The Coq Proof Assistant
Users friendly `Recursive Definition`

DRAFT - November 28, 1998

Version 5.10 *

Pascal MANOURY

Projet Coq

INRIA-Rocquencourt† — **CNRS - ENS Lyon** ‡

*This research was partly supported by ESPRIT Basic Research Action "Types" and by the GDR "Programmation" co-financed by MRE-PRC and CNRS.

†B.P. 105, 78153 Le Chesnay CEDEX, France.

‡LIP URA 1398 du CNRS, 46 Allée d'Italie, 69364 Lyon CEDEX 07, France.

1st Printing November 25th, 1994

©INRIA 1994

As a new feature of CoqV5.10, one can define functions in the following syntax :

```
Recursive Definition f [X1:U1;..;Xk:Uk] : T1-> .. -> Tn -> T :=
 p11 .. p1n => t1
...
|pm1 .. pmn => tm.
```

This definition schema must be understood as :

- `f` is the name of the constant we want to define;
- `(X1:U1)..(Xk:Uk) T1-> .. -> Tn -> T` is its intended type;
- `p11 .. p1n => t1 | ... |pm1 .. pmn => tm.` describe its behaviour in terms of patterns. We call the *patterns clauses* this part of the `Recursive Definition`.

The semantics of the `Recursive Definition` is to define the new constant `f` as a term (say `F`) of the intended type providing that `F` has the intended computational behaviour expressed in the *patterns clauses*. Moreover, a `Recursive Definition` states and proves all equational theorems corresponding to *patterns clauses*. In other words, a `Recursive Definition` is equivalent to :

```
Definition f : (X1:U1)..(Xk:Uk) T1-> .. -> Tn -> T := F.
Theorem f_eq1 : (X1:U1)..(Xk:Uk)(x11:V11)..(x1l:V1l)(F X1 .. Xk p11 .. p1n)=t1.
Intros; Unfold F; Simpl; Trivial. Save.
...
Theorem f_eqm : (X1:U1)..(Xk:Uk)(xm1:Vm1)..(xml:Vml)(F X1 .. Xk pm1 .. pmn)=tm.
Intros; Unfold F; Simpl; Trivial. Save.
```

For instance, one can write :

```
Coq < Recursive Definition Ack : nat -> nat -> nat :=
Coq <  0 m => (S m)
Coq < |(S n) 0 => (Ack n (S 0))
Coq < |(S n) (S m) => (Ack n (Ack (S n) m)).

Coq < Ack_eq1 is defined
Ack_eq2 is defined
Ack_eq3 is defined
Ack is recursively defined.

Coq <
```

Unfortunately, the problem of finding a term (of Coq) satisfaying a set of *patterns clauses* has not always an answer as shown by this example:

```
Recursive Definition fix :  nat -> nat := n => (fix n).
```

The point is that, in Coq, any term `F` of type `nat -> nat` represents a function which always terminates and it is easy to see that the function `fix` never terminates. This general property of Coq's terms is called *strong normalization*.

The fact that a syntactically correct `Recursive Definition` as the one of `fix` does not correspond to any term in Coq points out that we cannot content with the *syntactical level* of the declaration to obtain a term, but we have to lift up onto the *logical level* to ensure that the aimed function satisfies some strong property as termination. And here comes the difficulty as the termination problem is know to be undecidable in general.

In fact, termination is the main logical property we have to face with. Then the main job of the `Recursive Definition` is to build a proof of `(X1:U1)..(Xk:Uk) T1-> .. -> Tn -> T` understood as a termination proof of the aimed function.
Let's see how it works on the Ackermann's function example :

```
Coq < Theorem Ack : nat -> nat -> nat.
1 subgoal

  ============================
   nat->nat->nat

Ack < Intro n; Elim n.
2 subgoals

  n : nat
  ============================
   nat->nat

subgoal 2 is:
 nat->(nat->nat)->nat->nat

Ack < Intro m; Exact (S m). (* Because of : O m => (S m) *)
1 subgoal

  n : nat
  ============================
   nat->(nat->nat)->nat->nat

Ack < Intros p Ack_n m; Elim m.
2 subgoals

  n : nat
  p : nat
  Ack_n : nat->nat
  m : nat
  ============================
   nat

subgoal 2 is:
 nat->nat->nat
```

```
Ack < Exact (Ack_n (S O)). (* Because of : (S n) O => (Ack n (S O) *)
1 subgoal

  n : nat
  p : nat
  Ack_n : nat->nat
  m : nat
  ============================
   nat->nat->nat

Ack < (* Because of : (S n) (S m) => (Ack n (Ack (S n) m)) *)
Ack < Intros q Ack_Sn_m; Exact (Ack_n Ack_Sn_m).
Subtree proved!

Ack < Save.
(Intros n;Elim n).
(Intros m;Exact (S m)).

(Intros p Ack_n m;Elim m).
Exact (Ack_n (S O)).

(Intros q Ack_Sn_m;Exact (Ack_n Ack_Sn_m)).

Ack is defined
```

One can check that the term `Ack` (*eg* : the above proof of `nat -> nat -> nat`) actually satisfies the intended *patterns clauses* of the Ackermann's function.

Such a proof is *automaticaly* synthesized by a specialized strategy which was originaly designed for the ProPre programming language. It has been adapted for the Coq's framework. For a short description of the ProPre programming language, we refer the reader to [2]. For details concerning the original proof search strategy, we refer the reader to [4]. For theoretical settings of the method involved in ProPre, we refer the reader to [1] and [5].

We list bellow some correct and uncorrect usages of the `Recursive Defintion` feature.

Correct `Recursive Definition`'s

```
Recursive Definition IfSet [X:Set] : bool -> X -> X -> X :=
  true x y => x
 |false x y => y.

Recursive Definition equal_nat : nat -> nat -> bool :=
  O O => true
 |O (S m) => false
```

```
 |(S n) O => false
 |(S n) (S m) => (equal_nat n m).

Recursive Definition less_equal : nat -> nat -> bool :=
  O m => true
 |(S n) O => false
 |(S n) (S m) => (less_equal n m).

(* Needed for the following function *)
Inductive Set bintree :=
  Le : bintree
 |Br : nat -> bintree -> bintree -> bintree.

Recursive Definition ins_bsr [n:nat] : bintree -> bintree :=
  Le => (Br n Le Le)
 |(Br m a1 a2)
  => (si bintree
         (inf_egal n m)
         (Br m (ins_bsr n a1) a2)
         (Br m a1 (ins_bsr n a2))).

(* Needed for the following function *)
Inductive Set list [X:Set] :=
  Nil : (list X)
 |Cons : X -> (list X) -> (list X).

Recursive Definition append [X:Set;ys:(list X)] : (list X) -> (list X) :=
  (Nil X) => ys
 |(Cons X x xs) => (Cons X x (append X ys xs)).

Recursive Definition soml : (list nat) -> nat :=
  (Nil nat) => O
 |(Cons nat O x) => (soml x)
 |(Cons nat (S n) x) => (S (soml (Cons nat n x))).

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

Uncorrect Recursive Definition's

```
Recursive Definition equal_list : (list nat) -> (list nat) -> bool :=
  (Nil nat) (Nil nat) => true
```

```
 |(Nil nat) (Cons nat n y) => false
 |(Cons nat n x) (Nil nat) => false
 |(Cons nat n x) (Cons nat n y) => (equal_list x y).
Coq < Error : in Recursive Definition "equal_list" (equation 4) :
 variable "n" can't have more than one occurence on left of "=>"
```

As explains the error message, a same pattern variable can't be used more than one time.

```
Recursive Definition min : nat -> nat -> nat :=
  O m => m
 |n O => n
 |(S n) (S m) => (S (min n m)).
Coq < Error : in Recursive Definition "min" (equations 2,1) :
 overlapping
```

We do not allow the various left members of *patterns clauses* to overlapp.

```
Recursive Definition wrong_equal : nat -> nat -> bool :=
  O O => true
 |(S n) (S m) => (wrong_equal n m).
Coq < Error : in Recursive Definition "wrong_equal" pattern matching is not exhaustive
```

As we need to prove that the function is totally defined, we need an exhaustive pattern matching of the inputs.

```
Recursive Definition div2 : nat -> nat :=
  O => O
 |(S O) => O
 |(S (S n)) => (S (div2 n)).
Coq < Error : in Recursive Definition "div2" (equation 3) :
 can't find argument on which apply structural induction
```

The strategy makes use of structural induction to search a proof of `nat -> nat`, then it can only accept arguments decreasing in one step (*eg* from `(S n)` to `n`) which is not the case here.

```
Recursive Definition wrong_fact : nat -> nat :=
 n => (IfSet nat (equal_nat n O) (S O) (mult n (wrong_fact (pred n)))).
Coq < Error : in Recursive Definition "wrong_fact" (equation 1) :
 can't find argument on which apply structural induction
```

This error comes from the fact that the strategy doesn't recognize that the `IfSet` is used as matching on naturals and that, when `n` is not `O`, `(pred n)` is less than `n`. Indeed, matching must be explicit and decreasing too.

```
Recursive Definition heap_merge : bintree -> bintree -> bintree :=
  Le b => b
 |(Br n a1 a2) Le => (Br n a1 a2)
```

```
 |(Br n a1 a2) (Br m b1 b2)
  => (IfSet bintree
         (less_equal n m)
         (Br n (heap_merge a1 a2) (Br m b1 b2))
         (Br m (Br n a1 a2) (heap_merge  b1 b2))).
Coq < Error : in Recursive Definition "heap_merge" (equation(s) 1,2,3) ;
 can't find argument on which apply structural induction
```

This failure of the strategy is due to the fact than with simple structural induction one can't get any induction hypothesis for both inductive calls `(heap_merge a1 a2)` and `(heap_merge b1 b2)`.

References

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

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

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

[4] P. Manoury and M. Simonot *Automatizing termination proof of recursilvely defined functions* ; to appear in TCS.

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

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

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