ..) or a parameter of the function. For instance, the concatenation function of polymorphical lists can be defined as :

```
Recursive Definition append [A:Set] :
  (p_list A) -> (p_list A) -> (p_list A) :=
 (p_Nil A) y => y
|(p_Cons A a x) y => (Cons A a (append A x y)).
```

Note that when we use one instance of `p_list`, this instance must be the same in all pattern clauses.

Local dependency Unfortunately this solution forbids the use of local dependency in the constructors. Consider, for instance the type of heterogeneous lists we can define as :

```
Inductive h_list : Set :=
 h_Nil : h_list
|h_Cons : (A:Set)A -> h_list -> h_list.
```

Now if we want to define the concatenation of those lists, we would write :

```
Recursive Definition h_append : h_list -> h_list -> h_list :=
 h_Nil y => y
|(h_Cons A a x) y => (h_Cons A a (h_append x y)).
```

But then, the system raises the error message :

`variable "A" should be declared in parameters list.`

Functional constructors In the Calculus of Inductive Constructions, one can define inductive sets whose constructors are functional. For instance :

```
Inductive ord : Set :=
 o_zero : ord
|o_succ : ord -> ord
|o_lim  : (nat -> ord) -> ord.
```

The difficulty with adapting our method to such kind of data types comes from the central role that the notion of *inductive position* (of an argument of a constructor) plays in our algorithm. Indeed, *R-proofs* themselves are strongly connected with this notion of inductive position. The first point is that, in our proof search, we identified inductive variables and free variables (see 2.6). An other point is that the analysis of *R-proofs* on which we based our algorithm assumes a structural induction principle whose shape reflects the distinguishing between inductive positions and other positions. As, in the `o_lim` case of `ord` objects, the inductive argument itself is hidden by a function, we don't see how it could be possible to define the so called *generalized successor relation* for this type.

Inductive families Let us discuss now an other case of possible inductive definition in Coq : inductive families of types. A well known example of this kind of inductive definition is the set of lists of given length :

```
Variable A : Set.

Inductive n_list : nat -> Set :=
 n_Nil  : (n_list O)
|n_Cons : (n:nat)A -> (n_list n) -> (n_list (S n)).
```

Here, as `nat` is an admissible data type for pattern clauses, we could think that objects of `n_list` are admissible too. For instance, a simple case analysis gives the length of a list :

```
Recursive Definition n_length : (n:nat)(n_list n) -> nat :=
 n_Nil => O
|(n_Cons n a x) => (S n).
```

But we face now with an other problem. Indeed, let us write, for instance, what should be the concatenation function for such lists :

```
Recursive Definition n_append :
  (n,m:nat)(n_list n) -> (n_list m) -> (n_list (plus n m)) :=
 n_Nil y => y
|(n_Cons n a x) n_Nil => (n_Cons n a x)
|(n_Cons n a x) (n_Cons m b y)
 => (n_Cons (plus n m) a
            (n_append (pred (n_length x)) m x (n_Cons m b y))).
```

where pred is the predecessor function. To accept such a definition implies that we should be abble to check, that is to say to *prove*, the following :

- `(plus O m)) = m` for the first equation;
- `(plus n O)) = n` for the second equation;
- `(plus n m) = (S (plus (pred (n_length x)) m))` for the third equation.

Those proofs are widely out of the scope of our method.

Mutual Inductive Types The last case we will discuss is the one of mutualy defined inductive types. For instance, one can define finitary branching trees as the mutual inductive type :

```
Mutual Inductive tree : Set :=
 Lf : tree
|Br : forest -> tree
with forest : Set :=
 No_tree : forest
|Some_trees : tree -> forest -> forest.
```

Then the function extracting the list of `Lf` from a `tree` could be :

```
Recursive Definition Lf_list_of_tree :
  tree -> (list tree) :=
 Lf => (Cons tree Lf (Nil tree))
|(Br No_tree) => (Nil tree)
|(Br (Some_trees t f))
 => (append tree (Lf_list_of_tree t)
                 (Lf_list_of_tree (Br f))).
```

But if you try it, the `Recursive Definition` command raises the error message:
`Error: in Recursive Definition "Lf_list_of_tree" (equation 3) :`
`can't find argument on which apply structural induction`
Here again, the reason for it is that our method is very closed to the the notion of *inductive position*. Indeed, at the moment, our method is not able to detect that `t` should be regarded as being in *inductive position* in `(Br (Some_trees t f))`. Then, our *generalized successor relation* is too weak to deal with this case.

4 To conclude

The lessons we have learned from our experiments of adapting the method presented in 2 to the richer framework of the Calculus of Inductive Construction are of two kinds.

Firstly, even if we use only first order features we gain that *what is simple can be simply written.* And this is true what ever can be the "host framework". This could be an argument in favour of keeping a simple framework as the basis of our method. Indeed, the simple logical basis of ProPre ensures its portability.

Secondly the failures of the method we discussed in 3.4 leads to the following remarks. Some of those failures seem to come from a lack of analysis of the method in front of a definition of data types more general that the one originaly used. Then a deeper study of this subject can surely improve our method.

But we must keep in mind that trying to improve more and more an algorithm which attacks an undecidable problem (as the one of termination) is nothing but trying to fill the Danaïd's barrel.
Of course, there are important possible improvements as using a generalized induction principle instead of the structural one in the definition of *R-proofs*. But, in our opinion, a more significant improvement should be to enhance the present method (with its limitation) in an *interactive definition tool.* The idea should

be to build (may be step by step) a pattern definition as it is done if Alf, for instance, and to generate in parallel some constraints for termination. Those constraints can be compared to the *logical subgoals* that the Program tactic of Coq generates (see [11]). Those constraints could be solved either automaticaly as it is currently done when they are simple enough or using the powerness of all the proof assistant if they are not.

Acknowledgement I would like to thank G. Huet who offered me the opportunity to joint his group and to develop the ProPre's methodology in the Coq system.

References

[1] H. Barendregt *The Lambda Calculus, its Syntax and Semantics* ; North Holland, 1981, 1984.

[2] C. Cornes et al. *The Coq Proof Assistant Reference Manual Version 5.10* ; Internal Report INRIA.

[3] K. Gödel *Über eine bisher noch nicht Erweiterung des finiten Standpunktes* ; Dialectica 12, 1958.

[4] W. Hodge, B. Watson *On a hitherto unexploited extension of finitary standpoint* ; J.P.L. 9, 1980, english translation of [3].

[5] W.A. Howard *The formulæ-as-types notion of construction* ; in J.R. Hindley, editor, To H.B. Curry : Essays on Combinatory Logic, lambda-calculus and formalism, Seldin, J.P. 1980.

[6] J.L Krivine, M. Parigot *Programming with proofs* ; in SCT 87 (1987), Wendish-Rietz ; in EIK 26 (1990) pp 146-167.

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

[8] P. Manoury, M. Parigot, M. Simonot ProPre *: a programming language with proofs* ; in proceedings LPAR 92, LNAI 624.

[9] P. Manoury and M. Simonot *Des preuves de totalité de fonctions comme synthèse de programmes* ; Phd Thesis, Université Paris 7, December 1992.

[10] P. Manoury and M. Simonot *Automatizing termination proof of recursilvely defined functions* ; in TCS, 135 (1994), pp 319-343..

[11] C. Parent *Synthese de preuves de programmes dans le Calcul des Constrctions Inductives* ; PhD thesis, ENS Lyon, January 1995.

[12] M. Parigot *Recursive programming with proofs* ; in TCS 94 (1992) pp 335-356.

[13] C. Paulin-Mohring *Inductive Definitions in the System Coq - Rules and properties* ; proceedings TLCA 93, LNCS 664 (1993).

[14] B. Werner *Une théorie des constructions inductives* ; Phd Thesis, Université Paris 7, May 1994.