Article Submitted to Journal of Symbolic Computation

Dealing with Algebraic Expressions over a Field in Coq using Maple[*]

David Delahaye[†1] and Micaela Mayero[‡2]

[1]CPR, CNAM, *Département d'Informatique, Paris, France*
[2]PPS, *Université Denis Diderot (Paris 7), Paris, France*

Abstract

We describe an interface between the Coq proof assistant and the Maple symbolic computation system, which mainly consists in importing, in Coq, Maple computations regarding algebraic expressions over fields. These can either be pure computations, which do not require any validation, or computations used during proofs, which must be proved (to be correct) within Coq. These correctness proofs are completed automatically thanks to the tactic `Field`, which deals with equalities over fields. This tactic, which may generate side conditions (regarding the denominators) that must be proved by the user, has been implemented in a reflexive way, which ensures both efficiency and certification. The implementation of this interface is quite light and can be very easily extended to get other Maple functions (in addition to the four functions we have imported and used in the examples given here).

1. Introduction

Computer Algebra and Theorem Proving are two distinct domains of theoretical Computer Science, which are quite wide and involved in many distinct applications. These two domains have their own scientific communities, which seem to be also rather distinct from each other and have little interaction. Actually, Computer Algebra focuses on (symbolic) computation whereas Theorem Proving focuses on validation and, basically, that does not seem to be related. However, over the last 5 years, the situation has begun to change and some people are increasingly interested in making these two worlds communicate. The fact is that

[†]David.Delahaye@cnam.fr, http://cedric.cnam.fr/~delahaye/.

[‡]Micaela.Mayero@pps.jussieu.fr, http://www.pps.jussieu.fr/~mayero/.

[*]This work has been realized within the Programming Logic Group of the Chalmers University of Technology (Department of Computer Science, Gothenburg, Sweden).

Computer Algebra and Theorem Proving are quite complementary, especially regarding their respective weak points. If we consider a Computer Algebra System (CAS), there is no notion of consistency (we are only interested in the power of computation) and it is quite trivial to perform *false* computations, which obviously have no sense. For example, in the Maple system [8], if we consider the equation $a = 0$, we can derive directly that $1 = 0$ asking for the division and the simplification by a of the equation. Here, the first equation is valid, but we cannot divide it by a because a is equal to 0. Maple does not make any remark and performs the division. Maple supposes that the user knows what he/she is doing and in this trivial example, it seems quite clear that we could have been a little more careful before asking for this division by 0. However, in some more significant examples, this kind of false computations could very easily occur without being noticed by the user and obtaining a wrong result could have more serious consequences. Of course, this is not specific to Maple and could apply to many other CASs. In such a case (as well as in many others), Deduction Systems (DSs) can clearly bring a solution. For example, in the Coq proof assistant [24], it is impossible to derive $1 = 0$ from $a = 0$. Indeed, the division by a is performed by means of a lemma which makes appear explicitely the side condition over a. Thus, a must not be equal to 0 and this cannot be proved.

Conversely, efficient computations are, in general, rather difficult to perform in a proof system. In a DS, it is as easy to crash the system during a *big* computation as it was to derive false propositions in a CAS. As an example, still in Coq, if we try to compute 100×1000, we can notice that it is quite time-consuming (about 20s on a Sun UltraSPARC 450 MHz with a native compiled version of Coq), whereas the computation of 1000×1000 saturates the stack. Again, this is not specific to Coq and this could apply to some other DSs[1]. In the same way as for the previous wrong derivation, CASs can bring a solution to these problematic computations. For example, Maple realizes these two computations immediately (less than a second) on the same machine.

All these examples emphasize that, as said previously, CASs are dedicated to computation whereas DSs are dedicated to validation. But, these examples also show that these domains are complementary because computation (resp. validation) is clearly a weak point of DSs (resp. CASs). So, there is a real interest to be gained in making them interact. This could be done in several ways: to import validation in CASs, to import computation in DSs or, more ambitiously, to build a unique system with both (efficient) computation and validation. In this paper, we focus on the second way, i.e. to import computation in DSs, and we want to share our experience of an interface between Coq and Maple. This idea of interface was proposed some time after the design of the tactic `Field` [12] for Coq, which can automatically prove equalities over fields. Beyond the fact

[1]Actually, in Coq and in most of DSs, there exist more efficient numbers (based on a binary coding) which can be seen as dedicated to computations. However, the recursion scheme is not the usual one and the proofs are less *natural* with these numbers (which are not really appropriate for reasoning).

that this tactic has considerably improved the possibility of Computer Algebra development in Coq, it has also made it possible to import computations from some external CASs (not only Maple) and to validate them automatically, so that it is almost transparent for the user, who has the impression of performing only a computation. Thus, with this interface, we can use, in Coq, some Maple functions regarding expressions over fields and the computations coming from these functions are systematically proved (to be correct) by the tactic `Field`. This work presents a concrete realization and thus contributes to showing the effectiveness of general methods describing the interaction between CASs and DSs (see [4], for example). It should be noted that up to now, in this specific context of CAS/DS, very few interfaces have been designed or at least, formally presented in publications. Compared to some of these interfaces, the main novelty of this paper consists in providing the corresponding automation in the DS to *automatically* prove the external computations from the CAS. Indeed, we claim that such a method is worth being applied in practice only if the automation (at least partial) to verify the correctness of the imported computations has been also developed.

2. The computation/proof paradigm

2.1. Computation vs proof

In Mathematics, there are two distinct traditions: an operational tradition and a denotational tradition. For example, if we want to show that $2 + 2 = 4$, the question does not seem to be the same according to the two traditions. In an operational way, the previous equation is actually not symmetric and this means that we have to compute (or reduce) $2 + 2$ to 4. In a denotational way, the equation is really symmetric and this is a proposition that we have to prove using the axioms of Arithmetics. Historically, Mathematics of Antiquity, coming from Egyptian, Babylonian or Greek knowledge, was centered around operational methods and, little by little, these methods have been replaced by the denotational view of Modern Mathematics. However, nowadays, with the growing development of Computer Science, we can no longer ignore the operational tradition.

Thus, in a *modern* mathematical language, we must be able to show that $2 + 2 = 4$ but also to transform the expression $2 + 2$ into the expression 4. So, this language must not only provide deduction rules, but also computation rules, which are called *rewriting rules*. One of the simplest rewriting rules, which is, in general, implicitly used, is the β-reduction, which simply consists in replacing the formal arguments by their effective arguments. For example, if we apply the function $x, y \mapsto x + y$ to 3 and 4, we obtain the expression $(x, y \mapsto x + y)(3, 4)$, which is computed by replacing x by 3 and y by 4 to get the expression $3 + 4$. If we want to further reduce the expression $3 + 4$, we need other computation rules (regarding the symbol $+$, in particular). Currently, one of the most interesting theoretical frameworks which allows us to handle both deductions and computations is certainly the *deduction modulo* [13]. In this formalism, thanks to a congruence on propositions, deductions (the undecidable part) and computations (the decidable part) are clearly separated in a clean way. However, as far as the authors are aware, no proof system, based on such a theory, has yet been implemented. In the current other proof systems, the user has to apply explicit computation rules when building proofs and it is more difficult to distinguish computations from deductions.

2.2. Importing computation

In general, especially compared to programming languages, computations are a weak point of proof systems, even if, in these tools, computations are realized considerably faster than deduction checkings, i.e. means which allow us to build a proof tree structure w.r.t. the underlying logic. This can be explained by the fact that proof systems use essentially a pure functional specification language. In particular, it is impossible to handle imperative features like, for example, mutable objects or exceptions, which can be of great help when writing efficient code. Moreover, to ensure consistency, proof systems must only deal with functions which terminate. In some systems, the termination argument must be explicitly given (as in PVS [19]) and in some others, the syntactical class of functions are constrained in order to get the proof termination automatically (as in Coq [24]). In every case, the user is a bit limited in the way of writing his/her code in a language which is clearly not Turing-complete. In the same way, it is quite difficult to handle partial functions, which can only be simulated.

Actually, all the previous features express that we cannot add any computation rule to a logic system. We have to take care of the consistency of the system because these rules are used not only for pure computations but also in proofs. However, this is a real limitation because we may want to perform computations outside a proof and, in this case, there is no special need for restriction regarding the computation rules. To do so, a natural idea is to call external functions (w.r.t. the proof system) which are only used for pure (i.e. not in a proof) and local computations. In this case, the proof system is only interested in the result and not in the function which makes the computation.

This method is quite general and the proof system can use functions from any external tool. Moreover, the proof system only needs a very superficial knowledge of the external tool in order to build the interface. In particular, this can be applied to Computer Algebra which can provide very efficient procedures to any proof system whereas it is quite useless, for the proof system developers, to know how these procedures actually work. Thus, this principle can easily build bridges between Theorem Proving and other very different domains, which may, in turn, lead to a new computational behavior.

2.3. Validating computation

The previous method is a general idea to perform external computations in a proof system but this cannot be used inside proofs. As said previously, to extend these computations to proofs, we have to ensure that they do not break the consistency of the proof system. To do so, we have to prove that every external computation is correct w.r.t. the computation and deduction rules of the proof system. This idea is currently well known and for example, in the context of CA, [4] essentially proposes two approaches regarding this process of validation: a *believing* approach, where the correctness of the external computation is assumed, i.e. added as an axiom, and a *skeptical* approach, where we have to establish the correctness of the computation. Actually, there is also a third approach, called the *autarkic* approach, where the proof assistant makes the computation on its own. This last approach will not be discussed in this paper since we want to import computations. As claimed in [3], the believing approach is unsatisfactory because CASs may have some bugs but especially, as can be seen in the introduction, some necessary side conditions may not be required. Thus, we will have a clearly skeptical view in this paper. More precisely, let us consider the following sequent:

$$\Gamma \vdash P(t)$$

where Γ is a context of hypotheses, P is a predicate and t is a term. If we want to transform t using an external function f_{ext}, we have to use the contextual rule of the equational logic, i.e.:

$$\frac{\Gamma \vdash t = t' \qquad \Gamma \vdash P(t')}{\Gamma \vdash P(t)}\ (=_{\mathsf{Cont}})$$

where $=$ is an equality (it could be either the syntactical equality, also called Leibniz's equality, or more generally a setoid equality) and $t' = f_{\mathsf{ext}}(t)$. Once this rule has been applied, we can go on to the proof with the sequent $\Gamma \vdash P(t')$, but we have also to prove that $t = t'$, i.e. $t = f_{\mathsf{ext}}(t)$, which is the required validation. To do so, there are two alternatives. Either there is a function of the proof system, called g_{prf}, s.t. we can obtain, by computation $g_{\mathsf{prf}}(t) = f_{\mathsf{ext}}(t)$ and conclude using the reflexivity axiom of the equality, but in this case, the use of f_{ext} is completely useless because we can use g_{prf} directly to transform t (as said previously, we are not interested in autarkic computations in this paper). Or there is no function of the proof system which is extensionally equal to f_{ext} and we have to prove $t = t'$ using the deduction rules. As already mentioned, the first alternative will generally not occur because the proof system will call external functions which do not already belong to its environment. The second option is more realistic and shows very clearly the duality computation/proof. Indeed, here, we make an external computation with $f_{\mathsf{ext}}(t)$ and also a corresponding proof $t = f_{\mathsf{ext}}(t)$.

Regarding the proof of $t = t'$, it would be particularly interesting if it could be completed automatically, even partially, so that the user really has the impression of making a computation using an external function. However, even if this proposition cannot be automatically proved, it is always worth calling the external function which returns t' because, in every case, we use the external tool as an *oracle* which provides a result.

2.4. Application to Coq and Maple

As said previously, even if it is easier, in a proof system, to verify that a computation is correct than to make the computation itself, it is quite important to automate this verification so that the user only sees the computation side of this paradigm. Thus, this restricts the application of Theorem Proving to domains which are decidable or at least partially decidable. Here, in the context of the Coq proof assistant [24] (a direct descendent of LCF, one of the first proof systems), we have designed a strategy (or a *tactic*), called `Field` [12], for reasons which will be explained in section 3, which automatically proves equalities between algebraic expressions over a field. As the problem is not decidable in general, the tactic generates some conditions (typically that some expressions occurring in inverses must not be equal to 0), that the user must prove manually. Even though it was not one of the initial goals, this tactic has built quite a direct bridge toward Computer Algebra Systems, which can perform various symbolic computations over algebraic expressions, because it is now possible to call any computation procedure of a CAS to get a result and to verify the correctness of this result with this new tactic. As a Computer Algebra System, we have chosen Maple [8] because, beyond the fact that Maple is both popular and easy to use, at least for a beginner, it provides all the functions we would like to have in Coq and it allows us to implement very quickly a light interface between the two systems (see section 4). We will see that this cooperation between Coq and Maple makes usual but also certified operations over algebraic expressions possible very easily (the list of available operations can be extended quite quickly by the user) and opens up, as a bonus, new possibilities regarding the automation of Coq in the domain of Computer Algebra.

3. Presentation of the tactic `Field`

Before describing the interface between Coq and Maple, let us focus on the tactic `Field`, which has made it possible to increase the automation of Coq in the domain of Computer Algebra.

3.1. History and motivations

Initially, the tactic `Field` [12] was designed to automate many *small* parts of proofs over the real numbers using the field structure. These parts of proofs were small in the sense that they were quite trivial when considered from the usual and informal mathematical point of view, but they turned out to be quite tedious to build in a formal proof system. In particular, the idea was to deal with the ε/δ proofs[2] involved in the theorems about limits and derivatives. As a typical example, let us consider the derivative of the addition: given two functions f and g, as well as their derivatives at x_0, denoted $f'(x_0)$ and $g'(x_0)$ (we assume that these two functions are both side differentiable at x_0). By definition, the two derivatives are expressed as follows:

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$g'(x_0) = \lim_{x \to x_0} \frac{g(x) - g(x_0)}{x - x_0}$$

Using the theorem of limit addition, we obtain directly:

$$f'(x_0) + g'(x_0) = \lim_{x \to x_0} \left(\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} \right)$$

Using the limit definition[3], we have, for every domain D of $\mathbb{R}$:

$$\forall \varepsilon > 0,\ \exists \delta > 0,\ \forall x \in D \backslash x_0,\ \text{if } |x - x_0| < \delta \text{ then}$$

$$\left| \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} - (f'(x_0) + g'(x_0)) \right| < \varepsilon$$

Finally, we must show the following equality:

$$\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} = \frac{f(x) + g(x) - (f(x_0) + g(x_0))}{x - x_0}$$

The equality above is quite trivial but the formal proof requires much more work than expected. Indeed, we must reduce the left-hand side with the same denominator and then we can show the equality over the numerators using distributivity, commutativity, etc. In Coq, 10 rewritings are needed to complete this proof.

Initially, the tactic `Field` was implemented to deal with that kind of proofs over the real numbers. Next, the tactic was generalized to handle every field (not only $\mathbb{R}$, but also $\mathbb{C}$ for example). The tactic is currently available in the latest version of Coq (V7.3).

[2]ε/δ proofs are proofs using the explicit definition of limit, i.e. of the form: $\forall \varepsilon > 0. \exists \delta \ldots$.

[3]Here, we have to get to the ε/δ level even if it seems sufficient to perform the algebraic manipulations in the argument of the limit because in the formalization made in Coq, the x occurring in the limit is an abstract variable and it is not allowed to rewrite *under* an abstraction.

3.2. Algorithm

The algorithm is based on the fact that there is already a decision procedure for Abelian rings in the Coq system (tactic `Ring` [5; 6]). So, the idea is to minimize the simplification operations in order to be plugged into the decision procedure on Abelian rings as soon as possible. This essentially means that we simply have to get rid of all the inverses involved in the equality to be solved.

To do so, we propose the following steps:

1. To transform the expressions $x - y$ into $x + (-y)$ and x/y into $x * 1/y$.
2. To look for the inverses involved in the equality in order to build a product of these inverses.
3. To perform a full distribution in the left-hand side and the right-hand side of the equality, except in the inverses.
4. To associate to the right each monomial, except in the inverses.
5. To multiply the left-hand side and the right-hand side of the equality by the product of inverses (built in step 2), generating the side condition that all the inverses must not be equal to 0.
6. To distribute only the product of inverses on the sum of monomials in the left-hand and right-hand side without re-associating to the right.
7. To simplify the inverses in the monomials using the field rule $x*1/x = 1,\ \text{if } x \neq 0$ and performing permutations of the monomial components if necessary, that is to say if there are some inverses remaining and that the field rule cannot be applied[4].
8. To loop to step 2 if there are some inverses remaining.

The first step allows us to limit the operations we have to deal with (we get rid of the binary minus and the division which are not primitive in the field definition) before reducing the problem to the ring level. Step 4 is clearly not necessary, but rather more efficient, because this avoids a double recursive call in the functions which handle these expressions. The last step, which may involve further iterations, is justified by the possibility of having other inverses in the inverses. After all these steps, we obtain an expression without inverses and we only have to call the decision procedure for Abelian rings to conclude. The correctness and the complexity of this algorithm will be discussed later in subsection 3.4.

3.3. A complete example

Let us consider a small example of the evaluation of the previous algorithm where it is needed to make a recursive call. Given x, y and z, three variables over a field, we propose to show the following equality under the hypotheses that $x \neq 0$, $y \neq 0$ and $z \neq 0$:

[4]Here, we do not have to verify that $x \neq 0$, because this condition has already been generated during the step of multiplication by the product of all the inverses.

$$\frac{1}{x} * (x - \frac{x}{\frac{y}{z}}) = 1 - \frac{z}{y}$$

First, we transform the binary minus and the divisions:

$$\frac{1}{x} * (x + (-x) * \frac{1}{y * \frac{1}{z}}) = 1 + (-z) * \frac{1}{y}$$

We build the product of inverses, we call p:

$$p = x * ((y * \frac{1}{z}) * y)$$

We perform a full distribution in the left-hand and right-hand sides of the equality, except in the inverses:

$$\frac{1}{x} * x + \frac{1}{x} * ((-1) * x * \frac{1}{y * \frac{1}{z}}) = 1 + (-1) * z * \frac{1}{y}$$

We associate to the right each monomial, except in the inverses:

$$\frac{1}{x} * x + \frac{1}{x} * ((-1) * (x * \frac{1}{y * \frac{1}{z}})) = 1 + (-1) * (z * \frac{1}{y})$$

We multiply the left-hand side and the right-hand side of the equality by p generating a correctness condition:

$$\begin{gathered} x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * x + \tfrac{1}{x} * ((-1) * (x * \tfrac{1}{y * \frac{1}{z}}))) = \\ x * ((y * \tfrac{1}{z}) * y) * (1 + (-1) * (z * \tfrac{1}{y})) \end{gathered}$$

with $x * ((y * \frac{1}{z}) * y) \neq 0$.

We distribute this product on the monomials without re-associating to the right:

$$\begin{gathered} x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * x) + x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * ((-1) * (x * \tfrac{1}{y * \frac{1}{z}}))) = \\ x * ((y * \tfrac{1}{z}) * y) * 1 + x * ((y * \tfrac{1}{z}) * y) * ((-1) * (z * \tfrac{1}{y})) \end{gathered}$$

We simplify the inverses performing permutations if necessary:

$$(y * \frac{1}{z}) * y * x + y * ((-1) * x) = x * ((y * \frac{1}{z}) * y) * 1 + x * y * (-1)$$

Some inverses are remaining (two occurrences of $\frac{1}{z}$) and we have to apply the previous steps again. The new product of inverses, which we call p', is the following:

$$p' = z$$

Here, each member of the equality is already fully distributed and there is no need to apply step 3. Next, we associate to the right each monomial:

$$y * (\frac{1}{z} * (y * x)) + y * ((-1) * x) = x * (y * (\frac{1}{z} * (y * 1))) + x * (y * (-1))$$

We multiply both sides of the equality by p' generating another side condition:

$$z * (y * (\frac{1}{z} * (y * x)) + y * ((-1) * x)) = z * (x * (y * (\frac{1}{z} * (y * 1))) + x * (y * (-1)))$$

with $z \neq 0$.

We distribute p' on the monomials without re-associating to the right:

$$z*(y*(\frac{1}{z}*(y*x)))+z*(y*((-1)*x)) = z*(x*(y*(\frac{1}{z}*(y*1))))+z*(x*(y*(-1)))$$

We simplify the inverses:

$$y * (y * x) + z * (y * ((-1) * x)) = x * (y * (y * 1)) + z * (x * (y * (-1)))$$

Thus, we obtain an equality over an Abelian ring structure, which can be solved calling the corresponding decision procedure. We have also two side conditions (there are as many conditions as recursive calls of the algorithm), $p \neq 0$ and $p' \neq 0$, which have to be proved manually (more precisely, they are not proved automatically by the algorithm).

3.4. Implementation

An originality of the tactic `Field` is that it has been implemented in a reflexive way. This particular coding ensures both efficiency and soundness. To fully understand how this is possible, let us briefly recall what a reflection is.

To code a tactic in a formal proof system, there are globally two possible options. An explicit coding using rewriting or a reflexive coding using reduction. An explicit coding, also called LCF's approach, may be very inefficient due to the use of rewriting, which may be quite time-consuming but also space-consuming in some systems based on Curry-Howard's isomorphism[5], where proofs are λ-terms, like in Coq, Lego [21] or Alfa [11], for example. A reflexive coding is an alternative method, which is quite satisfactory according to these two criteria. Indeed, rewritings are replaced by more efficient reduction steps and proof term size is the same as the size of the goals to be proved.

[5]This isomorphism consists in the fact that it is possible to build a bijection between propositions and types of a typed λ-calculus, which implies a second bijection between proofs and λ-terms.

The reflection principle is the following: given a language of concrete terms (typically any type of the system), say C, and a language of abstract terms (typically an inductive type), say A. As it is not possible, in general, to handle the terms of C as we would like to (we may not be able to perform pattern-matching, for example), the idea is to reflect a part of C into the language A, which is supposed to be isomorphic to this part. The first step, called metaification[6], consists in converting terms of C into terms of A. More precisely, this consists, for a term c of C, in building the term a of A such that $[\![a]\!]_v = c$, where $[\![.]\!]$ is the interpretation function from A to C (which can be coded in the system), v is the canonical map of the parts of C which are not reflected and $=$ is the syntactical equality.

Next, we can handle the converted terms of A and we can write some transformation functions on A. To use these functions, we only have to prove correctness lemmas, which, for example, given a transformation function φ from A to A, have the following form: $\forall a \in A.[\![\varphi(a)]\!]_v = [\![a]\!]_v$. In particular, this point means that, in the reflection process, the tactic and its correctness proof cannot be dissociated, they are built simultaneously.

Finally, once this lemma has been applied, we only have to fully reduce the expression in order to perform the transformation on the abstract term of A and to re-obtain a concrete term of C.

The situation can be summarized by figure 1 and here is a small example to better understand how this general scheme of reflection actually works:

$$\begin{tikzcd}
C \arrow[r, "\text{Metaification}"] \arrow[d, "\text{Explicit coding}"'] \arrow[d, bend left=90, "\text{Reflection}"] & A \arrow[d, "\varphi"] \\
C & A \arrow[l, "[\![\cdot]\!]"]
\end{tikzcd}$$

Figure 1: Reflection principle.

Example 3.1 (Handling propositions): *The probably best example of objects which can only be handled at the meta-logic level are basic propositions. For*

[6]This word was coined by Samuel Boutin [6], but, in the literature, this step is also called quotation or reification.

instance, let us consider the strategy which replaces $\Phi \wedge \top$ with Φ, where Φ is a proposition and $\top$ is the universally true proposition. To write such a strategy according to a reflective approach consists in reflecting a part of propositions, actually $\wedge$ as well as $\top$, using a concrete type (an inductive type). Let us call T this concrete type, which is made of three constructors: and *corresponding to $\wedge$,* true *to $\top$ and a last one to variables (these variables are mapped to propositions which are not reflected, i.e. propositions formed from $\vee$, $\Rightarrow$, etc). Now, the strategy can be written as a function, we call $\mathcal{S}$, over terms of T; this is a quite trivial recursive function working by usual pattern-matching. To be applied, this strategy needs its correctness lemma (that is one of the main points of reflection's principle: the correctness of the strategy is established during the implementation process), which is:*

$$\forall t \in T. [\![\mathcal{S}(t)]\!]_v = [\![t]\!]_v$$

where $[\![.]\!]$ is the interpretation function of abstracted propositions of T into usual propositions (as $\mathcal{S}$, this is also a quite trivial recursive function working by usual pattern-matching) and v is the canonical map of the parts which have not been reflected during the metaification pass. Here, since we use equality between propositions, we must remark that we need an extensionality axiom to prove this lemma.

If we consider the expression $\Phi \wedge \top \Rightarrow \Phi$, where Φ is an arbitrary complicated proposition, let us see how to apply the previous strategy. First, we have to metaified the expression (this is the only pass which requires to switch to the meta-logic level) and we obtain:

$$[\![\mathsf{and}(p, \mathsf{true})]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

Next, we apply the correctness lemma of $\mathcal{S}$ (a simple rewriting):

$$[\![\mathcal{S}(\mathsf{and}(p, \mathsf{true}))]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

We can apply the definition of $\mathcal{S}$:

$$[\![p]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

Finally, we apply the definition of $[\![.]\!]$ to get $\Phi \Rightarrow \Phi$.

For further information regarding the use of reflection, see [5; 6; 15; 22; 18].

Thus, with this very specific implementation, the proofs using `Field` can be verified faster using only reductions and are smaller. Moreover, this reflexive coding ensures the correctness of the tactic and the built procedure is then *certified.* It is important to realize how this last point is fundamental because, with a direct coding, it is not even possible to state the correctness *inside* the proof system (indeed, this is a meta-statement which can only be verified outside the system). Here, we provide an efficient but also valid tactic. For further details regarding the correctness lemmas and the complexity of `Field`, see [12].

4. Interfacing Coq and Maple

4.1. Method

According to the paradigm described in section 2, we have imported, into Coq, computations from Maple to build terms, in definitions for example, but also to be used in proofs of propositions, called *goals*. Pure computations, i.e. not used in proofs, can be realized with the following expression (we will see some examples in section 5):

`Eval` $<$ Maple *function* $>$ `in` $<$ Coq *term* $>$

which simply consists in applying the Maple function to the Coq term and returning a new Coq term. Obviously, there is interfacing work to do between Coq terms and Maple terms, which will be described in subsection 4.2. As said in section 2, computations of this kind do not require any verification from Coq (that is why they are called pure computations) and no tactic is called to build any proof (even if the Maple function does not do what it was coded for, it cannot break the consistency of Coq).

Computations which occur in proofs are called with the usual syntax of Coq's tactics as follows:

$<$ Maple *function* $>$ $<$ Coq *term* $>$.

where we assume that we are in the proof editing mode and that the Coq term has an occurrence in the goal we try to prove. If we call f, the Maple function, and t, the Coq term, the effect of this "Maple tactic" on a goal of the form $\Gamma \vdash P(t)$ is to generate a subgoal $\Gamma \vdash P(f(t))$, the user has to complete next, and a subgoal $\Gamma \vdash t = f(t)$, to verify, as seen in section 2, that the Maple function has given a correct result. The latter subgoal is automatically solved by the application of the tactic `Field` since, as said in subsection 2.4, we deal with Maple functions which handle algebraic expressions over fields. As seen in section 3, applying `Field` may generate side conditions (typically, that some expressions must not be equal to 0) and, after applying this Maple tactic, the user may have to prove more than one subgoal. Figure 2 gives a global view of how Maple computations are used in Coq proofs.

The Maple functions dealing with algebraic expressions over fields which have been exported are the following (we also give their informal and brief semantics coming from Maple's documentation):

Maple function	Action
`simplify`	Apply simplification rules to an expression
`factor`	Factorize a multivariate polynomial
`expand`	Expand an expression
`normal`	Normalize a rational expression

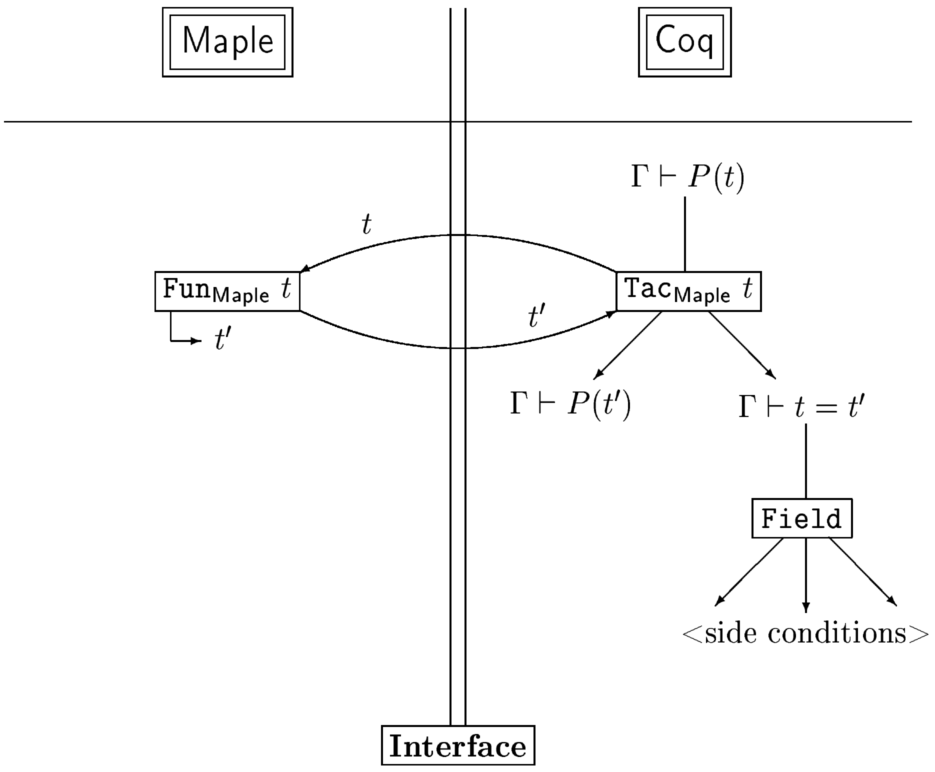

Figure 2: Using Maple computations in Coq proofs.

In section 5, we will see some examples which give a more precise idea regarding the behavior of these functions, but for further information, the reader can refer to the Maple reference manual [8]. Syntactically, these functions are imported in Coq with the same names where the first character has been capitalized giving `Simplify`, `Factor`, etc. Currently, we only have functions with an arity of 1 (we will see in subsection 4.2 that adding other functions is quite direct and very easy for the user), but there is no specific difficulty in extending this interface to functions with higher arities.

4.2. Implementation

Implementing the interface between Coq and Maple has been done quite quickly and almost for free, essentially due to the reflexive coding of the tactic `Field`. When we want to transform a Coq term t with a Maple function, we use the reflexive structure of field which has been created for `Field` and we perform a metaification (see figure 1) to get a term t_{abs}. With this new term t_{abs}, which is only an algebraic expression over a field, we build another concrete term t_{Maple}, expressed in the Maple syntax and which is completely isomorphic to t_{abs}. Next, we can call the corresponding Maple function to obtain a term t'_{Maple}, which can be directly transformed into a term t'_{abs} of the reflexive structure. Finally, the

t'_{abs} is interpreted by the function $[\![.]\!]$ of figure 1 to re-obtain a Coq term t', which can be used either as the result of a pure computation or in a proof.

To use the previous result, i.e. t' coming from t, in a proof, we simply make a cut of the equality $t = t'$ and we rewrite t into t' in the goal where t occurs. The equality $t = t'$ is then proved by `Field`. This is done by the following short Coq script:

```
Replace t with t';[Idtac|Field].
```

where the tactic `Replace` makes the cut of `t=t'` (we obtain two subgoals) to rewrite `t` into `t'` in the main subgoal, and the tactic `Idtac` leaves the goal unchanged. The notation `Tac;[Tac`$_1$`|...|Tac`$_n$`]` means that we apply the tactic `Tac` to the current goal, `Tac`$_1$ to the first subgoal, ..., and `Tac`$_n$ to the `n`th subgoal. Here, the first subgoal is the initial goal where `t` has been replaced by `t'`, which is left identical by `Idtac` (this is the expected subgoal, which must be completed by the user afterwards), and the second subgoal is the equality `t=t'`, which is automatically proved by `Field`.

The whole implementation has been realized in a very light way. The interface between Coq and Maple does not require a specific architecture and, in particular, there is no transfer protocol, no use of sockets or other sophisticated communication systems, but only a simple pipe created by Coq for each call of a Maple function. Of course, neither of the two systems has any information regarding the state of the other one, but this is quite useless and when Maple is called, there is no need for a specific context. Incidentally,we can also notice that the roles of the two tools are asymmetric. Indeed, Coq is in charge and deals with the interface. Maple is only called when Coq needs a computation, but Maple cannot call Coq to verify a result in its own context.

The implementation of this interface is available as the Coq contribution Chalmers/MapleMode, which can be found on the Coq web site: http://coq.inria.fr. As just said, the implementation is quite light and the corresponding code is very short with about 300 lines of ML. The contribution contains also some examples of use for each imported Maple function.

5. Examples

Here, we give some examples of the new tactics `Simplify`, `Factor`, `Expand` and `Normal`, now available in Coq. As seen previously, these tactics call the corresponding Maple functions and we propose some examples where they can be used either to prove a goal or only to compute a result. These examples are rather small in order to show quite clearly how the new corresponding tactics work and in particular, to understand when some side conditions are generated. A more complete and realistic example can be found in appendix A.

To use Maple in Coq, we have to import the specific module `Maple` (the interface) with the Coq command `Require`. If Maple is available on the machine where Coq is running then the following heading appears:

```
Welcome to Coq 7.3 (May 2002)

Coq < Require Maple.
[Loading ML file maple.cmo ...
Coq is now powered by Maple
[Maple V Release 5 (Chalmers Tekniska Hogskola)]

    |\^/|              v
._|\|    |/|_.  ====>  <O___,,
 \  MAPLE  /    ====>   \VV/
 <____ ____>             //
       |
done]
```

The examples we give are over the field of real numbers. We suppose that the module `Reals` has been imported, which allows us to use the whole real number library and, in particular, to call the tactic `Field` instantiated for this field. This module has a specific syntax and the real expressions are parenthesized by ``‘‘``. The opposite of x, the inverse of x and $x \neq y$ are respectively noted `-x`, `/x` and `x<>y`. Additionally, there are also a binary minus and a division, which are represented by `-` and `/`.

5.1. Maple tactics

5.1.1. Simplify

We want to prove the following lemma, we call `simp1`, stating that for every real number x and y, if $x \neq 0$ and $y \neq 0$ then:

$$(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1 > 0$$

After having entered the goal and assumed the hypotheses (tactic `Intros`), we obtain the following goal:

```
simp1 < Intros.
1 subgoal

  x : R
  y : R
  H : ‘‘x <> 0‘‘
  H0 : ‘‘y <> 0‘‘
  ============================
   ‘‘(x/y+y/x)*x*y-(x*x+y*y)+1 > 0‘‘
```

The most direct way of completing this goal is to simplify the left-hand side term $(\frac{x}{y}+\frac{y}{x})\ x.y-(x.x+y.y)+1$ with the new tactic `Simplify` as follows:

```
simp1 < Simplify ``(x/y+y/x)*x*y-(x*x+y*y)+1``.
2 subgoals

  x : R
  y : R
  H : ``x <> 0``
  H0 : ``y <> 0``
  ============================
   ``1 > 0``

subgoal 2 is:
 ``y*x <> 0``
```

The tactic `Simplify` calls the Maple function `simplify` to get the simplified term `1` and replaces it in the goal to obtain `1 > 0`. We can notice that another subgoal appears. This subgoal is generated by applying the tactic `Field`, which is called to validate the previous replacement by proving the equality between the initial and simplified terms, i.e. $(\frac{x}{y}+\frac{y}{x})\ x.y-(x.x+y.y)+1=1$. This automatic proof creates an additional goal requiring that the denominator product must not be equal to 0, that is to say $y\times x\neq 0$. These two subgoals are then completed by very simple tactics on real numbers, respectively `Sup0` and `SplitRmult`.

5.1.2. `Factor`

Regarding the tactic `Factor`, we can give the following example in which we want to factorize the term:

$$a^3+3a^2b+3ab^2+b^3$$

```
fact1 < Intros.
1 subgoal

  a : R
  b : R
  H : ``a+b > 0``
  ============================
   ``a*a*a+3*a*a*b+3*a*b*b+b*b*b > 0``

fact1 < Factor ``a*a*a+3*a*a*b+3*a*b*b+b*b*b``.
1 subgoal

  a : R
  b : R
  H : ``a+b > 0``
  ============================
   ``(a+b)*(a+b)*(a+b) > 0``
```

5.1.3. Expand

For the tactic `Expand`, let us consider the following example in which we want to expand the term:

$$(a-b)(a+b)$$

```
expd1 < Intros.
1 subgoal

  a : R
  b : R
  H : ``a <> b``
  ============================
   ``(a-b)*(a+b) <> 0``

expd1 < Expand ``(a-b)*(a+b)``.
1 subgoal

  a : R
  b : R
  ============================
   ``a*a+ -(b*b) <> 0``
```

We can notice that the tactic `Expand` (actually, the Maple function `expand`) expands and also simplifies the expression.

5.1.4. Normal

We may use the tactic `Normal` in the following example in which we want to normalize the term:

$$\frac{x^2}{(x-y)^2} - \frac{y^2}{(x-y)^2}$$

In Maple, the `normal` function provides a basic form of simplification. It recognizes those expressions equal to zero which lie in the domain of "rational functions". This includes any expression constructed from sums, products and integer powers of integers and variables. The expression is converted into the "factored normal form". This is the form numerator/denominator, where the numerator and denominator are relatively prime polynomials with integer coefficients. Thus, in Coq, let us try to prove that if $x - y \neq 0$ then the previous expression is normalized into $\frac{x+y}{x-y}$:

```
norm1 < Intros.
1 subgoal

  x : R
  y : R
  H : ``x-y <> 0``
  ============================
   ``x*x/((x-y)*(x-y))-y*y/((x-y)*(x-y)) == (x+y)/(x-y)``

norm1 < Normal ``x*x/((x-y)*(x-y))-y*y/((x-y)*(x-y))``.
2 subgoals

  x : R
  y : R
  H : ``x-y <> 0``
  ============================
   ``(x+y)*/(x+ -y) == (x+y)/(x-y)``

subgoal 2 is:
 ``(x+ -y)*((x+ -y)*(x+ -y)) <> 0``
```

where the notation `/a` is just a syntactical abbreviation for the inverse, i.e. `1/a`.

The left-hand side term of the equality in the first subgoal is the result returned by Maple. The second subgoal has been generated by `Field`. Actually, we can notice that, applying the algorithm of subsection 3.2, two denominators must not be equal to 0, that is to say $x - y$ and $(x - y)^2$, even if it is redundant.

5.2. Maple evaluation

As seen previously in subsection 4.1, we can also perform only pure Maple computations thanks to the command `Eval...in`. These computations do not need to be proved and there is no guarantee that the results are correct but it is not necessary to generate side conditions because the consistency of the system is not endangered in any way. In this case, we just want to use Coq as a calculator with the help of Maple.

For example, let us consider the computation of the following characteristic polynomial:

$$\begin{vmatrix} -1-x & 4 & 2 \\ -5 & 7-x & 5 \\ 7 & -6 & -6-x \end{vmatrix}$$

We define the characteristic polynomial, we call `car1`, by expanding the determinant w.r.t. the first column:

```
Coq < Definition car1 [x:R] := Eval Factor in
Coq <   ``(-1-x)*((7-x)*(-6-x)+30)+5*(4*(-6-x)+12)+
Coq <     7*(20-2*(7-x))``.
car1 is defined
```

As we have factorized (with `Factor`) the previous development, we have a definition `car1` which can be directly used to build the corresponding diagonal matrix:

```
Coq < Print car1.
car1 = [x:R]`` -(x+ -1)*(x+ -2)*(x+3)``
     : R->R
```

We can perform other computations, in exactly the same way, with the other Maple functions that have been imported.

6. Conclusion

6.1. Summary

In this paper, several goals have been achieved:

- We have emphasized the paradigm computation/proof, which appears, for example, in the opposition CAS/DS. In particular, it is apparent that no concrete system (not only formalisms), which could handle both computation and deduction in a clean way, has been ever designed.
- In the context of the Coq proof assistant, we have presented the tactic `Field`, which can automatically prove the equalities over fields. Regarding the implementation, this tactic has been realized in a reflexive way, which ensures both efficiency and certification.
- We have described a very light interface between Coq and Maple, which allows us to perform Maple computations in Coq. These can be pure computations, which do not require any validation, or computations used during proofs, which are then *automatically* certified using the tactic `Field`.
- We have provided some examples of use (for each new Coq computation based on a Maple function) with either pure computations (`Eval...in`) or computations used during proofs (usual tactic call), which may generate side conditions arising from the application of the tactic `Field`.

6.2. Related work

The idea of making CASs and DSs interact has been also explored by some other authors. A study of this interaction can be found in [4] where, as seen in section 2, several approaches (believing, skeptical and autarkic) are discussed. This is a very general characterization of the communication of mathematical contents between CASs and DSs so that any concrete interface between a CAS and DS should, *a priori*, fall into the scope of one of these approaches.

Other approaches, which differ from that put forward in this paper, consist in providing CAS users with an interface with a DS to verify side conditions. For example, in [1], the idea is to track the conditions, which are implicit in the CAS, but which can make the result go wrong if they are ignored. In this case, the CAS is Maple and the DS is PVS [19]. This view is quite different because it is dedicated to CAS users and there is a graphic interface which is used to build some proofs interactively without interacting directly with PVS. Moreover, the PVS proofs must be completed manually (no automation is provided for a specific Computer Algebra domain). In a similar way, another work [10] aims at using the automated theorem prover Otter to discard trivial conjectures generated by the HR theory generation tool [9] about Maple functions. Once these conjectures proved, Otter leaves the *interesting* conjectures, which cannot be easily proved.

In more closely related approaches, the CAS is devoted to help a DS. For instance, in [2], Maple is used to extend the power of rewriting of the Isabelle theorem prover [20]. No proof obligation is generated and according to [4], the interface is based on a believing principle (i.e., each rewriting using a Maple computation is set as an axiom; obviously, this is managed by the Isabelle simplifier at the meta-logic level and this is quite transparent for the user). Another example is [16] consisting in an interface between HOL [14] and Maple, where Maple is supposed to help HOL users to perform certain computations. This tool is mainly dedicated to the domain of real numbers and Maple is called with the two methods `SIMPLIFY/FACTORIZE`. The results returned by Maple are proved in HOL (as mentioned in the title of the paper, this is a skeptical approach) but not systematically so. In contrast to our approach, the imported functions are not embedded in specific HOL tactics, which call Maple for computations and prove the correctness of these computations automatically. In fact, the user calls the Maple functions in HOL and, if he/she wants to use the results of these computations in proofs, he/she has to prove their correctness either manually or automatically (if there is a corresponding tactic). Finally, in the idea of integrating Computer Algebra into Proof Planning, [23] presents a concrete implementation where Maple is used to enhance the computational power of the Ωmega system [25]. The computations are checked by means of an external tool, which provide protocol information to aid the verification in Ωmega. This external tool deals essentially with verifications involving arithmetic and for other kind of verifications, it must be modified by integrating an appropriate algorithm.

6.3. Future work

An extension of this work could be to deal with other kinds of computations such as limits, derivatives, etc. (these notions have already been formalized in Coq). With this interface, these computations can already be quite easily imported into Coq and if we also want to deal with them in proofs, it is always possible to prove their correctness, at least manually. However, in the same way as for the tactic `Field`, the idea would be to also develop the corresponding automation.

For example, if we denote f', the derivative of f, we would like to provide a new tactic which can automatically prove in Coq that f' is the derivative of f, for some specific forms of f, precisely as is done for algebraic expressions over a field with the tactic `Field`.

Another application of this interface could be to use Maple to compute polynomial gcd's. Currently, we are developing a decision procedure for first order formulae over algebraic closed fields based on a method of quantifier elimination, and as such methods require many gcd computations, we plan to use Maple to do so. The correctness proofs will also be automated using the tactic `Field` and Bezout's relation (the coefficients will be given by Maple). Once realized, a large part of this procedure could be also reused to deal with real closed fields. It is particularly interesting to finalize this work because it will mean that CASs can be used in DSs not only to carried out some computations but also to enhance the automation of such systems.

Finally, we would like to use a more general language to express the output data coming from Coq. The idea would be to interface Coq with other CASs. Currently, the data produced by the interface are quite *ad hoc* and can only be parsed by Maple. A good choice could be to adopt languages like OpenMath [7] or OMDoc [17], which seem to be emerging standards and already have bindings with many existing CASs.

A. A complete example: quadratic forms

In this example, we propose to show the equality between two quadratic forms where the right-hand side form is expressed in such a way that we can compute the rank of the quadratic form trivially (i.e. a linear combination of squares of independent linear forms). To do so, we use Gauss's algorithm[7] on the left-hand side quadratic form to get the right-hand side form. Actually, we could rather expand and reduce the right-hand side to get the left-hand side expression, but we suppose that we are in the situation where we try to compute the rank of the left-hand side quadratic form (which is usually the case). The equality we consider is the following:

$$x^2 + y^2 - xy - yz - zx + z^2 = (x - \frac{y+z}{2})^2 + \frac{3}{4}(y-z)^2$$

After some simple algebraic modifications (associativity and commutativity), we have to prove (the lemma has been named `quadratic`):

```
quadratic <
1 subgoal
```

[7]In Coq and some other proof assistants based on the Curry-Howard isomorphism, a typical proof style can be to use an algorithm to guide a proof. Afterwards, a program implementing the given algorithm can be automatically extracted from this proof.

```
  x : R
  y : R
  z : R
  ============================
   ``x*x-x*y-z*x+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

The first step consists in partially factorizing $x^2-xy-zx$ into $(x-\frac{y+z}{2})^2-\frac{(y+z)^2}{4}$:

```
quadratic < Replace ``x*x-x*y-z*x`` with
quadratic <         ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``.
2 subgoals

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``

subgoal 2 is:
 ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4 == x*x-x*y-z*x``
```

The second goal has been generated by the tactic `Replace`. To prove this goal, we use the new tactic `Simplify`, coming from Maple, to simplify the left-hand side term and to get the right-hand side term. The idea is to conclude using the tactic `Reflexivity` which applies the reflexivity property of the equality. As `Simplify` calls the tactic `Field`, another subgoal is also generated, i.e. the side condition that $2 \times (2 \times 4) \neq 0$. This condition is simply proved by the tactic `DiscrR` which deals with equalities and inequalities over reals involving constants:

```
quadratic < 2:Simplify ``(x-(y+z)/2)*(x-(y+z)/2)-
quadratic <                 (y+z)*(y+z)/4``;
quadratic <   [Unfold Rminus;Rewrite (Rmult_sym z x);
quadratic <    Reflexivity |DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

where `Unfold Rminus` replaces the binary minus with the unary minus (in the right-hand side part) and `Rewrite (Rmult_sym z x)` commutates `z` and `x` (also in the right-hand side part).

After some associativity manipulations, we can perform a full distribution in $-\frac{(y+z)^2}{4}+y^2-yz+z^2$ thanks to the tactic `Expand` (the side condition $4\times 2\neq 0$ is generated and proved by `DiscrR`):

```
quadratic < Expand ``-((y+z)*(y+z)/4)+y*y-y*z+z*z``;
quadratic <   [Idtac|DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)+(3*/4*y*y+
     -(3*/2*y*z)+3*/4*z*z) ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

Now, we can factorize to get the last square using `Factor` (again a side condition, i.e. $4\times 2\neq 0$ is generated and proved by `DiscrR`):

```
quadratic < Factor ``3*/4*y*y+ -(3*/2*y*z)+3*/4*z*z``;
quadratic <   [Idtac|DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)+3*/4*(y+ -z)*(y+ -z) ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

Finally, once `3*/4` and `y+-z` have been respectively transformed into `3/4` and `y-z`, we can conclude using the tactic `Reflexivity`.

The Coq proof script is given in detail below:

```
Lemma quadratic:(x,y,z:R)
  ``x*x+y*y-x*y-y*z-z*x+z*z==
    (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``.
Proof.
  Intros.
  Replace ``x*x+y*y-x*y-y*z-z*x+z*z`` with
          ``x*x-x*y-z*x+y*y-y*z+z*z``;[Idtac|Ring].
  Replace ``x*x-x*y-z*x`` with
          ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``.
  2:Simplify ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``;
```

```
    [Unfold Rminus;Rewrite (Rmult_sym z x);Reflexivity
    |DiscrR].
  Unfold 2 Rminus.
  Replace ``(x-(y+z)/2)*(x-(y+z)/2)+
            -((y+z)*(y+z)/4)+y*y-y*z+z*z`` with
          ``(x-(y+z)/2)*(x-(y+z)/2)+
            (-((y+z)*(y+z)/4)+y*y-y*z+z*z)``;[Idtac|Ring].
  Expand ``-((y+z)*(y+z)/4)+y*y-y*z+z*z``;[Idtac|DiscrR].
  Factor ``3*/4*y*y+ -(3*/2*y*z)+3*/4*z*z``;[Idtac|DiscrR].
  Fold ``3/4``;Fold ``y-z``.
  Reflexivity.
Save.
```

References

1. Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer Algebra Meets Automated Theorem Proving: Integrating Maple and PVS. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs 2001*, volume 2152 of *LNCS*, pages 27–42. Springer-Verlag, 2001.

2. Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In *International Symposium on Symbolic and Algebraic Computation*, pages 150–157, 1995.

3. Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning (JAR)*, 28(3):321–336, April 2002.

4. Henk Barendregt and Arjeh M. Cohen. Electronic Communication of Mathematics and the Interaction of Computer Algebra Systems and Proof Assistants. *Journal of Symbolic Computation (JSC)*, 32(1/2):3–22, July/August 2001.

5. Samuel Boutin. *Réflexions sur les quotients.* PhD thesis, Université Paris 7, April 1997.

6. Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software*, pages 515–529, 1997.

7. Olga Caprotti and Arjeh M. Cohen. On the Role of OpenMath in Interactive Mathematical Documents. *Journal of Symbolic Computation (JSC)*, 32(4):351–364, September 2001.

8. Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *The* Maple V *Language Reference Manual.* Springer-Verlag, New York, 1991. ISBN 0387976221.

9. Simon Colton. The HR Program for Theorem Generation. In *Proceedings of the 18th International Conference on Automated Deduction*, LNCS, pages 285–289. Springer-Verlag, 2002.

10. Simon Colton. Making Conjectures about Maple Functions. In *Proceedings of the Joint International Conferences on Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, pages 259–274. Springer-Verlag, 2002.

11. Thierry Coquand, Catarina Coquand, Thomas Hallgren, and Aarne Ranta. The Alfa Home Page, 2001.
http://www.md.chalmers.se/~hallgren/Alfa/.

12. David Delahaye and Micaela Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier (France)*. INRIA, January 2001.
ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/JFLA2000-Field.ps.gz.

13. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. Technical Report RR-3400, INRIA-Rocquencourt, France, April 1998.
ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz.

14. M. J. C. Gordon and T. F. Melham. *Introduction to* HOL*: a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

15. John Harrison. Metatheory and Reflection in Theorem Proving: a Survey and Critique. Technical Report CRC-053, SRI Cambridge, UK, 1995.
http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

16. John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

17. Michael Kohlhase. OMDoc: an Open Markup Format for Mathematical Documents. Technical Report SEKI SR-00-02, Universität des Saarlandes, Saarebrucken (DE), 2000.
http://www.mathweb.org/omdoc.

18. Martijn Oostdijk and Herman Geuvers. Proof by Computation in the Coq System. *Theoretical Computer Science*, 272(1–2):293–314, 2002.

19. Sam Owre, Natarajan Shankar, and John Rushby. PVS: A Prototype Verification System. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.

20. Larry Paulson and Tobias Nipkow. The Isabelle Home Page, 2003.
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html.

21. Randy Pollack. *The Theory of* Lego*: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994. http://www.dcs.ed.ac.uk/home/rap/export/thesis.ps.gz.

22. Harald Rueß. Computational Reflection in the Calculus of Constructions and its Application to Theorem Proving. In R. Hindley, editor, *Proceedings for the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Lecture Notes in Computer Science, Nancy, France, April 1997. Springer-Verlag.

23. Volker Sorge. Non-Trivial Computations in Proof Planning. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems : Third International Workshop, FroCoS 2000*, volume 1794 of *LNCS*, pages 121–135, Nancy, France, 22–24 March 2000. Springer-Verlag, Berlin, Germany.

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

25. the Ωmega Group. Ωmega: Towards a Mathematical Assistant. In *Proceedings of CADE-14*, volume 1249 of *LNAI*, pages 252–255. Springer-Verlag, 1997.