Electronic Notes in Theoretical Computer Science 70 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 14 pages

A Proof Dedicated Meta-Language [★]

David Delahaye [1]

Programming Logic Group
Department of Computing Science
Chalmers University of Technology
S-412 96 Gothenburg, Sweden

Abstract

We describe a proof dedicated meta-language, called $\mathcal{L}_{tac}$, in the context of the Coq proof assistant. This new layer of meta-language is quite appropriate to write small and local automations. $\mathcal{L}_{tac}$ is essentially a small functional core with recursors and powerful pattern-matching operators for Coq terms but also for proof contexts. As $\mathcal{L}_{tac}$ is not complete, we describe an interface between $\mathcal{L}_{tac}$ and the full programmable meta-language of the system (Objective Caml), which is also the implementation language. This interface is based on a quotation system where we can use $\mathcal{L}_{tac}$'s syntax in ML files, and where it is possible to insert ML code in $\mathcal{L}_{tac}$ scripts by means of antiquotations. In that way, the two meta-languages are not opposed and we give an example where they fairly cooperate. Thus, this shows that a LCF-like system with a two-level meta-language is completely realistic.

1 Introduction

In a LCF-like proof [2] assistant, we can generally distinguish, beyond the *object language* (the logic language), between two kinds of languages: a *proof language*, which corresponds to basic or more elaborate primitives and a *tactic language* (also called *meta-language*), which allows the user to write his/her own proof schemes. 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).

Initially, the first versions of LCF [7] were implemented in Lisp and the meta-language was ML [6]. Currently, with the evolution of ML, which is now a genuine programming language (with numerous variants) and not only

[★] This work has been realized within the LogiCal project (INRIA-Rocquencourt, France).
[1] delahaye@cs.chalmers.se, http://www.cs.chalmers.se/~delahaye/.
[2] 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 meta-language of a proof system, the direct descendants of LCF use ML also as an implementation language. For example, this is the case of HOL or Coq. This evolution of ML has some interesting consequences for the LCF-like proof assistants. Indeed, this fusion between the meta-language and the implementation language allows us to write any tactic, which can have *stronger* and *deeper* interactions with the system. This is a significant evolution, which makes the design of more complex tactics possible[3]. However, the choice of such a meta-language has several consequences that must be taken into account:

- 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 and in HOL, this is done by means of the abstract data type of theorems and only operations (which are supposed to be safe) given by this type can be used. In Coq, the tactics are not constrained, it is the type-checker which, as a Cerberus, verifies that the term (Curry-Howard isomorphism), 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.
- the tactics are not portable (contrary to the first versions of LCF, where the system evolutions did not affect the meta-language layer) and must be maintained.
- the language is quite general-purpose and does not provide proof dedicated procedures in a primitive way.

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 prove expressions of Prestburger's arithmetic (over integers), 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

[3] It is now possible to use all the system functionalities to develop tactics, because the meta-language is not any more only a partial interface for the proof engine.

numbers has more than two elements. This can be expressed as follows:

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

where $\bot$ is the logic notation for false. To show this lemma, we introduce the left-hand member of the conclusion (say H) and eliminate it [4], 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 detailed 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 [5], 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 and that, anyway, this language does not provide appropriate primitives to do so easily. 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.

As it seems that there is a gap between the proof language and the language used for writing tactics, the idea has been then to propose, in the context of Coq [12], an intermediate language, called $\mathcal{L}_{tac}$ [2,3], integrated in the prover and less powerful than the 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

[4] An elimination is the application of an inductive schema.

[5] This is the meta-language and the implementation language of Coq.

programmable language. As this new language is not complete (the aim is to provide a more proof dedicated language, and not necessary another full programmable language), we describe also the design of an interface between the two layers of meta-languages, which can cooperate easily and naturally. This interface has been implemented as a very specific and novel application of the primitive quotation system of Objective Caml [8] (provided by the Camlp4 [10] tool).

2 Presentation of $\mathcal{L}_{tac}$

2.1 Definition

Before the introduction of $\mathcal{L}_{tac}$ in Coq (version V7), the only way to combine the primitive tactics was to use predefined operators called tacticals. These are listed in figure 1 [6].

tac_1 `;` 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`]`	Applies the first tac_i which does not fail
`Solve` `[`tac_1 `|`...`|` tac_i `|`...`|` tac_n`]`	Applies the first tac_i which solves
`Idtac`	Leaves the goal unchanged
`Fail`	Always fails

Fig. 1. Coq's tacticals

As seen previously, no tactical given in figure 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 higher order structures and with pattern-matching operators both for terms as well as for proof contexts to handle the proof process. The whole syntax of this language, called $\mathcal{L}_{tac}$ [2,3], is given, using a BNF-like notation, by the entry <*expr*> in figures 2 and 3, where the entries <*nat*>, <*ident*>, <*term*> and <*primitive_tactic*> represent respectively the natural numbers, the authorized identifiers, 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 meta-variables 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.

[6] Most of these tacticals can be found in other tactic-style theorem provers.

```
<expr>      ::= <expr> ; <expr>
              | <expr> ; [ (<expr> |)* <expr> ]
              | <pre-atom>

<pre-atom>  ::= <expr> <expr>+
              | <atom>

<atom>      ::= Fun <input-fun>+ -> <expr>
              | Let (<let-clauses> And)* <let-clauses> 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>
```

Fig. 2. Syntax of $\mathcal{L}_{tac}$ (1/2).

The pattern-matching operators make non-linear first order unification and have a more specific behavior (compared to the pattern-matching of ML, for example). Indeed, they can perform backtracking. For instance, with a `Match Context`, we try 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

```
<input-fun>      ::=  <ident>
                  |   ()

<let-clauses>    ::=  <ident> = <expr>

<rec-clause>     ::=  <ident> <input-fun>+ -> <expr>

<context-rule>   ::=  [ (<context-hyps> ;)* <context-hyps> |-
                        <pattern> ] -> <expr>
                  |   [ |- <pattern> ] -> <expr>
                  |   _ -> <expr>

<context-hyps>   ::=  <ident> : <pattern>
                  |   _ : <pattern>

<pattern>        ::=  <term>
                  |   [ <term> ]

<match-rule>     ::=  [ <pattern> ] -> <expr>
                  |   _ -> <expr>

<arg>            ::=  ()
                  |   <nat>
                  |   <ident>
                  |   '<term>
```

Fig. 3. Syntax of $\mathcal{L}_{tac}$ (2/2).

a tactic expression which fails then another matching with the same pattern (using a different combination in hypotheses) is tried. This mechanism allows powerful backtrackings and we will discuss an example of use below. It is also possible to perform subterm pattern-matching with patterns of the form `[t]`, which look for subterms matched by `t`. We will see an example in section 3.

2.2 An example

As an example, we can consider the one we discussed in the introduction. We want to show that the set of natural numbers has more than two elements. Using only the primitive tactics and the tacticals of Coq, the proof could look like the following script:

```
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.
```

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:

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

We can notice that the proof is considerably shorter[7] 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

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

can be written without the help of the interactive toplevel loop. This results in a proof style which is much more batch mode like.

2.3 Discussion

Beyond small automations, 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 intuitionistic propositional logic, based on the contraction-free sequent calculi LJT* of Roy Dyckhoff [5]. There was already a tactic called `Tauto` and written in Objective Caml by César Muñoz. 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. Also, readability has been greatly improved so that maintainability has been made much easier[8]. Finally, it is worth noticing that $\mathcal{L}_{tac}$ scripts can be be reused from a version of Coq to another one, because no direct call to implementation functions is made, contrary to tactic codes in Objective Caml.

As another non-trivial example using $\mathcal{L}_{tac}$, we have been able to write the tactic `Field` [4] to solve equalities over (commutative) fields (generating side conditions over the inverses that have been simplified). This tactic has been coded in a total reflexive way directly in the toplevel of Coq, even for the *metaification* step (also called *quotation* or *reification*) of the reflection process, which is traditionally done in Objective Caml. This example has shown how $\mathcal{L}_{tac}$ could be of great help when writing reflexive tactics.

3 Meta-language cooperation

However, even if $\mathcal{L}_{tac}$ is much more expressive than ML to describe tactics, $\mathcal{L}_{tac}$ is not complete and, in particular, $\mathcal{L}_{tac}$ cannot handle the environment of Coq. For example, the original `Tauto` tactic can work with operators, which are isomorphic to the propositional logic connectors. This cannot be done directly using $\mathcal{L}_{tac}$ because this requires to look for objects in the environment and to study their structures, which are essentially low-level operations. Thus, in order to perform these operations, we must use the main meta-language, but we would like to keep also the script written with $\mathcal{L}_{tac}$, which, as seen previously, has some very interesting features. To do so, we have to provide an interface, which allows us to use $\mathcal{L}_{tac}$ syntax directly in Objective Caml files,

[8] There is a debugger dedicated to $\mathcal{L}_{tac}$, which is directly available in the toplevel of Coq. See [12] for more details.

and much more convenient, which allows us to use Objective Caml code in $\mathcal{L}_{tac}$ scripts.

This interface between $\mathcal{L}_{tac}$ and Objective Caml must be quite light (in the use, but also in the implementation) so that it can be worth keeping two levels of meta-languages. To do so, we chose a syntactical way and, in particular, we have defined a system of quotations. This solution is quite natural since quotations are now primitive in Objective Caml with the Camlp4 [10] tool.

3.1 Quotations

To realize quotations in Objective Caml, we use a tool, called Camlp4 [10], which is a Pre-Processor-Pretty-Printer[9] for Objective Caml. With Camlp4, it is possible to define (dynamic) grammars, to make syntax extensions (for Objective Caml, but also for user grammars) and to build easily quotations.

A quotation in Camlp4 is a part of a program enclosed by special parentheses (with `<<...>>`), whose treatment is done by an user function, called a *quotation expander* (if `exp` is a quotation expander, the corresponding quotation will be noted `<:exp<...>>`). A quotation can only appear in position of expression or pattern, which is a strong limitation compared to syntax extensions also possible with Camlp4. There are two kinds of quotations: those returning strings (which must be of Objective Caml syntax) and those returning directly Objective Caml syntax trees (that is why Camlp4 quotations can be considered as primitive quotations of Objective Caml).

Historically, quotations have been introduced in the first versions of ML, to represent the propositions of the LCF logic in a concrete form. Then, they followed the evolution of Edinburgh ML (INRIA, Cambridge), and were still there in Caml. Finally, they have been implemented [9] in a variant of Caml Light, before being available in Objective Caml by Camlp4.

As an example, we can define the type of complex numbers and the corresponding quotation in order to have the usual notation, which is, given $z \in \mathbb{C}$, $z = a + i * b$, with $a, b \in \mathbb{R}$:

```
        Objective Caml version 3.04

# type complex = C of float * float;;
type complex = C of float * float
# #load "camlp4o.cma";;
        Camlp4 Parsing version 3.04

[...]

# <<1.2+i*3.5>>;;
- : complex = C (1.2, 3.5)
```

[9] This explains why p4 in Camlp4.

In the previous script, we define the (concrete) type of complex numbers, called `complex`. Next, we load the Camlp4 tool (file `camlp4o.cma`) in order to be able to parse quotations. Then, we can define the quotation (we have skipped the corresponding code, which is a bit technical and not relevant for our purpose). This consists mainly in building the parser for complex numbers and plugging the quotation expanders into this parser. Finally, we can use the natural notation of complex numbers with `<<...>>` (here, it is useless to prefix with the corresponding quotation expander because we have set the complex number quotation as the default quotation), which produces objects of type `complex` as expected.

In the previous parser, we might want to be able to write arbitrary complicated Objective Caml expressions for a and b in $a + i * b$. To do so, instead of extending the parser with the whole syntax of Objective Caml, an idea is to add a rule for identifiers. These identifiers are called *antiquotations* and will not be parsed in the context of the quotation but in the context of the expression using the quotation. Antiquotations are not a predefined notion of Camlp4, but a programming technique to insert code in quotations. For example, once the antiquotation rule added to our parser, we can write:

```
# let r = sqrt 15. in <<5.+i*r>>;;
- : complex = C (5, 3.87298334621)
```

3.2 Description of the interface

To build the interface between $\mathcal{L}_{tac}$ and Objective Caml, the idea has been to provide a quotation for $\mathcal{L}_{tac}$. This quotation, called `tactic`, is just plugged into the parser entry which recognizes the language of $\mathcal{L}_{tac}$. With this quotation, it is possible to write $\mathcal{L}_{tac}$ scripts (with their usual syntax) in Objective Caml files.

To make the converse insertion, it is a bit more difficult and to do so, we have used antiquotations. But a typing problem arises. The parser entry for $\mathcal{L}_{tac}$ returns Coq syntax trees and this implies that the `tactic` quotation, as well as the associated antiquotations, return also Coq syntax trees. This is a too restrictive condition over the antiquotations, that is to say over the ML code that can be inserted. To overcome this problem, an idea has been to add a *dynamic* node to the Coq syntax tree, where it is possible to insert, *a priori*, any ML code. As dynamics are not primitive in Objective Caml, this has been implemented using the `Obj` module, which allows us to by-pass the type-checker locally. To ensure a correct evaluation, we have provided a system of tags, which allows the tactic interpreter to know how a dynamic node must be executed.

This quotation system has been implemented in the version V7 of Coq and is available as a part of the Coq current distribution. As an example, let us write a tactic which unfolds all the constants of the conclusion of the current goal. This tactic cannot be directly coded using $\mathcal{L}_{tac}$ because we cannot build patterns which match terms as constants. So, we must use Objective Caml, but, as said previously, we do not want to loose the $\mathcal{L}_{tac}$ capabilities. Using the tactic quotation, we could write this tactic in the following way:

```
Welcome to Coq 7.3 (May 2002)

Coq < Drop.
        Objective Caml version 3.04

        Camlp4 Parsing version 3.04

[...]

# let is_constant ist =
    if isConst (List.assoc 1 ist.lmatch) then <:tactic<Idtac>>
    else <:tactic<Fail>>;;

# let unfolds =
    let isc = tacticIn is_constant in
    <:tactic<
      Repeat
        Match Context With
        | [|-[?1]] -> $isc;Unfold ?1>>;;
```

Here, we launch Coq and we go down to the underlying Objective Caml toplevel with the command Drop (this allows to write ML tactics directly under Coq in an interactive way). Next, we setup the Coq environment to write our tactic (we have skipped this part, which consists in loading and opening some Coq implementation modules).

To understand the tactic, we have to describe the function `unfolds` first. In `unfolds`, we embed the `is_constant` function as a dynamic node with `tacticIn` to insert it in the quotation. Then, we use the `tactic` quotation to be able to use $\mathcal{L}_{tac}$'s syntax. The `Match Context` allows us to match every subterm of the conclusion of a goal (pattern `[?1]`). In the corresponding action rule, we test if the matched subterm is a constant by means of the antiquotation `$isc` (in the quotation `tactic`, antiquotations are identifiers prefixed by `$`), which corresponds to the ML function `is_constant`. If this is a constant, this function returns `Idtac`, which allows `Unfold` to be applied to replace the constant by its body. Otherwise, the function returns `Fail`, which makes the matching fail and the `Match Context` catches this failure to backtrack looking for the next subterm.

To perform the test, the function `is_constant` must take the tactic interpretation environment as an argument (`ist`). This environment is a record, which contains, in particular, the list of meta-variable instantiations coming from pattern-matchings (field `lmatch`). As seen in the function `unfolds`, the subterm is bound to meta-variable `?1`, so we look (with `List.assoc`) for the instantiation corresponding to the meta-variable with the number `1`. Next, we verify if this instantiation is a constant or not (with `isConst`). As expected, if this is a constant, we return `Idtac` (using the `tactic` quotation to express the result in concrete syntax), otherwise, we return `Fail`.

Once we have *registered* this tactic with the name `Unfolds`, we can go up to Coq toplevel and we can test it as follows:

```
Coq < Definition def0 := '2'.
Coq < Definition def1 := '2+def0'.
Coq < Definition def2 := '4'.

Coq < Goal def1=def2.
1 subgoal

  ============================
   'def1 = def2'

Unnamed_thm < Unfolds.
1 subgoal

  ============================
   '4 = 4'
```

The parentheses `'...'` are used to provide a concrete syntax for integers. We can notice that all the constants have been unfolded. The left-hand side member has been also reduced due to the tactic `Unfold`, which performs a normalization once the constants have been unfolded.

4 Conclusion

4.1 Achievements

As seen in section 2, $\mathcal{L}_{tac}$ is intended to make a real link between the primitive tactics and the full programmable meta-language (Objective Caml) used to write large tactics. In particular, it deals with small parts of proofs (that are to be automated) but can be also used to build some more complex tactics. This language has also some interesting features, which can be expected from a tactic language. First, it is integrated in the toplevel of Coq. Moreover, the code length is, in general, quite short. Finally, the scripts are more readable, more portable and, as a consequence, more maintainable.

However, as $\mathcal{L}_{tac}$ is not complete, we have presented, in section 3, an interface between $\mathcal{L}_{tac}$ and Objective Caml. This interface is based on a quotation system, which allows us to use directly $\mathcal{L}_{tac}$'s syntax in Objective Caml code, but also to insert Objective Caml code in $\mathcal{L}_{tac}$ scripts by means of antiquotations. The quotation system is implemented using the Camlp4 tool of

Objective Caml and to insert a wide variety of ML functions in $\mathcal{L}_{tac}$ parts, we have simulated a notion of dynamics, which is not primitive in Objective Caml. The global implementation is quite light and we have described an example, which shows how this bridge can be used very easily and naturally.

4.2 Generalization and contribution to LCF

Even if $\mathcal{L}_{tac}$ has been realized in the context of Coq, the approach is quite general and such a proof dedicated language could be also formalized in some other LCF-like theorem provers like HOL or Lego, for example. Indeed, $\mathcal{L}_{tac}$ is based on some abstract principles (pattern-matching over proof contexts, backtracking, ...), which do not use any specific features of Coq and which are also relevant in any other tactic-style proof systems. In particular, this means that this work could be seen as a contribution to the general LCF view where it can be worth handling a two-level meta-language with a full programmable layer and a layer containing elaborated proof primitives to build easily some automations in an abstract way. The situation becomes quite ideal as we know that the two levels can communicate.

However, the implementation technique used to build the bridge between the two levels of such a meta-language could be quite different in some other proof systems depending on the implementation language. In HOL or Lego, for example, it could be done exactly in this way using also primitive quotations provided by Standard ML. But in some other systems, like Isabelle (compiled using Poly/ML) or HOL Light (using Caml Light), the implementation languages do not provide built-in quotations and instead of simulating quotations by means of *ad hoc* filters, it could be interesting to find a more generic embedding technique to deal also with those situations. For that purpose, ideas of MetaML [11] for multi-stage programming or embedding solutions [1] used by the Domain-Specific Language (DSL) community could be of great help.

References

[1] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Department of Computing Science, Chalmers University of Technology, April 2001.

[2] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. `http://logical.inria.fr/~delahaye/biblio.html`.

[3] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système* Coq. PhD thesis, Université Pierre et Marie Curie (Paris 6), Décembre 2001.

[4] David Delahaye and Micaela Mayero. Field: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001.
http://logical.inria.fr/~delahaye/biblio.html.

[5] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, volume 57(3), September 1992.

[6] M. J. C. Gordon et al. A Metalanguage for Interactive Proof in LCF. In *5th POPL*, ACM, 1978.

[7] 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.

[8] Xavier Leroy et al. *The Objective Caml system release 3.04*. INRIA-Rocquencourt, December 2001.
http://caml.inria.fr/ocaml/htmlman/.

[9] Michel Mauny and Daniel de Rauglaudre. A Complete and Realistic Implementation of Quotations for ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[10] Daniel de Rauglaudre. *Camlp4 - Reference Manual version 3.04*. INRIA-Rocquencourt, December 2001.
http://caml.inria.fr/camlp4/manual/.

[11] Walid Taha and Tim Sheard. Multi-stage Programming with Explicit Annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*. ACM, June 1997.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.3*. INRIA-Rocquencourt, May 2002.
http://coq.inria.fr/doc-eng.html.