A Tactic Language for the System Coq

David Delahaye*

Project Coq
INRIA-Rocquencourt**

Abstract. We propose a new tactic language for the system Coq, which is intended to enrich the current tactic combinators (tacticals). This language is based on a functional core with recursors and matching operators for Coq terms but also for proof contexts. It can be used directly in proof scripts or in toplevel definitions (tactic definitions). We show that the implementation of this language involves considerable changes in the interpretation of proof scripts, essentially due to the matching operators. We give some examples which solve small proof parts locally and some others which deal with non-trivial problems. Finally, we discuss the status of this meta-language with respect to the Coq language and the implementation language of Coq.

1 Introduction

In a proof[1] system, we can generally distinguish between two kinds of languages: a proof language, which corresponds to basic or more elaborate primitives and a tactic language, which allows the user to write his/her own proof schemes. In this paper, we do not deal with the first kind of language which has been already extensively studied by, for example, John Harrison in a comparative way ([7]), Don Syme with a declarative prover ([11]) and Yann Coscoy with a "natural" translation of proofs ([2]). Here, we focus on the tactic language which is essentially the criterion for assessing the power of automation of a system (to be distinguished from automation which is related to provided tactics). In some systems, the tactic language does not exist and the automation has to be quite powerful to compensate for this lack. For example, this is the case for PVS ([10]) where nothing is given to extend the system. Also, Mizar ([12]), one of the oldest provers, is based on a unique tactic by and it is impossible to automate some parts of the proofs or more generally, some logic theories.

The tactic language must be Turing-complete, which is to say that we must be able to build proof strategies without any limitation imposed by the language itself. Indeed, in general, this language is nothing other than the implementation language of the prover. The choice of such a language has several consequences that must be taken into account:

* David.Delahaye@inria.fr, http://coq.inria.fr/~delahaye/.

** INRIA-Rocquencourt, domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France.

1 The word "proof" is rather overloaded and can be used in several ways. Here, we use "proof" for a script to be presented to a machine for checking.

- the prover developers have to provide the means to prevent possible inconsistencies arising from user tactics. This can be done in various ways. For example, in LCF ([6]) and in HOL ([5]), this is done by means of an abstract data type and only operations (which are supposed to be safe) given by this type can be used. In Coq ([1]), the tactics are not constrained, it is the type-checker which, as a Cerberus, verifies that the term, built by the tactic, is of the theorem type we want to prove.
- the user has to learn another language which is, in general, quite different from the proof language. So, it is important to consider how much time the user is ready to spend on this task which may be rather difficult or at least, tedious.
- the language must have a complete debugger because finding errors in tactic code is much harder than in proof scripts developed in the proof language, where the system is supposed to assist in locating errors.
- the proof system must have a clear and a well documented code, especially for the proof machine part. The user must be able to easily and quickly identify the necessary primitives or he/she could easily get lost in all the files and simply give up.

Thus, we can notice that writing tactics in a full programmable language involves many constraints for developers and more especially for users. In fact, we must recognize that the procedure is not really easy but we have no alternative if we want to avoid restrictions on tactics. However, we can wonder if this method is suitable for every case. For example, if we want a tactic which can solve linear equations on an Abelian field, it seems to be a non-trivial problem which requires a complete programming language. But, now suppose that we want to show that the set of natural numbers has more than two elements. This can be expressed as follows:

$$\vdash (\exists x : \mathbb{N}.\exists y : \mathbb{N}.\forall z : \mathbb{N}.x = z \vee y = z) \rightarrow \ \bot$$

To show this lemma, we introduce the left-hand member of the conclusion (say H) and eliminate it, then we introduce the witness (say a) and the instantiated hypothesis H (say H_a), finally, we eliminate H_a to introduce the second witness (say b) and the instantiation of H_a (say H_b). At this point, we have the following sequent:

$$..., H_b : \forall z : \mathbb{N}.a = z \vee b = z \vdash \ \bot$$

It remains to eliminate H_b with any three natural numbers (say 0, 1 and 2). Finally, we have three equalities (that we introduce) with a or b as the left-hand member and 0, 1 or 2 as the right-hand member. To conclude in each case, it is simply necessary to apply the transitivity of the equality between two equations with the same left-hand member, then we obtain an equality between two distinct natural numbers which validates the contradiction (depending on the prover, this last step must be detailled or not).

Of course, the length of this proof depends on the automation of the prover used. For example, in PVS, it may be imagined that applying the lemma of transitivity is quite useless and assert would solve all the goals generated by the eliminations of H_b. In Coq, the proof would be done exactly in this way and we may want to automate the last part of the proof where we use the transitivity. Unfortunately, even if this automation seems to be quite easy to realize, the current tactic combinators (tacticals) are not powerful enough to make it. So, the user has two choices: to do the proof by hand or to write his/her own tactic, in Objective Caml[2] ([8]), which will be used only for this lemma.

Thus, it is clear that a large and complete programming language is not a good choice to automate small parts of proofs. This is essentially due to the fact that the interfacing is too heavy with respect to the result the user wants to obtain. Moreover, the need for small automations must not only be seen as a lack of automation of the prover because tactics are intended to solve general problems and sometimes, user problems are too specific to be covered by primitive tactics. Thus, it seems that there is a gap between the proof language and the language used for writing tactics.

Here, we want to propose, in the context of Coq, the idea of an intermediate language, integrated in the prover and less powerful than the Turing-complete language for writing tactics, which is able to deal with small parts of proofs we may want to automate locally. This language is intended to be a kind of middle-way where it is possible to better enjoy both the usual language of Coq and some features of the full programmable language.

2 Presentation of the language

2.1 Definition

Currently, the only way to combine the primitive tactics is to use predefined operators called tacticals. These are listed in table 1.

As seen previously, no tactical given in table 1 seems to be suitable for automating our small proof. In fact, we would like to do some pattern matchings on terms and even better, on proof contexts. So, the idea is to provide a small functional core with recursion to have some high order structures and with pattern matching operators both for terms as well as for proof contexts to handle the proof process. The syntax of this language, we call $\mathcal{L}_{tac}$, is given, using a BNF-like notation, by the entry *expr* in table 2, where the entries *nat*, *ident*, *term* and *primitive_tactic* represent respectively the natural numbers, the authorized identificators, Coq's terms and all the basic tactics. In *term*, there can be specific variables like ?n, where n is a *nat* or ?, which are metavariables for pattern matching. ?n allows us to keep instantiations and to make constraints whereas ? shows that we are not interested in what will be matched. We can also use this language in toplevel definitions (Tactic Definition) for later calls.

[2] This is the implementation language of Coq.

$\text{tac}_1;\text{tac}_2$	Applies tac_1 and tac_2 to all the subgoals
tac;[tac_1|...|tac_i|...|tac_n]	Applies tac and tac_i to the i-th subgoal
tac_1 Orelse tac_2	Applies tac_1 or tac_2 if tac_1 fails
Do n tac	Applies tac n times
Repeat tac	Applies tac until it fails
Try tac	Applies tac and does not fail if tac fails
First [tac_1|...|tac_i|...|tac_n]	Apply the first tac_i which does not fail
Solve [tac_1|...|tac_i|...|tac_n]	Apply the first tac_i which solves
Idtac	Leaves the goal unchanged
Fail	Always fails

Table 1. Coq's tacticals

2.2 Semantics

We do not wish to give a formal semantic here. It is not our main aim and would be premature. We can just say that in the context of a reduction semantics (small steps), the interpretation is almost usual. This language can give expressions which are tactics (to apply to a goal) and others which represent terms, for example. Thus, we must evaluate the expressions in an optional environment which is a possible goal. This environment is used for Match Context which makes non-linear first order unification as well as Match. Match Context has a very specific behavior. It tries to match the goal with a pattern (hypotheses are on the left of |- and conclusion is on the right) and if the right-hand member is a tactic expression which fails then it tries another matching with the same pattern. This mechanism allows powerful backtrackings and we will discuss an example of use below.

2.3 Typechecking

This language is not yet typechecked; although this might be useful in the future for at least two reasons. First, we have some ambiguities which must be solved by syntactic means and a consequence is the presence of a quote to mark the application of $\mathcal{L}_{tac}$ (see table 2). Another reason for building a typechecker is that we want to detect statically the free variables in a proof script. Experience of proof maintainability shows that proofs are quite sensitive to naming conventions and the idea is mainly to watch the names of hypotheses. Thus, typechecking will be an interesting and original feature of the language and will allow robust scripts to be built.

2.4 Implementation

To implement $\mathcal{L}_{tac}$, we had to make some choices regarding the existing code. First, we decided to keep an interpreted language. We are not really convinced

```
expr          ::=  expr ; expr
               |   expr ; [ (expr |)* expr ]
               |   atom

atom          ::=  Fun input_fun+ -> expr
               |   Let (let_clause And)* rec_clause In expr
               |   Rec rec_clause
               |   Rec (rec_clause And)* rec_clause In expr
               |   Match Context With (context_rule |)* context_rule
               |   Match term With (match_rule |)* match_rule
               |   '( expr )
               |   '( expr expr+ )
               |   atom Orelse atom
               |   Do (int | ident) atom
               |   Repeat atom
               |   Try atom
               |   First [ (expr |)* expr ]
               |   Solve [ (expr |)* expr ]
               |   Idtac
               |   Fail
               |   primitive_tactic
               |   arg

input_fun     ::=  ident
               |   ()

let_clause    ::=  ident = expr

rec_clause    ::=  ident input_fun+ -> expr

context_rule  ::=  [ (context_hyps ;)* context_hyps |- term ] -> expr
               |   [ |- term ] -> expr
               |   _ -> expr

context_hyps  ::=  ident : term
               |   _ : term

match_rule    ::=  [ term ] -> expr
               |   _ -> expr

arg           ::=  ()
               |   nat
               |   ident
               |   term
```

Table 2. Definition of $\mathcal{L}_{tac}$

that we could save a significant amount of time in the execution of compiled scripts, in general run once, especially if we consider the cost of compilation time. Compared to the previous interpretation core[3], we have made great changes in the main function which executes the tactics, by, for example, adding the new structures we saw previously (see table 2). Also, to be able to deal with substitutions coming from abstracted variables (Fun) and metavariables (Match Context, Match), we interpret the tactic arguments in the main function. The tactics now take already interpreted arguments rather than AST's (Abstract Syntax Trees) coming from syntactical analysis. To be extendable, it is possible to dynamically associate interpretation functions to specific AST nodes.

3 Examples

A first natural example is the one we discussed in the introduction. We want to show that the set of natural numbers has more than two elements. With the current tactic language of Coq, the proof could look like the script given in table 3. As can be seen, after the three inductions (Elim), we have eight cases which can be solved by eight very similar instructions which are possibly different in the equality we cut and the term used to apply transitivity. As we know that this equality, say x=y, is such that there exist the equalities a=x and a=y in the hypotheses, it would be easy to automate this part provided that we can handle the proof context. This can be done by using $\mathcal{L}_{tac}$ and especially, the Match Context structure. Table 4 shows the corresponding script. We can notice that the proof is considerably shorter[4] and this is increasingly true when we add cases (with three, four , ... elements). Moreover, the work is much less tedious than in the case of the proof by hand and the script can be written without the help of the interactive toplevel loop. This results in a proof style which is much more batch mode like.

Another example, a little less trivial, is the problem of list permutation on closed lists. Indeed, we may be faced with this problem when we want to show that a list is sorted and it is quite annoying to do the proof by hand when we know it can be done automatically. To use Objective Caml[5] is certainly quite excessive compared to the difficulty of what we want to solve and $\mathcal{L}_{tac}$ seems to be much more appropriate. To do this, first, we define the permutation predicate as shown in table 5, where ˆ represents the append operation on lists. Next, we can write naturally the tactic by using $\mathcal{L}_{tac}$ and the result can be seen in table 6. We can notice that we use two toplevel definitions PermutProve and Permut. The function to be called is PermutProve which is intended to solve goals of the form ...|-(permut l1 l2), where l1 and l2 are closed list expressions. PermutProve computes the lengths of the two lists and calls Permut with the length if the two lists have the same length. Permut works as expected. If the two lists are equal, it

[3] Of the last release V6.3.1.

[4] In this respect, we can see that the non-linear pattern matching solves the problem in one pattern instead of two successive patterns.

[5] This is the full programmable language to write tactics in Coq.

```
Lemma card_nat: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
Proof.
    Red;Intro H.
    Elim H;Intros a Ha.
    Elim Ha;Intros b Hb.
    Elim (Hb (0));Elim (Hb (1));Elim (Hb (2));Intros.
    Cut (0)=(1);[Discriminate|Apply trans_equal with a;Auto].
    Cut (0)=(1);[Discriminate|Apply trans_equal with a;Auto].
    Cut (0)=(2);[Discriminate|Apply trans_equal with a;Auto].
    Cut (1)=(2);[Discriminate|Apply trans_equal with b;Auto].
    Cut (1)=(2);[Discriminate|Apply trans_equal with a;Auto].
    Cut (0)=(2);[Discriminate|Apply trans_equal with b;Auto].
    Cut (0)=(1);[Discriminate|Apply trans_equal with b;Auto].
    Cut (0)=(1);[Discriminate|Apply trans_equal with b;Auto].
Save.
```

Table 3. A proof on cardinality of natural numbers in Coq

concludes. Otherwise, if the lists have identical first elements, it applies Permut on the tail of the lists. Finally, if the lists have different first elements, it puts the first element of one of the lists (here the second one which appears in the permut predicate) at the end if that is possible, i.e., if the new first element has been at this place previously. To verify that all rotations have been done for a list, we use the length of the list as an argument for Permut and this length is decremented for each rotation down to, but not including, 1 because for a list of length n, we can make exactly $n-1$ rotations to generate at most n distinct lists. Here, it must be noticed that we use the natural numbers of Coq for the rotation counter. In table 2, we can see that it is possible to use usual natural numbers but they are only used as arguments for primitive tactics and they cannot be handled, in particular, we cannot make computations with them. So, a natural choice is to use Coq data structures so that Coq makes the computations (reductions) by Eval Compute in and we can get the terms back by Match.

Beyond these small examples, we discovered that $\mathcal{L}_{tac}$ is much more powerful than might have been expected and, even if it was not our initial aim, this language can deal with non-trivial problems. For example, we coded a tactic to decide intuitionnistic propositional logic, based on the contraction-free sequent calculi LJT* of Roy Dyckhoff ([4]). There was already a tactic called Tauto and written in Objective Caml by César Muñoz ([9]). We observed several significant differences. First, with $\mathcal{L}_{tac}$, we obtained a drastic reduction in size with 40 lines of code compared with 2000 lines. This can be mainly explained by the complete backtracking provided by Match Context. Moreover, we were very surprised to get a considerable increase in performance which can reach 95% in some examples. In fact, this is understandable since $\mathcal{L}_{tac}$ is a proof-dedicated language and we can suppose that some algorithms (such as Dyckhoff's) may be coded very naturally.

```
Lemma card_nat: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
Proof.
    Red;Intro H.
    Elim H;Intros a Ha.
    Elim Ha;Intros b Hb.
    Elim (Hb (0));Elim (Hb (1));Elim (Hb (2));Intros;
        Match Context With
            [_:?1=?2;_:?1=?3|-?] ->
                Cut ?2=?3;[Discriminate|Apply trans_equal with ?1;Auto].
Save.
```

Table 4. A proof on cardinality of natural numbers using $\mathcal{L}_{tac}$

```
Section Sort.

Variable A:Set.

Inductive permut:(list A)->(list A)->Prop:=
     permut_refl:(l:(list A))(permut l l)
    |permut_cons:
        (a:A)(l0,l1:(list A))(permut l0 l1)->(permut (cons a l0) (cons a l1))
    |permut_append:(a:A)(l:(list A))(permut (cons a l) (l^(cons a (nil A))))
    |permut_trans:
        (l0,l1,l2:(list A))(permut l0 l1)->(permut l1 l2)->(permut l0 l2).

End Sort.
```

Table 5. Definition of the permutation predicate

Finally, readibility has been greatly improved so that maintainability has been made much easier (even if there is no debugger for $\mathcal{L}_{tac}$ yet).

We dealt with another important example which was to verify equalities between types and modulo isomorphisms. We chose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and *unit* type (see, for example, [3]). Again, the code, we wrote by using $\mathcal{L}_{tac}$, was quite short (about 80 lines with the axiomatization) and quite readable so that extensions to more elaborated λ-calculi can be easily integrated.

4 Conclusion

We have presented a language ($\mathcal{L}_{tac}$) which is intended to make a real link between the primitive tactics and the implementation language (Objective Caml) used to write large tactics. In particular, it deals with small parts of proofs that

```
Tactic Definition Permut n:=
    Match Context With
        [|-(permut ? ?1 ?1)] -> Apply permut_refl
       |[|-(permut ? (cons ?1 ?2) (cons ?1 ?3))] ->
           Let newn=Eval Compute in (length ?2)
           In
               Apply permut_cons;'(Permut newn)
       |[|-(permut ?1 (cons ?2 ?3) ?4)] ->
           '(Match Eval Compute in n With
                [(1)] -> Fail
               |_ ->
                   Let l0'=(?3^(cons ?2 (nil ?1)))
                   In
                       Apply (permut_trans ?1 (cons ?2 ?3) l0' ?4);
                           [Apply permut_append|
                           Compute;'(Permut (pred n))]).

Tactic Definition PermutProve ():=
    Match Context With
        [|-(permut ? ?1 ?2)] ->
           '(Match Eval Compute in ((length ?1)=(length ?2)) With
                [?1=?1] -> '(Permut ?1)).
```

Table 6. Permutation tactic in $\mathcal{L}_{tac}$

are to be automated. It can be seen that this language has some interesting features:

- it is in the toplevel of Coq. We do not need a compiler or any specification of the implementation of Coq to write tactics in this language. Moreover, to learn this small language would be certainly easier than tackling the manual of the implementation language. Of course, these remarks must be considered with regard to small tactics.
- the code length is, in general, quite short compared to the same proofs made by hand (see tables 3 and 4) and, even when solving non-trivial problems, we still have reductions in size, which are sometimes very impressive (as in the case of Tauto seen previously). Thus, the scripts are more compact and much simpler.
- the scripts are more readable. This is already the case with small proofs but even more so with large tactics (as with Tauto again).
- the scripts are more maintainable, as a direct consequence of the increase in readibility.

It is important to carefully define the scope of $\mathcal{L}_{tac}$ compared to Objective Caml. We must not be tempted to enrich $\mathcal{L}_{tac}$ too much in order to write tactics which are more and more complex. Even if we can at present deal with some complex examples, this must be considered as a bonus and not as a goal. We must make sure that Coq does not draw too much upon Objective Caml and, for the moment, we think that $\mathcal{L}_{tac}$ is complete enough. However, we plan to enable Objective Caml to enjoy the advantages of $\mathcal{L}_{tac}$ by a quotation or a syntax extension. With this system, we could use $\mathcal{L}_{tac}$ in Objective Caml like a true Application Programming Interface (API for short) with specific calls, as seen previously, so that we could write tactics more easily and without any limitation.

From the user point of view, it could be a tricky problem to decide which language is the most appropriate to solve his/her problem. The user must know whether the problem in hand can be coded with $\mathcal{L}_{tac}$. There is no general rule but we can identify several criteria by which Objective Caml must be used rather than $\mathcal{L}_{tac}$. First, $\mathcal{L}_{tac}$ is not suitable for tactics which handle the environment. For example, searching the global context is only possible by using Objective Caml and certain functions of Coq's code. Another indicator that $\mathcal{L}_{tac}$ is not suitable is the use of data structures. The more we use data structures, the more complex the problem is, as is the tactic to build. As shown previously with the example of list permutation (see tables 5 and 6), we can use data structures in $\mathcal{L}_{tac}$ by means of Coq's data structures[6] which can be handled by Match (and possibly Match Context) and the number of data structures we need is a good indication of the difficulty of the tactic we want to write. Moreover, if you are concerned about performances, it is better to use Objective Caml's data structures which are much more efficient than those of Coq. Finally, there are more libraries implementing usual data structures in Objective Caml than in Coq and this may be a decisive argument in some cases. Thus, in general, the use of data structures must be limited in $\mathcal{L}_{tac}$ and the user must make choices. For example, the use of natural numbers in the previous example concerning list permutation seems to be quite reasonable and we may consider that this is also the case for other data structures such as booleans or lists.

References

1. Bruno Barras et al. *The Coq Proof Assistant Reference Manual Version 6.3.1.* INRIA-Rocquencourt, May 2000. http://coq.inria.fr/doc-eng.html.
2. Yann Coscoy. A natural language explanation for formal proofs. In C. Retoré, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Liguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.
3. Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design.* Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.
4. Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, volume 57(3), September 1992.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[6] This can be seen as a step towards a bootstrapped system.

6. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. Edinburgh LCF: a mechanised logic of computation. In *Lectures Notes in Computer Science*, volume 78. Springer-Verlag, 1979.
7. John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.
8. Xavier Leroy et al. *The Objective Caml system release 3.00*. INRIA-Rocquencourt, April 2000.
http://caml.inria.fr/ocaml/htmlman/.
9. César Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Mémoire du DEA d'informatique fondamentale, Université Paris 7, Septembre 1994.
10. Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.
11. Don Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.
12. Andrzej Trybulec. The Mizar-QC/6000 logic information language. In *ALLC Bulletin (Association for Literary and Linguistic Computing)*, volume 6, pages 136–140, 1978.