Proving Equalities in a Commutative Ring Done Right in Coq

Benjamin Grégoire and Assia Mahboubi

INRIA Sophia-Antipolis, 2204, routes des Lucioles - B.P. 93,
06902 Sophia Antipolis Cedex, France
{Benjamin.Gregoire, Assia.Mahboubi}@sophia.inria.fr

Abstract. We present a new implementation of a reflexive tactic which solves equalities in a ring structure inside the Coq system. The efficiency is improved to a point that we can now prove equalities that were previously beyond reach. A special care has been taken to implement efficient algorithms while keeping the complexity of the correctness proofs low. This leads to a single tool, with a single implementation, which can be addressed for a ring or for a semi-ring, abstract or not, using the Leibniz equality or a setoid equality. This example shows that such reflective methods can be effectively used in symbolic computation.

1 Introduction

In the context of a computer algebra system, one of the most extensively used functionalities is the simplification of symbolic expressions, and in particular, the use of algebraic identities. These identities are usually established by elementary combinations of canonical identities, stored in a very large database, in a quite efficient way. Programing similar tools in a proof assistant consists in programing decision procedures, as the user is concerned with the reliability of the result.

Algebraic identities that the user of proof assistant is to handle are often equalities modulo the axioms of a ring. There are numerous examples of such identities: the product of two bi-squares is itself a bi-square, remarkable identities like the famous $(a + b)^2 = a^2 + 2ab + b^2$ or event more complex properties like the fact that the product of sums of eight squares is a sum of eight squares. These equalities are decidable and it seems natural to relieve the user of a proof assistant of such goals, by providing an automatic tool. Otherwise the proof of the identity:

$$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

would require no more than thirty elementary rewriting steps of the ring axioms.

The Coq [12] proof assistant already provides such a tool called `ring`. It is not based on an automatic rewriting strategy but built using a reflexive technique [3]. The use of reflexivity has already reduced the size of the generated proof terms and the time for building and checking them. Nevertheless, the efficiency of `ring` is not satisfactory. For example, proving $10 * 100 = 1000$, is immediate if the multiplication ranges over the integers, while it takes about a

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 98–113, 2005.

hundred seconds on a 3GHz machine if the multiplication ranges over the axiomatic implementation of real numbers. The efficiency of the method on such goals should not depend on the computational nature of the underlying ring structure. This bad behaviour on constants strongly affects the efficiency of the method on algebraic identities of higher degree. Moreover the implementation choices made in the `ring` developpment are really limiting the size of the entries `ring` is able to deal with.

Currently, there exists eight different implementations of `ring` depending on the kind of ring: semi-ring or ring, abstract or not, setoid equality or Leibniz equality. Here, we factorize these eight implementations through a modular implementation which will be finally instantiated to fit the kind of ring required.

The Coq system has recently been improved by the introduction of a compiler and an abstract machine, which now allows the evaluation of Coq programs with the same efficiency as Ocaml programs [8]. After the experiences of marrying computer algebra systems with theorem provers to get both efficiency and reliability [9], it now seems reasonable to use Coq as a single environment for programming, certifying and evaluating computer algebra algorithms. Our `newring` decision procedure is one of these efficient tools required for the manipulation of symbolic expressions, showing that the reflexive methods are the way to separate computations from checking, inside the proof assistant. Furthermore it is the first step for a bunch of other decision procedures, like the simplification of field equalities [6], or decision methods in geometry [11].

In Section 2, we begin with some general remarks about the reflexive method and its use in our particular context. The Section 3 is dedicated to our choice to get efficient representation of polynomials, which is a crucial point for the efficiency. The Section 4 shows the major importance of the choice of coefficients set for these polynomials. In the Section 5, we introduce a new axiomatic structure, called *almost-ring*, which allows to unify the implementations of the procedure for rings and semi-rings. In Section 6 we show how the use of the new metalanguage Ltac [5,2] allows to completely avoid the use of external Ocaml code. Section 7 is dedicated to examples and benchmarks before we conclude in Section 8.

2 Overall View of the Method

2.1 Reflexivity

In the Coq system, the rewriting steps are explicit in a proof: each step builds a predicate having the size of the current goal when the rewriting was performed, hence the size of the proof term heavily depends on the number of these rewriting steps. The reflection technique introduced by [1] takes benefit of the reduction system of the proof assistant to reduce the size of the proof term computed and consequently to speed up its checking. It relies on the following remark:

- Let $P : A \rightarrow Prop$ be a predicate over a set A.

- Suppose that we are able to write in the system a semi decision procedure f, such that f is computable and if f returns *true* on the entry x, then $P(x)$ is valid, that is to say:
  `f_correct: forall x, f(x)=true -> P(x).`

If we want to prove $P(y)$ for a particular y, and if we know that $f(y)$ *reduces* to *true*, then we can simply apply the lemma `f_correct` to y and to a proof that *true* = *true*. Thanks to the conversion rule which allows to change implicitly the type of a term by an equivalent (modulo β-reduction):

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

This latter proof, which is `(refl_equal true)`, is also implicitly a proof that $f(y) = \mathit{true}$ because $f(y)$ reduces to *true*, so $\mathit{true} = \mathit{true}$ is convertible with $f(y) = \mathit{true}$. Finally the proof of $P(y)$ we have built is :

```
f_correct y (refl_equal true)
```

The size of such a proof now only depends on the size of the particular argument y and does not depend on the number of implicit β-reduction steps: explicit rewriting steps have been replaced by implicit β-reductions. The size of the proof term of the correctness lemma for f may be large, it is only done once and for all. It will be shared by all the instantiations and will no more be type-checked. The efficiency of this technique of course strongly depends on the efficiency of the system to reduce the application of the decision procedure $f(y)$, hence on the efficiency of the decision procedure itself.

2.2 General Scheme of the `newring` Tactic

The `newring` tactic operates on a ring structure A, which includes a base type for its elements, two constants 0 and 1, three binary operations $+, *, -$ over A and an opposite unary function $-$, together with the usual axioms defining a commutative ring structure. Its goal is to prove the equality of two terms t_1 and t_2 of type A modulo the ring axioms.

Working by reflection means that we want to build a semi decision procedure f, which will take t_1 and t_2 as arguments and return *true* if t_1 and t_2 are equal modulo associative-commutative rewriting in the ring structure.

A natural way to perform a comparison between two terms seems to be the pattern-matching. Yet the Coq system does not allow pattern matching over arbitrary terms, but only over inductive types. That is why terms of the type A are going to be *reflected* into an appropriate inductive type *PolExpr*, which describes the syntax of terms of type A. This step is also called the *metaification*. A term of type A is mapped by the meta-function $\mathcal{T}$ to a polynomial expression in *PolExpr* by:

- interpreting every ring constant as a constant polynomial expression (eg. 0,1)
- interpreting every ring operation as an operation over polynomial expressions

- hiding every subterm which is neither a ring constant, nor the application of a ring operation to other subterms behind a labeled variable and building the corresponding association list.

$\mathcal{T}$ is a kind of oracle, we will explain in Section 6 how to build such a function using the *meta*-language Ltac[5] which allows to do pattern-matching over an arbitrary Coq expression.

Once we have built the two *PolExpr*, e_1 and e_2, corresponding to t_1 and t_2, the idea is to check the equality of the normal forms of e_1 and e_2 and to prove that this implies the equality of t_1 and t_2. For this purpose, we should ensure the correctness of the following diagram:

$$\begin{tikzcd}
e_1 = e_2 & \mathit{PolExpr} \arrow[r, "\mathit{norm}"] \arrow[d, "\varphi_{PE}"] & \mathit{Pol} \arrow[d, "\varphi_P"] & \mathit{norm}(e_1) = \mathit{norm}(e_2) \\
t_1 = t_2 & A \arrow[u, dotted, bend left, "\mathcal{T}"] \arrow[r, phantom, "\Longleftrightarrow"] & A & \varphi_P(\mathit{norm}(e_1)) = \varphi_P(\mathit{norm}(e_2))
\end{tikzcd}$$

by the correctness lemma:

$$\forall e \in \mathit{PolExpr},\ \varphi_{PE}(e) = \varphi_P(\mathit{norm}(e))$$

φ_{PE} (resp. φ_P) are the evaluation functions. They evaluate polynomial expressions (resp. normalized polynomial *Pol*) into elements of A, by interpreting back each constant polynomial to a constant of A, each variable by the ring term it was hiding and each representation of an operator by the corresponding ring operator.

These functions can be easily defined within the theory by pattern matching over the reflected inductive types.

The inductive type *PolExpr* is adapted to the metaification. To ensure the completeness of our tactic it should verify the following meta property:

$$\forall a \in A.\ \varphi_{PE}(\mathcal{T}(a)) = a.$$

Note that we do not have to prove this property, which can not be expressed inside Coq. It does not affect the correctness of our decision procedure, but only its completeness.

The type *Pol* stands for the set of the normalized forms of polynomial expressions, which does not need to be the same as *PolExpr*. It is adapted to build normal forms efficiently. The *norm* function bridges the gap between these two kind of constraints: *PolExpr* suits to the syntax of the terms in A and *Pol* allows efficient computations.

To prove the equality of t_1 and t_2, our tactic first computes e_1 and e_2 using $\mathcal{T}$, and then checks the equality of their normal forms. If it holds, the correctness lemma and the transitivity of equality ensure the equality of t_1 and t_2:

$$t_1 = \varphi_{PE}(\mathcal{T}(t_1)) = \varphi_P(\mathit{norm}(\mathcal{T}(t_1))) = \varphi_P(\mathit{norm}(\mathcal{T}(t_2))) = \varphi_{PE}(\mathcal{T}(t_2)) = t_2$$

3 Sparse Horner Normal Forms

Choosing the shape of the normal form is a crucial point for the complexity. The normal form for terms in the ring will be determined by the choice made for the normal form of polynomial expressions. We present here the choice we made for the normal form, the sparse Horner normal form, which provides the required efficiency.

3.1 Representation

Horner form for polynomials in $C[X]$ can be represented by the following inductive type:

```
Inductive Pol1 (C:Set) : Set :=
 | Pc : C -> Pol1 C
 | PX : Pol1 C -> C -> Pol1 C.
```

where `(Pc c)` represents the constant polynomial c and `(PX P c)` represents the polynomial $P * X + c$. The problem with such a representation is that a polynomial can have a lot of holes due to gaps in the degrees. For example, $X^4 + 1$ is represented in the Horner form as:
`(PX (PX (PX (PX (Pc 1) 0) 0) 0) 1)`. The number of nested `PX` constructors of such a polynomial is indeed its degree. To get a more compact representation of the Horner form we can factorize these gaps by adding a power index in the constructor of non constant polynomials:

```
Inductive Pol1 (C:Set) : Set :=
 | Pc : C -> Pol1 C
 | PX : Pol1 C -> positive -> C -> Pol1 C.
```

where `positive` is a inductive type representing $\mathbb{N}^*$.

Now `(PX P i c)` stands for the polynomial $P * X^i + c$. So $X^4 + 1$ is now represented as `(PX (Pc 1) 4 1)`.

Once the representation of univariate polynomials is fixed, there is a natural way to extend it to multivariate polynomials, using the canonical isomorphism $C[X_1, \ldots, X_n] = C[X_1 \ldots X_{n-1}][X_n]$. In Coq this can be done by declaring the following fixpoint using dependent type:

```
Fixpoint Poln (C:Set) (n:nat) {struct n} : Set :=
 match n with
 | O => C
 | S m => Pol1 (Poln C m)
 end.
```

The type `(Poln C n)` represents the set of polynomials with n variables. Namely `(Poln C (S n))` represents the set of univariate polynomials with coefficients in `(Poln C n)` and `(Poln C O)` is the set of constant polynomials in C.

This representation creates another kind of holes corresponding to holes in variables. For example the polynomial 1 will be encoded either by `(Pc 1)` if it is seen as an element of $Z[X]$ or by (Pc (Pc (Pc (Pc 1)))) if it is seen as an element of $Z[W, X, Y, Z]$. To solve this problem, we give up the idea of defining multivariate polynomials recursively from univariate ones. We now define the set of polynomials in an arbitrary number of variables in one shot.

```
Inductive Pol (C:Set) : Set :=
 | Pc : C -> Pol C
 | Pinj : positive -> Pol C -> Pol C
 | PX : Pol C -> positive -> Pol C -> Pol C.
```

- (Pc c) stands for the constant polynomial $c \in C[X_1, \ldots, X_n]$ for any n.
- If $Q \in C[X_1, \ldots, X_{n-j}]$, and Q is its representation, then (Pinj j Q) represents Q as a polynomial in n variables, namely $Q.X^0_{n-j+1} * \ldots * X^0_n$. We have "pushed" Q from $C[X_1, \ldots, X_{n-j}]$ to $C[X_1, \ldots, X_n]$. j is called the injection index.
- Finally, (PX P i Q) stands for $P * X_n^i + Q$ where $P \in C[X_1 \ldots X_n]$ and $Q \in C[X_1 \ldots X_{n-1}]$ is constant in X_n.

3.2 Normalization

Our sparse Horner form does not provide a unique representation for arbitrary polynomials. In $C[X]$ the polynomial $X^4 + 1$ can be represented by (PX (Pc 1) 4 (Pc 1)) or by (PX (PX (Pc 1) 3 (Pc 0)) 1 (Pc 1)). To solve this, we can define a normalization function that build a canonical representative of a polynomial, and then define the equality on polynomial as the equality of the canonical representatives.

Instead of normalizing before checking equality, our choice is to always manipulate canonical representatives verifying the three following properties:

- the coefficient of highest degree is never zero;
- the injection index is the biggest possible;
- the power index is the biggest possible.

So the canonical representative of $X^4 + 1$ is (PX (Pc 1) 4 (Pc 1)). Note that, it is also the most compact representation of a sparse Horner form. Since the complexity of operations depends on the size of the polynomials, linear for addition and quadratic for multiplication, it is interesting to work with canonical terms. This means that each operation on polynomials should only build canonical terms. If P and Q are in canonical form, building the canonical representation of (PX P i Q) is not expensive, since we only need to locally destruct P:

- if P = (Pc 0) then build the canonical representative of (Pinj 1 Q);
- if P = PX P' i' (Pc 0) then the canonical representative is:
  (PX P' (i+i') Q)
- else (PX P i Q) is the canonical representative.

Our defined operations on polynomial, denoted by Padd, Pmul, Psub, and Popp, keep the following invariant: if their arguments are canonical then their result is canonical. To ensure this, we use specialized constructors that perform local normalizations: `mkPinj` and `mkPX`. For example, the addition of `(PX P i Q)` and `(PX P' i Q')` leads to the term `(mkPX (Padd P P') i (Padd Q Q'))`. Since the addition of `P` and `P'` can be the zero polynomial, we need to use `mkPX` to ensure that the result is canonical. But we directly use constructors `Pinj` and `PX`, which are costless, each time the invariant allows it, as in the addition of `(PX P i Q)` and `(Pc c)` which reduces to `(PX P i (Padd Q (Pc c)))`, here `P` can not be zero or of the form `(PX P' i' (Pc 0))`, since `(PX P i Q)` is canonical, so `(PX P i (Padd Q (Pc c)))` is canonical.

For each operator, we prove a correctness lemma showing that the operator is correct up to evaluation. For the addition the lemma is:

```
 Lemma Padd_correct: forall P Q l,
      phiP l (Padd P Q) == (phiP l P) + (phiP l Q).
```

where `==` is the setoid equality over the initial ring (or semi-ring) structure and `+` is its addition.

Note that using `mkPX` instead of `PX` has no influence on the correctness, because `(phiP l (mkPX P i Q))` is equal to `(phiP l (PX P i Q))`. The only influence is for completeness, since using `PX` instead of `mkPX` can produce a non-canonical representative. But again, we do not need to prove completeness.

The normalization function from polynomial expressions to their canonical sparse Horner forms consists in mapping variables to monomials, constants to constant polynomials and operation constructors to operation functions on Horner form. The canonical representative is given by the evaluation of the term obtained.

After having defined the normalization function, we can prove its correctness:

```
Lemma norm_correct : forall l e, phiPE l e == phiP l (norm e).
```

And then the main lemma, which expresses the correctness of our decision procedure:

```
Lemma f_correct : forall l e1 e2,
  Peq (norm e1) (norm e2) = true -> phiPE l e1 == phiPE l e2.
```

where `Peq` stands for a defined function which checks the syntactic equality over sparse Horner forms.

The set of coefficients C is the carrier of the computations performed by the normalization function. The following section will show that the choice made for C is crucial, especially for the efficiency of the procedure, as C catches the "best computational part" of the ring.

4 Computations over the Parametric Coefficient Set

The normalization function we have described above strongly relies on the computational behavior of the set of coefficients. For example the normalization of

$x + (-x)$ leads to $(1 + (-1)).x$, which will *reduce* to $0.x$. C has to be chosen as a set over which we know how to compute, as efficiently as possible. In the Coq system, these kind of sets will be represented by inductive types, and the operations are defined as functional programs.

In the Coq system, `Z` is an implementation of $\mathbb{Z}$ as lists of binary digits. In the case `Z` is the underlying ring of the equality to be proved, `Z` itself is a good candidate. On the other hand, if the underlying ring is `R`, the axiomatic implementation of real numbers in Coq, `R` itself will not be an appropriate set of coefficients. Indeed, in `R`, $1 + (-1)$ is equal to 0 (using ring axioms) but does not reduce to 0: the subtraction as the other operations and constants of `R` are only symbols, and are not evaluable. Hence $x + (-x)$ would not reduced to $0.x$ by the normalization function. Since there is a natural inclusion of $\mathbb{Z}$ in $\mathbb{R}$, we can use `Z` as a set of coefficients. Moreover, whatever ring A we are dealing with, the canonical morphism from $\mathbb{Z}$ to A will enable us to use again `Z` as a set of coefficients. This type `Z` seems then to be a universal candidate for coefficients.

Nevertheless, `Z` will not always be the good choice. If the computational content of the ring operations is stronger than the ring axioms, this method will allow to prove more than what is provable by sole rewriting of the rings axioms. In the case we are working in the ring `bool`, the equality $x+x = 0$ holds, even if it is not provable using only the ring axioms. The good choice for C is now `bool` itself: the left side of the equality is again reflected in $X+X$ (with coefficients in `bool`), whose normal form $(1+1).X$ is reduced to $0.X = 0$ by the normalization function, thanks to the computations over the coefficients in `bool`. Hence our choice is to parametrize our tactic by the set of coefficients and to let the user make the most appropriate choice.

An inductive type has to fulfill some requirements to be admissible as a set of coefficients. These requirements will ensure the correctness of the normalization function. Formally, C will be admissible if it is equipped with the constants and operators of a ring, and with a decidable equality relation $=_C$. The last requirement is needed to implement the `mkPX` and `mkInj` constructors (we need to be able to check the equality at 0). It also allows to get a decidable equality on sparse Horner form.

We also require a suitable evaluation function from C to A, mapping the constants of C to the elements of A and this function should be compatible with the respective operations of C and A. These requirements can be expressed by the existence a so-called *morphism* between C and A (even if C does not need to be a ring). This morphism evaluates the constants and operators in C into their analogous in A, and the decidable equality relation $=_C$ over C should satisfies : if $(x =_C y)$ returns *true*, then the evaluations of x and y will be equal in A.

Once we have got C and a proof of all these specifications, we define in a generic way the operations over polynomials as explained in Section 3, and extend the morphism between C and A into two evaluation functions φ_{PE} and φ_P, from the polynomial expressions and sparse Horner form to A. We also obtain a proof of the general diagram of the reflection presented in 2.2, *Pol* and *PolExpr* being now replaced by their parametrized version $Pol(C)$ and $PolExpr(C)$.

We have implemented the identity morphism which corresponds to taking the ring itself as the set of coefficient. The user can always apply the resulting tactic even if it may not prove much equalities (like in the case `R` is involved). We have also implemented the morphism from `Z` to an arbitrary ring, which can always be used as an efficient default choice, but is not necessary the best choice (cf the case of `bool`).

In order to get the maximal efficiency from this method, the user has to make to most appropriate choice for C. If the ring structure is defined in an axiomatic way, like `R`, `Z` will always be a good choice for the set of coefficients. In the case the ring already presents a computational content, like `Z` or `bool`, it may be a good choice to take the ring itself as the coefficient set. Nevertheless, if the available operations are not efficient enough, like it is the case for example in the semi-ring of Peano numbers, it may be more appropriate to obtain the most efficient computational content by changing the set of coefficients all the same, here for example by taking a binary representation of natural numbers.

5 Unifying Rings and Semi-rings

A semi-ring is a ring where the axioms stating the existence of an opposite (and of a subtraction) have been replaced by an extra axiom : $\forall x, 0 * x = 0$. These structures are quite alike and we would like to get a tool also adapted to semi-rings without duplicating the code. For this purpose, we work with an intermediate structure, called *almost-ring.* The idea is to complete a semi-ring with a unary operator, called *almost-opposite* which is morally the opposite operator of a ring structure. This operator will be instantiated by a dummy function to equip a semi-ring with such a structure. In fact the fundamental remark is the following : in the correctness proof of the normalization function, the axiom defining the *opposite* operator as an inverse, by stating that $\forall x, x + (-x) = 0$ is never used itself, but only the properties which describe its combination with the other operators. Finally an almost-ring is defined by the following axioms:

- $\forall x,\ 0 + x = x$
- $\forall x\ y,\ x + y = y + x$
- $\forall x\ y\ z,\ x + (y + z) = (x + y) + z$
- $\forall x,\ 1 * x = x$
- $\forall x\ y,\ x * y = y * x$
- $\forall x\ y\ z,\ x * (y * z) = (x * y) * z$
- $\forall x\ y\ z,\ (x + y) * z = x * z + y * z$
- $\forall x, 0 * x = x$ (at that point we have a semi-ring)
- $\forall x\ y,\ -(x * y) = -x * y$ (combination of pseudo-opposite with product)
- $\forall x\ y,\ -(x + y) = -x + -y$ (combination of pseudo-opposite with addition)
- $\forall x\ y,\ x - y = x + -y$ (definition of an associated pseudo-subtraction)

It is straightforward to prove that every ring is an almost-ring. The axioms of an almost-ring do not allow to prove the missing axiom defining the opposite in ring $x + -x = 0$. Anyway, this identity will be proved by our tactic, provided that in the set of coefficients $1 + (-1)$ reduces to 0. This is ensured thanks to the existence of a morphism from the set of coefficients to the ring. Every semi-ring can also be equipped with an almost-ring structure if we take the identity as an almost-opposite operator and the defined addition operator of the semi-ring as subtraction.

The tactic is finally designed for an almost-ring structure. We have moreover built the proofs required to transform any ring or semi-ring into the associated almost-ring.

The last parameter given to the tactic is the equality relation used over the ring. It may not be the Leibniz equality, but an equivalence relation adapted to the ring structure. For example, this is the case for an implementation of $\mathbb{Q}$ as $\mathbb{Z} \times \mathbb{N}^*$. A set equipped with such an equality relation is called a setoid ([7],[10]). Proving equalities in such a setoid ring requires extra properties stating that all the ring operations are compatibles with the given setoid equality. In the case the equality involved in the goal is the Leibniz one, these requirements are trivial to fulfill. That is why the tactic will finally also be parametrized by a setoid equality and the related compatibility lemmas for the operations.

6 Programming the Metaification and the Tactic

The purpose of the `newring` tactic is to solve goals of the form $t_1 == t_2$ by applying the `f_correct` lemma. To do so we need to produce a list of values l and two polynomial expressions e_1 and e_2 such that the evaluation of e_1 (resp. e_2) at l is convertible to t_1 (resp. t_2). Consider the following equality

$$3 * \sin(x) * x = x * (\sin(x) + 2 * \sin(x)) + 0 * y$$

In this case l will be $[\sin(x);\ x;\ y]$, e_1 will be $3 * X_1 * X_2$ and e_2 will be $X_2 * (X_1 + 2 * X_1) + 0 * X_3$.

6.1 Programming the Metaification

We use the Coq proof-dedicated metalanguage Ltac[5] to design the oracle producing the expected values (l, e_1, e_2). This metalanguage allows to do pattern-matching on arbitrary Coq terms, and thereby to program this metafunction, which is a tactic, in a natural way as done in [6].

We first build a function `FV` which computes the list l containing the subterms to abstract. These are the ones which do not belong to the syntax of a ring. Then the `mkPolexpr` tactic computes the two expresssions e_1 and e_2 and the list l is used to know which variable is associated to a given subexpression to abstract.

```
Ltac mkPolexpr Cst add mul sub opp t l :=
 let rec mkP t :=
  match t with
  | (add ?t1 ?t2) =>
    let e1 := mkP t1 in
    let e2 := mkP t2 in constr:(PEadd e1 e2)
  | (mul ?t1 ?t2) => ...
  | (sub ?t1 ?t2) => ...
  | (opp ?t1) => ...
  | _ =>
    match Cst t with
    | false => let p := Find_at t l in constr:(PEX p)
    | ?c => constr:(PEc c)
    end
  end
 in mkP t.
```

The tactic `mkPolexpr` takes as arguments a term `t`, the list `l` of terms to abstract, the ring operators and a tactic `Cst`. It matches the head symbol of `t`:

- If this symbol is one of the given operators then it builds recursively the corresponding polynomial expression;
- If the head symbol is not an operator then either `t` is a constant or it has to be abstracted into a variable. This discrimination is performed by the tactic `Cst` given in argument :
  - If `Cst` returns `false` then the index of the proper variable is given by the position of `t` in the list `l` given in argument.
  - Otherwise `t` is mapped to the corresponding constant.

The definition of the `Cst` tactic depends on the ring A. If A is an abstract ring, the set of coefficients will be `Z`, and we can already define a naive tactic which matches only the neutral elements of A (`r0` and `rI`).

```
Ltac genCstZ r0 rI t :=
  match t with
   | r0 => constr:(0%Z)
   | rI => constr:(1%Z)
   | _ => constr:false
   end.
```

On the other hand, in the case A is `Z`, the set of coefficients will be `Z` itself, and we can match much more constants: in fact all the terms built only with the constructors of `Z`.

```
Ltac ZCst t :=
 match (is_ZCst t) with
 | true => constr:t
 | false => constr:false
 end.
```

Here is_ZCst is a tactic matching the terms built only with the constructors of the inductive type Z.

This method has also been generalized to the case of semi-rings, where N, the implementation of binary natural numbers plays the role of Z. We have also built such a tactic Cst for boolean, where the target constants are booleans.

6.2 The Generic Tactic

To define the newring tactic itself, we use the possibility given by Ltac to program a higher-order function, which builds a tactic, solving equalities in the structure given in argument. For the sake of clarity we present a simplified version that can be used only if the goal is a valid equality modulo ring axioms and fails otherwise. The real implementation also replace both members of the equality by their normal form if they are not equal.

```
Ltac Make_ring_tac add mul sub opp req Cst_tac :=
  match goal with
  | [ |- req ?r1 ?r2 ] =>
    let fv := FV Cst_tac add mul sub opp (add r1 r2) (nil R) in
    let e1 := mkPolexpr Cst_tac add mul sub opp r1 fv in
    let e2 := mkPolexpr Cst_tac add mul sub opp r2 fv in
    apply (f_correct fv e1 e2); compute; exact (refl_equal true)
  | _ => fail "not equality"
  end.
```

The tactic first checks that the current goal is an equality. If so, it computes a single list fv of subterms to be abstracted in both terms, and the two polynomial expressions e1 and e2 representing the members of the equality. Then the tactic applies the correctness lemma f_correct. At that point the tactic should prove the hypothesis of the lemma, namely check that (norm e1) ?== (norm e2) is equal to true.

If r1 and r2 are equal modulo ring axioms then this new goal is convertible to true = true. So it is now possible to complete the proof with the term (refl_equal true). The tactic exact checks that the provided term has a type convertible to the current goal ((norm e1) ?== (norm e2)) = true. This is performed using a lazy reduction strategy. Here checking the convertibility is equivalent to computing the normal form of the equality's left-hand side. The efficient strategy suitable to this problem is the call by value reduction. So the tactic first uses the compute tactic to reduce the goal in this way, before concluding with exact.

We can now apply the Make_ring_tac to obtain a tactic which automatically prove ring equality in Z:

```
Ltac zring := Make_ring_tac Zplus Zmult Zminus Zopp (@eq Z) ZCst.
```

We also have implemented such a tactic for booleans (bring), reals (rring) and natural numbers (nring), Peano numbers as well as their binary implementation.

Finally, the newring tactic analyzes the type of the equality to prove and calls the corresponding specialized tactic:

```
Ltac newring :=
 match goal with
 | [|- @eq Z    _ _ ] => zring
 | [|- @eq R    _ _ ] => rring
 | [|- @eq bool _ _ ] => bring
 | [|- @eq nat  _ _ ] => nring
 end.
```

To work with an other user-defined structure, one can always use the predefined tactic Make_ring_tac to build the appropriate tactic for proving equalities in this structure.

7 Examples and Benchmarks

The newring tactic has performed two orthogonal improvements compared to the choices made in the ring tactic developped by S. Boutin [3]. The first one is the choice of the sparse Horner form for the representation of normal forms instead of an ordered sum of monomials, being themselves an ordered product of variables. The second is to use Z as the set of coefficients for reflected expressions when working with abstract rings (R for example).

7.1 Sparse Horner Form

Figure 1 describes the time to normalize the expression $(x_1 + \ldots + x_n)^d$ seen as a polynomial with coefficients in Z. For ring, the normal form of this expression is its expansion in an ordered sum of monomials, each prefixed by a coefficient in Z. Both tactics use Z as a set of coefficients, so these benchmarks show the interest of the sparse Horner form to deal with polynomials of higher degree. The gain in time for $n = 5$ and $d = 5$ is a factor 6 and a factor 500 for $n = 7$ and $d = 9$, thanks to the compactness of sparse Horner form representation. Using a naive Horner form (without power and injection index, or not maintaining canonical representatives) introduces an overhead of 30%. Moreover, the ring tactic is not able to normalize this expression when $n = 8$ and $d = 9$, and when $n = 12$ it fails for $d = 6$. The newring tactic is able to normalize the expression for $n = 12$ and $d = 11$.

Comparing the time to normalize expressions of the form $(x_1 + \ldots + x_n)^d$ to the results given by the expand function of Maple, is deceiving. The algorithm used by the computer algebra system in mainly focused on the access to a database of stored identities, and possible simple combinations of them. When the precomputed identities are useless, the system is of course less efficient, and can even fail because of the size of the normal form. This is the case for expressions of the following form

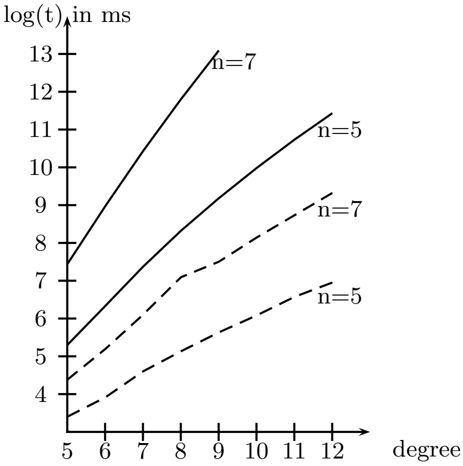

Fig. 1. Time to prove that $(x_1 + \ldots + x_n)^d$ is equal to its normal form

$$\begin{array}{ccccccccccc}
(y & + & x_2 & + & \ldots & + & x_{n-1} & + & x_n) & * \\
(x_1 & + & y & + & \ldots & + & x_{n-1} & + & x_n) & * \\
 & & & & \vdots & & & & & \\
(x_1 & + & x_2 & + & \ldots & + & y & + & x_n) & * \\
(x_1 & + & x_2 & + & \ldots & + & x_{n-1} & + & y\ ) &
\end{array}$$

For $n = 8$ the `newring` tactic is four times slower than the `expand` strategy of Maple (0.4s for `newring`, 0.12s for Maple). But Maple fails to expand the formula when $n = 9$ (`Error, (in expand/bigprod) object too large`), while `newring` finishes in 1.7s.

7.2 The Set of Coefficients

Beside the successful use of the Horner form, the use of `Z` as the set of coefficients when we are working with an abstract ring has been a major improvement for efficiency. For the previous `ring` tactic, the representation of normal forms in an abstract ring leads to coefficients equivalent to unary numbers, hence computations are completely inefficient. Proving that 10*100 = 1000 takes about one hundred of seconds on a 3GHz machine using `ring`, and it is now immediate with `newring` (as one would expect). It is worth paying attention to the efficiency of such a tactic over (large) integers. One often deals with expressions with small coefficients but successive computations may increase their size in a significant way. A well-known phenomenon of explosion in the size of the coefficients occur while computing a remainder sequence of polynomials, like the computation of a polynomial gcd in $\mathbb{Q}[X]$. For example, in the context of the checking of computations made by an external oracle [9] (Maple or any dedicated program producing a trace of certificates...), checking the successive steps of such a computation will force to deal with large coefficients, even if the initial polynomial entries had small ones.

8 Conclusion

This development shows that it is worth paying attention to the algorithmic aspects in programming such a procedure in the same way we would have done while programming it in a functional language. The choices we made in that sense turned out to be primordial for efficiency. This gain in efficiency could have lead to a complication of the associated correctness proofs. This is not the case, as the possible difficulties in the proofs lie in the mathematical complexity of the problem more than in the choices made for computations. This effort has even allowed to reduce the size of the development, by factorizing the eight versions of the tactic in a single one.

One other characteristic feature of the reflexive method is that it requires, for the reflection step, the use of an operator defined in the meta level, and hence using the meta-language of the system. The Ltac metalanguage turns out to be exactly the tool needed in reflexive tactic to program this reflection step in the meta-theory. The mechanism of pattern-matching over Coq terms indeed enables to write this function easily, without any knowledge of the inside of Coq and to work entirely at the top-level, without needing to compile again and again the whole sources of the system to integrate the new tactic.

A possible improvement for our development would be to allow negative powers in the representation of polynomials, to deal with Laurent series. But, one can also use the remark that proving an equality in a field can be transformed into a goal in a certain ring plus nonzero conditions for the denominators. This implementation of a `newfield` tactic has been achieved by L. Théry.

This work shows that the sparse Horner form is the right representation to compute efficiently with polynomials. We hope that existing developments, such as the decision procedure for geometry [11], strongly relying on the `ring` tactic will gain in efficiency and hence in power.

We are also convinced that this will allow the development of other efficient procedures to deal with symbolic expressions, providing a basic toolkit for larger developments in the domain of certified computer algebra. In particular, the second author uses the Horner representation of polynomials to develop a decision procedure for real numbers theory based on G. Collins' cylindrical algebraic decomposition [4], which is a quite complex algorithm resting on numerous computations over polynomials (computations of gcd, subresultant coefficients,...).

The efficiency of `newring` overcomes what was before a strongly limiting factor in such a development, showing that it is possible to compute efficiently within a proof assistant. This makes possible to use the proof assistant as a single environment for computing and proving as well as an efficient checker efficiently computations possibly performed by an external tool as described in [9].

The systematic use of `Z` as a set of coefficients has considerably increased the efficiency of the tactic. Yet `Z`, in which numbers are represented as lists of bits, is not the best possible implementation for integers. An other step toward the efficiency of a genuine computer algebra system will be to provide to the user the possibility to use a library of machine binary integers, comprising fast computing operations, in order to deal even more efficiently with the huge integers occurring during symbolic computations (eg. polynomial gcds, prime numbers).

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the 5^{th} Symposium on Logic in Computer Science*, pages 95–197, Philadelphia, Pennsylvania, June 1990. IEEE, IEEE Computer Society Press.
2. Y. Bertot and P.Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. 2004.
3. S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.
4. G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. volume 33 of *Lecture Notes In Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975.
5. D. Delahaye. A Tactic Language for the System Coq. In *LPAR, Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. http://cedric.cnam.fr/ delahaye/publications/LPAR2000-ltac.ps.gz.
6. D. Delahaye and M. Mayero. Field: une procédure de décision pour les nombres réels en Coq. In *Journees Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001.
http://cedric.cnam.fr/ delahaye/publications/JFLA2000-Field.ps.gz.
7. V. C. G. Barthe and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, March 2003.
8. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
9. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
10. M. Hofmann. A simple model for quotient types. In *TLCA '95*, pages 216–234, April 1995.
11. J. Narboux. A decision procedure for geometry in coq. In *TPHOLs*, pages 225–240, 2004.
12. The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, mars 2002. http://coq.inria.fr/doc8/main.html.