Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

A Method to Combine Algebraic Computations with Related Deductions

Wolfgang Gehrke

RT-DIA-28-97 **1997**

Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale 79
I - 00146 Roma, Italy
wgehrke@inf.uniroma3.it

sponsored by the project MURST 40% "Calcolo Algebrico e Simbolico"

ABSTRACT

We investigate the utilization of parametric code to perform algebraic computation on the one hand and to allow for related deduction on the other hand. This method is illustrated by the implementation of several number systems and the study of their main properties. The examples are presented by means of the programming language SML but we argue that the method is not restricted to this particular language.

1 Introduction

The recent development of numerous automated systems brought a high level of sophistication for the support of mathematical calculation. One example are Computer Algebra Systems (CAS) which can perform a large number of algebraic manipulations. Another example are Automated Theorem Provers (ATP) which support the rigorous development of proofs.

Recent efforts concentrate on the combination of the two mentioned types of systems trying to enhance the proving capabilities of ATP and to verify the calculations of CAS. Our work can be seen in this context where we take a particular point of view: the correctness of algebraic algorithms. Our ultimate goal is to establish a proof of correctness for fundamental algebraic algorithms.

This aim is not new, for a discussion in the context of finite mathematics see [Sla94]. The availability of correct algebraic methods can be also seen as a contribution to the paradigm of Constraint Logic Programming (CLP) [JM94]. In the context of reasoning and logic programming sound algebraic methods are indispensable.

The method we will apply makes use of parametric code. This code is ready to be instantiated with operational structures to perform computations. Furthermore the same parametric code can be used for a symbolic instantiation suitable for proving properties of the resulting code.

It is essential that the parametric code is not changed itself but only instantiated in different ways. Thus operations assumed in the parameters can be used in the given construction. In the same way assumptions about properties of the parameters can be used to reason about the constructed code.

This method will be illustrated by the study of several number systems. Firstly the complex and hyper-complex numbers are considered on top of the real numbers. Secondly the integers and rational numbers are studied starting from the natural numbers.

The constructions are standard [EHH+90]. The well-known properties of these number systems are those of rings and fields. Therefore the universal and special word problem for rings was of crucial importance.

The proof procedures for the word problems of rings are related to Gröbner bases [BW93] which them-self are of computational nature. This more "semantic" approach significantly outperformed another "syntactic" approach working with AC-unification. Furthermore with the intended semantics in mind also constructions of counterexamples could be provided.

Illustrated by these examples we contribute the following:

- a method of using parametric code to calculate and to reason about it,
- a case study where "semantic" methods (Gröbner bases) outperform purely "syntactic" methods (equational reasoning),
- a discussion of the independence of our approach from the particular language chosen for presentation.

In Sec. 2 we sketch some background for the studied examples. The method is given in Sec. 3 followed by the examples in Sec. 4. A discussion of our approach follows in Sec. 5. Finally we conclude and suggest future work.

2 The Case Study and Related Background

This section gives an overview of the algebraic structures under consideration. We briefly sketch the known main properties of these structures. Finally we present the algebraic procedures which can be used for automated reasoning verifying these properties.

The considered examples in our study are various number systems. Firstly these are the integers and rational numbers starting from the natural numbers. Secondly these are the complex and hyper-complex numbers starting from the real numbers.

All these systems have in common that they are formed with the help of polynomial equations over the correspondingly underlying system. Therefore handling multivariate polynomials will be of importance for the proof procedures. In case of the integers and rational numbers additionally the presence of a congruence relation has to be taken into account.

Firstly we consider the **natural numbers** $\mathbb{N} = (0_\mathbb{N}, 1_\mathbb{N}, +_\mathbb{N}, *_\mathbb{N})$ as the underlying structure. On top of them the **integers** $\mathbb{Z} = (0_\mathbb{Z}, 1_\mathbb{Z}, +_\mathbb{Z}, -_\mathbb{Z}, *_\mathbb{Z})$ are introduced as equivalence classes of pairs of natural numbers with

$$(a, b) \sim_\mathbb{Z} (c, d) :\iff a +_\mathbb{N} d =_\mathbb{N} c +_\mathbb{N} b$$

where (a, b) denotes a solution of $b +_{\mathbb{N}} x =_{\mathbb{N}} a$ for $a, b \in \mathbb{N}$. On top of the integers the **rational numbers** $\mathbb{Q} = (0_{\mathbb{Q}}, 1_{\mathbb{Q}}, +_{\mathbb{Q}}, -_{\mathbb{Q}}, *_{\mathbb{Q}}, 1/_{\mathbb{Q}})$ are introduced as equivalence classes of pairs of integers with

$$(e, f) \sim_{\mathbb{Q}} (g, h) :\Longleftrightarrow e *_{\mathbb{Z}} h \sim_{\mathbb{Z}} g *_{\mathbb{Z}} f$$

where (e, f) denotes a solution of $f *_{\mathbb{Z}} y \sim_{\mathbb{Z}} e$ for $e \in \mathbb{Z}$ and $f \in \mathbb{Z} \setminus \{0_{\mathbb{Z}}\}$.

The operations on these pairs are introduced in the usual way cf. [EHH+90]. The constructions are motivated by a search for inverse operations of addition and multiplication. In fact these constructions allow for an easy definition of the corresponding inverse operations:

$$-_{\mathbb{Z}}(a, b) := (b, a) \text{ for } a, b \in \mathbb{N}$$
$$1/_{\mathbb{Q}}(e, f) := (f, e) \text{ for } e, f \in \mathbb{Z} \setminus \{0_{\mathbb{Z}}\}$$

Secondly we consider the **real numbers** $\mathbb{R} = (0_{\mathbb{R}}, 1_{\mathbb{R}}, +_{\mathbb{R}}, -_{\mathbb{R}}, *_{\mathbb{R}}, 1/_{\mathbb{R}})$ as the underlying structure. On top of them we introduce the **complex** and **hyper-complex numbers**. The complex numbers $\mathbb{C}$ can also be understood as an algebraic extension of the real numbers allowing for a solution i of the equation $i *_{\mathbb{C}} i =_{\mathbb{C}} -_{\mathbb{C}} 1_{\mathbb{C}}$ and all complex numbers can be represented as $\alpha + \beta i$ with $\alpha, \beta \in \mathbb{R}$.

As a short reminder we present the hyper-complex numbers namely the **quaternions** $\mathbb{H}$ and the **octonions** $\mathbb{O}$ by means of a multiplication table. This table makes use of the fact that $\mathbb{H}$ is a 4-dimensional (a base is $1, i, j, k$) and $\mathbb{O}$ is a 8-dimensional (a base is $1, i, j, k, E, I, J, K$) **algebra** over the real numbers [EHH+90]. The Tab. 1 does not contain an entry for the corresponding unit 1 of multiplication:

*	i	j	k	E	I	J	K
i	-1	k	-j	I	-E	-K	J
j	-k	-1	i	J	K	-E	-I
k	j	-i	-1	K	-J	I	-E
E	-I	-J	-K	-1	i	j	k
I	E	-K	J	-i	-1	-k	j
J	K	E	-I	-j	k	-1	-i
K	-J	I	E	-k	-j	i	-1

Table 1. Multiplication Table of $\mathbb{H}$ and $\mathbb{O}$

After having introduced the object of study we sketch the involved abstract algebraic notions. We are interested in the basic algebraic properties of these number systems. These can be mainly characterized by rings and fields.

A **ring** with unit is an algebraic structure $R = (0, 1, +, -, *)$ such that the usual properties hold, i.e. $(0, +, -)$ form an **Abelian group**, 1 is a neutral element of the multiplication and distributivity holds. A ring is called **commutative** or **associative** if this property holds for the multiplication.

A **division ring** is an algebraic structure $F = (0, 1, +, -, *, 1/)$ such that $(0, 1, +, -, *)$ is a ring where all elements except 0 have a multiplicative inverse denoted by $1/$. A **skew field** is an associative division ring. A **field** is a commutative skew field.

For completeness we mention the notion of an **integral domain** which is an algebraic structure $D = (0, 1, +, -, *)$ which is a commutative and associative ring. Furthermore it is not just zero ring, thus $0 \neq 1$. Finally there are no divisors of 0, i.e. $z \neq 0 \;\&\; z' \neq 0 \Rightarrow z * z' \neq 0$.

Recall the following facts:

Lemma 1 (cf. [EHH+90]).
$\mathbb{Z}$ is an integral domain.
$\mathbb{R}$ is a field.

Based on these facts we wish to derive:

Proposition 2 (cf. [EHH+90]).
$\mathbb{Q}$ is a field.
$\mathbb{C}$ is a field.

$\mathbb{H}$ *is a skew field.*
$\mathbb{O}$ *is a division ring.*

We will attempt to prove these facts automatically.

Obviously the most important notion is the one of a ring. Furthermore we have to model the division in order to arrive at a division ring, skew field and field. There it will be necessary to keep track of constraints which in this context are expressions which have to be different from zero.

Recall the following:

Theorem 3 (universal word problem for rings). *The universal word problem for associative and commutative rings with unit is decidable.*

Proof. This can be shown by making use of a simplification procedure for multivariate polynomials [BCL83]. Another way is to make use of a canonical term rewriting system for rings with AC-unification [Hul80]. □

This fact will allow to derive most of the properties needed for the complex and hyper-complex numbers if we only consider the ring axioms. The algorithm has to be applied component-wise according to the dimension. It remains to handle the division properly.

In order to handle expressions with a multiplicative inverse we translate the expression with the help of

$$0 \mapsto \frac{0}{1}, 1 \mapsto \frac{1}{1}, 1/x \mapsto \frac{1}{x}, x \mapsto \frac{x}{1}$$

In case of $1/x$ we keep track of $\{x\}$ as a constraint. Furthermore the operations work in the following way

$$\frac{r}{s} + \frac{t}{u} := \frac{r * u + t * s}{s * u} \qquad \frac{r}{s} * \frac{t}{u} := \frac{r * t}{s * u}$$

with propagating the set $\{s, u, s * u\}$ in both cases as constraints.

We will only keep track of these constraints and provide them as an information for the user. Similar to the case of rational numbers we work with a congruence $\frac{r}{s} \sim \frac{t}{u} :\Longleftrightarrow r * u = t * s$. This approach reduces the universal word problem for a field to the universal word problem to an associative and commutative ring with unit modulo a set of constraints.

It remains to provide a mean for working with the congruence relations $\sim_{\mathbb{Z}}$ and $\sim_{\mathbb{Q}}$. A solution of the universal word problem does no longer suffice since we have to start with some additional equations expressed in form of polynomials. This is called the **special word problem**.

Recall the following:

Theorem 4 (special word problem for rings). *The special word problem for associative and commutative rings with unit is decidable.*

Proof. This can be shown by using Gröbner bases [Buc65, Buc85], here working with polynomials over the integers [BW93]. The main insight is the equivalence of the following two statements:

1. $\forall x_1 \ldots \forall x_n [\bigwedge_{l=1}^{m} f_i(\mathbf{x}) = 0 \Rightarrow f_0(\mathbf{x}) = 0]$ holds in the class of all associative and commutative rings with unit.
2. $f_0 \in Id(f_1, \ldots, f_m)$ where the ideal is taken in $\mathbb{Z}[\bar{X}]$.

The Gröbner bases is a convenient representation of a polynomial ideal allowing for solving numerous problems related to ideals like in this case the membership test. □

This will allow for automated proofs showing properties of $\mathbb{Z}$ and $\mathbb{Q}$. Actually we will make use of an even stronger result using Gröbner bases over **principal ideal domains** cf. [BW93].

Note 5. So far we made use of solutions for the universal and special word problem for rings. This has been accompanied by a constraint handling which collects all occuring divisors. However there are situations when a "yes/no" answer is not satisfactory and in particular in case of "no" we might be interested in a counterexample.

We will reconsider the equality of multivariate polynomials for the construction of counterexamples. Note that the integers $\mathbb{Z}$ can always be embedded into any ring with unit. Since the considered number systems are also integral domains the following is helpful:

Lemma 6 (cf. [Lip81]).
i) If D is an integral domain also $D[x]$ is an integral domain.
ii) Let D be an integral domain, $p_1, p_2 \in D[x]$ and $q > max(deg(p_1), deg(p_2))$. If there are different $x_1, \ldots, x_q$ such that $p_1(x_i) = p_2(x_i) \quad (i = 1, \ldots, q)$ then $p_1 \equiv p_2$.

Since $\mathbb{Z}$ is an integral domain we can make use of the last lemma. Multivariate polynomials can be constructed recursively. With the embedding of $\mathbb{Z}$ and the knowledge of working with an integral domain this provides a finite search space for counter examples or in other words gives another alternative to check equality of polynomials.

	$\mathbb{Q}$	$\mathbb{C}$	$\mathbb{H}$	$\mathbb{O}$
UWPR		ring properties		
SWPR	ring properties			
UWPR + constraints		laws for 1/		
SWPR + constraints	laws for 1/			
counterexamples			commutativity of $*$	associativity of $*$

Table 2. Overview of Proof Procedures

We have summarized the proof procedures in Tab. 2. UWPR refers to the universal and SWPR to the special word problem for rings. We will show another counterexample illustrating the problem with the attempt of defining a division beyond $\mathbb{O}$.

3 Overview of the Method

Here we present the method which makes use of parametric code. The key idea is that parametric code can be instantiated in different ways. According to the choice of instantiation the instance can be used either for computation or for symbolic manipulation.

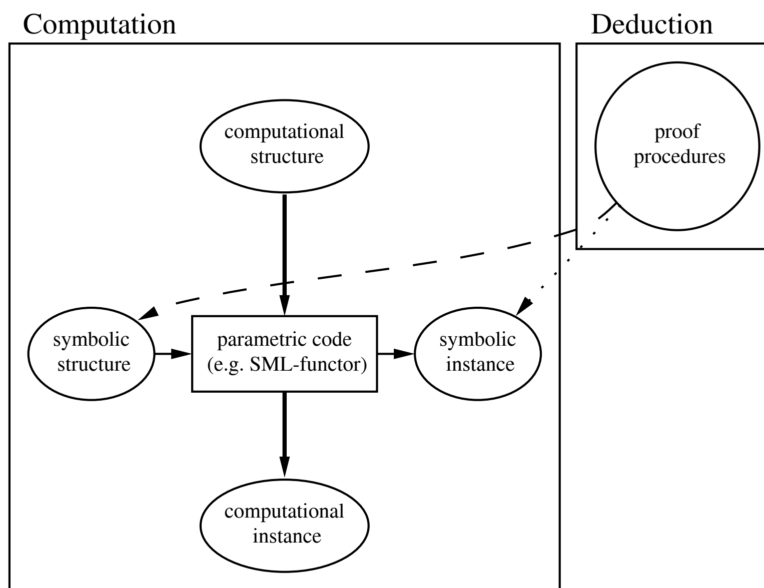

Fig. 1. Overview of the Method

This method is visualized in Fig. 1. On the left is shown the computational part. On the right is shown the deductive part.

The algebraic code is implemented with the help of parametric code. This code can be instantiated with an operational or a symbolic structure. The process of instantiation is indicated by a normal and a bold arrow indicating that the main interest is put on computing.

The provided symbolic structure and the generated symbolic instance can be used for reasoning. The dashed arrow indicates that for the provided symbolic structure some proving techniques are assumed. Those can be used for constructing proof procedures for the symbolic instance of the parametric code which is indicated by a dotted arrow.

The main steps of applying this method can be summarized as follows:

1. write parametric code for a new notion
2. write the intended input structure for computations
3. instantiate the code with this computational structure
4. make experiments with the resulting computational instance
5. write a symbolic structure which can be used by proof procedures
6. prepare corresponding proof procedures for the symbolic structure
7. instantiate the code with this symbolic structure
8. try to adapt the proof procedures to the symbolic instance

In this way the organization of the computational part induce a similar organization of the deductive part.

Remark. This method is very flexible since it does not influence the degree of detail which has to be provided for the reasoning part. Therefore one can adapt the needed characterization of the input structure according to the needs of proving for the output structure. For our examples we remained entirely in the range of fully automated procedures.

4 Examples

This section gives the flavor by showing the implementation of examples with the help of the programming language SML [MTH90, MT91, Pau91]. We mainly concentrate on the previously outlined method. Firstly we present the construction of the complex and hyper-complex numbers and thereafter we show the construction of the integers and rational numbers.

The complex and hyper-complex numbers can be constructed by a single so called "doubling construction" over the real numbers which is reflected by the following parametric code:

```
functor Double(structure G : FieldLike
               structure F : FieldLike
               sharing type F.ground = G.$ and
                       type G.ground = G.$) :
    sig
        include FieldLike sharing type ground = G.$
    end =
    struct
        (* ... *)
    end
```

The *structure* `G` refers to the ground field and the *structure* `F` refers to the ring which should be duplicated where the *signature* `FieldLike` looks like this:

```
signature FieldLike =
    sig
        type ground
        val dimension : int
        type $
        val g2f : ground -> $
        val base : $ list

        exception Dimension
        val gl2f : ground list -> $
        val f2gl : $ -> ground list
        val f2s : $ -> string
        val eq : $ * $ -> bool

        val plus : $ * $ -> $
        val zero : $
        val neg : $ -> $
        val minus : $ * $ -> $

        val times : $ * $ -> $
        val one : $

        val smult : ground * $ -> $
        val sprod : $ * $ -> ground
        val norm : $ -> ground
        val conj : $ -> $

        val re : $ -> $
        val im : $ -> $
        val vprod : $ * $ -> $

        exception ZeroInverse
        val reci : $ -> $
    end
```

The *sharing* in the *functor* `Double` guarantees that the type of the ground field will always be inherited.

If we are interested in computations a structure for the real numbers has to be provided which here is approximated with the built-in floating point numbers:

```
structure Reals =
    struct
        type ground = real
        val dimension = 1
        type $ = real
        fun g2f(r) = r
        val base = [1.0]

        (* ... *)
        fun eq(r1 : real, r2 : real) = (r1 = r2)

        val plus = Real.+
        val zero = 0.0
        val neg = Real.~
        val minus = Real.-

        val times = Real.*
        val one = 1.0

        (* ... *)
    end
```

With the help of this structure which *matches* the signature `FieldLike` we can construct all the computational instances for experiments:

```
structure Rc = Reals

structure Cc = Double(structure G = Reals
                      structure F = Rc)
structure Hc = Double(structure G = Reals
                      structure F = Cc)
structure Oc = Double(structure G = Reals
                      structure F = Hc)
structure Ac = Double(structure G = Reals
                      structure F = Oc)
```

The last structure `Ac` is another application of duplication which can be used for expermiments to investigate why the division cannot be defined in the same way any more.

Now we prepare related deduction and for this purpose we introduce "symbolic" rings and fields:

```
signature AbstractRing =
    sig
        datatype term = Zero | One | Var of int |
            Neg of term | Plus of term * term | Times of term * term
        val t2s : term -> string
        type intmultipoly

        val simplify : term -> term
        val meaning : term -> intmultipoly
        val equal : term * term -> bool

        val vars : term -> int list
        val degrees : term -> (int -> int)
        val eval : term * (int -> int) -> int
    end

signature AbstractField =
    sig
        datatype term = Zero | One | Var of int | Neg of term |
            Reci of term | Plus of term * term | Times of term * term
        type ring
        type field = ring * ring list * ring   (* num, constraints, den *)

        val t2f : term -> field
        val t2s : term -> string
        val equal : field * field -> bool
    end
```

The main ingredients are the datatypes `term` for symbolic rings and fields. Note that all operations of rings and fields are reflected by constructors with the exception that variables are added for the purpose of deduction.

These symbolic structures for rings and fields prepare the ground for constructing symbolic real numbers:

```
structure SymbolicReals =
    struct
```

```
        local open abstractField
        in
            type ground = term
            val dimension = 1
            type $ = term
            fun g2f(s) = s
            val base = [One]

            (* ... *)
            fun eq(t1, t2) = equal(t2f(t1), t2f(t2))

            fun plus(l,r) = abstractField.Plus(l,r)
            val zero = abstractField.Zero
            fun neg(t) = abstractField.Neg(t)
            fun minus(l,r) = plus(l,neg(r))

            fun times(l,r) = abstractField.Times(l,r)
            val one = abstractField.One

            (* ... *)
        end
    end
```

It becomes clear in which way the operations correspond to the constructors. Even though this code can almost be generated automatically we believe that it is still readable and short enough.

As before for computation we can generate now all symbolic instances:

```
structure Rs = SymbolicReals

structure Cs = Double(structure G = SymbolicReals
                      structure F = Rs)
structure Hs = Double(structure G = SymbolicReals
                      structure F = Cs)
structure Os = Double(structure G = SymbolicReals
                      structure F = Hs)
structure As = Double(structure G = SymbolicReals
                      structure F = Os)
```

Finally we have to wrap some proof procedures around these symbolic instances which will use the possibility to define variables inside terms:

```
signature Strategy =
    sig
        structure SF : FieldLike

        val proveEq : int * (SF.$ list -> SF.$ * SF.$) -> bool
        val cex4base : int * (SF.$ list -> SF.$ * SF.$) -> (SF.$ list) option
        val disproveEq : int * (SF.$ list -> SF.$ * SF.$)
            -> ((int list) list) option
    end
```

`proveEq` is a function to check for equality, `cex4base` checks all combinations of base elements for a counter example, `disproveEq` tries a finite set of points to find a counter example for the equality of two polynomials.

According to the organization of the code for the computational instances the code for deduction inherits a similar structure which is reflected by the functor `Strategy` and its application:

```
functor Strategy(structure SF : FieldLike
                 sharing type SF.ground = abstractField.term) : Strategy =
    struct
        (* ... *)
    end

structure Rt = Strategy(structure SF = Rs)
structure Ct = Strategy(structure SF = Cs)
structure Ht = Strategy(structure SF = Hs)
structure Ot = Strategy(structure SF = Os)
structure At = Strategy(structure SF = As)
```

These structures provide now a way for checking properties of complex and hyper-complex numbers. Implicitly all work with the assumption that the ground structure is indeed a field.

Here we show several checks of properties:

```
val htc = Ht.proveEq(2, (fn [x,y] => (Hs.times(x,y),
                                     Hs.times(y,x))))
val hta = Ht.proveEq(3, (fn [x,y,z] =>
          (Hs.times(Hs.times(x,y),z),Hs.times(x,Hs.times(y,z)))))
```

```
val SOME(htce) = Ht.cex4base(2, (fn [x,y] => (Hs.times(x,y),
                                              Hs.times(y,x))))
val SOME(htce´) = Ht.disproveEq(2, (fn [x,y] =>
                                 (Hs.times(x,y),Hs.times(y,x))))

val otc = Ot.proveEq(2, (fn [x,y] => (Os.times(x,y),Os.times(y,x))))
val ota = Ot.proveEq(3, (fn [x,y,z] => (Os.times(Os.times(x,y),z),
                                      Os.times(x,Os.times(y,z)))))
val SOME(otce) = Ot.cex4base(2, (fn [x,y] => (Os.times(x,y),
                                           Os.times(y,x))))
val SOME(otce´) = Ot.disproveEq(2, (fn [x,y] => (Os.times(x,y),
                                              Os.times(y,x))))
val SOME(otae) = Ot.cex4base(3,
                 (fn [x,y,z] => (Os.times(x,Os.times(y,z)),
                                Os.times(Os.times(x,y),z))))
val SOME(otae´) = Ot.disproveEq(3,
                  (fn [x,y,z] => (Os.times(x,Os.times(y,z)),
                                 Os.times(Os.times(x,y),z))))

val SOME(aal´e) =
    At.disproveEq(2,(fn [x,y]=>
                    (As.times(As.times(x,y),As.conj(y)),
                     As.times(x,As.times(y,As.conj(y))))))
val SOME(alr´e) =
    At.disproveEq(2, (fn [x,y] =>
                     (As.times(As.conj(x),As.times(x,y)),
                      As.times(As.times(As.conj(x),x),y))))
```

The first two blocks deal with commutativity and associativity of the multiplication in $\mathbb{H}$ and $\mathbb{O}$ and make use of the possibility to search for counterexamples in two ways. Running the last block demonstrates that after another duplication of the octonions a version of the **alternativity** law does not hold any more which is essential in defining the inverse with respect to multiplication.

This concludes the presentation for the complex and hyper-complex numbers. Now we proceed to the integers and rational numbers. We will only focus on the most interesting aspects since the previous example should have provided enough details of the method at work.

In a similar way firstly structures for computation are constructed:

```
structure Nc = NaturalNumbers
structure Zc = IntegerNumbers(Nc)
structure Qc = RationalNumbers(Zc)
```

Again the constructions are expressed as parametric code. There are two functors: `IntegerNumbers` constructs the integers and `RationalNumbers` the rational numbers.

Afterwards these functors are instantiated symbolically resulting in the structures:

```
structure Ns = NaturalSymbols
structure Zs = IntegerSymbols
structure Qs = RationalSymbols
```

where all these structures contain a function `impProve` which takes three arguments and makes use of a Gröbner base computation. The first argument is the number of variables, the second argument is a list of equalities which hold, the third argument is a list of equalities to be proved.

Several application of this function can be seen in the following:

```
val t1z = Zs.impProve(1, [], [(fn [a] => (a,a))])
val t2z = Zs.impProve(2, [(fn [a,b] => (a,b))], [(fn [a,b] => (b,a))])
val t3z = Zs.impProve(3, [(fn [a,b,c] => (a,b)),(fn [a,b,c] => (b,c))],
         [(fn [a,b,c] => (a,c))])
val t4z = Zs.impProve(4, [(fn [a,b,c,d] => (a,b)),(fn [a,b,c,d] => (c,d))],
 [(fn [a,b,c,d]=>(Zs.IntSym.plus(a,c),Zs.IntSym.plus(b,d))),
 (fn [a,b,c,d]=>(Zs.IntSym.times(a,c),Zs.IntSym.times(b,d)))])
val t5z = Zs.impProve(2, [(fn [a,b] => (a,b))],
         [(fn [a,b] => (Zs.IntSym.neg(a),Zs.IntSym.neg(b)))])

val zam´ =
    Zs.impProve(6,[(fn [a,b,c,d,e,f] => (a,b)),
   (fn [a,b,c,d,e,f] => (c,d)),
   (fn [a,b,c,d,e,f] => (e,f))],
[(fn [a,b,c,d,e,f] =>
  (Zs.IntSym.times(a,Zs.IntSym.plus(c,e)),
   Zs.IntSym.plus(Zs.IntSym.times(b,d),Zs.IntSym.times(b,f)))),
 (fn [a,b,c,d,e,f] =>
  (Zs.IntSym.times(Zs.IntSym.plus(a,c),e),
   Zs.IntSym.plus(Zs.IntSym.times(b,f),
  Zs.IntSym.times(d,f))))])
```

The first block checks that $\sim_{\mathbb{Z}}$ is indeed a congruence relation and that the operations are independent of the representative. The last call is a check of the distributivity laws with different representative.

Finally we show some examples for checking properties of rational numbers. The polynomial equations which have to be treated are more complicated as in the case of integers.

```
val qmi =
    Qs.impProve(1, [],
                [(fn [a] => (Qs.RatSym.times(a,Qs.RatSym.reci(a)),
                             Qs.RatSym.one)),
                 (fn [a] => (Qs.RatSym.times(Qs.RatSym.reci(a),a),
                             Qs.RatSym.one))])
val qmi´ =
    Qs.impProve(2, [(fn [a,b] => (a,b))],
                [(fn [a,b] => (Qs.RatSym.times(a,Qs.RatSym.reci(b)),
                               Qs.RatSym.one)),
                 (fn [a,b] => (Qs.RatSym.times(Qs.RatSym.reci(a),b),
                               Qs.RatSym.one))])
```

These examples verify that the multiplication has an inverse where the constraints are simply displayed for information when running the code.

Remark. In the last part we have been automatically proving properties of $\mathbb{Z}$ and $\mathbb{Q}$. Nevertheless we have been already working with the multivariate polynomials over the integers and the related Gröbner bases construction. Is this a contradiction?

No, since in fact we would like to start out from $\mathbb{N}$ and its properties. But we are mainly interested in equations and the multivariate polynomials over the integers are intended as the equality of that polynomial and zero. By allowing entire polynomials on both hand sides of an equation we can completely avoid the use of negative terms.

Therefore the language of multivariate polynomials over the natural numbers and their equations is completely sufficient. It is only another way of coding and does not influence the work of e.g. the Gröbner bases computation. Thus we can consider the properties of $\mathbb{Z}$ and $\mathbb{Q}$ to be really derived from properties of $\mathbb{N}$.

5 Discussion

In this section we want to discuss two points. Firstly we comment on our experience with a purely syntactic approach by equational reasoning. Secondly we argue that the presented method does not depend on the programming language SML chosen for purpose of convenient presentation.

The studied properties of the various number systems were related to the universal and special word problem for rings. Semantically this universal word problem can be solved by normal forms for multivariate polynomials and the special word problem can be solved by means of Gröbner bases. In case of the universal word problem for rings one could also apply equational reasoning using a canonical system for rings with AC-unification or one can even simulated Buchberger's algorithm by rewriting techniques[Bün96].

In fact we also tried along these lines with the help of an interface to the Larch Prover [GG91]. However the necessity of AC-unification became a bottleneck. In particular for problems related to the higher-dimensional hyper-complex numbers the prover did not respond in a reasonable time and we interrupted the proving attempt.

For our method note that it relies on the availability of syntactic means in a given programming language to express parametric code. The programming language SML is only one example where this is quite conveniently possible. But it is not the only such language as for example Ada allows "generic packages" or C++ "class templates".

In all these languages it is possible to instantiate parametric code in an operational way and in a symbolic way. Naturally this symbolic instance can be used in a similar way to prove properties. The choice of the programming language does matter when properties of that language have to be shown.

We have been choosing SML because it has a formally defined semantics which potentially allows for proving statements about this language. The concept of functor which provides the mean for parametric code could be used to functorize also over all language dependent constructs. In this way not only the dependency on certain constructions but also the dependency on properties of the language can be made completely transparent.

6 Conclusions and Future Work

We have presented a method how to combine algebraic computations with related deductions making use of parametric code. One instantiation by operational structures can be immediately used to test examples. Another instantiation by a symbolic structure provides a framework to prove properties of the parametric code.

This idea is demonstrated by the study of different number systems and their properties. The necessary proof procedures are related to multivariate polynomials and Gröbner bases. This semantical treatment itself has the character of a computation rather than a deduction.

In order to close the cycle and to arrive at verified code for algebraic algorithms this method has to be applied to the proof procedures them-self. For this purpose the availability of a formal semantics – as in the case of SML – becomes an important point for the attempt of such correctness proofs what we intend to try as future work.

Another gap we would like to close is the lack of a suitable implementation for the real numbers in the sense of computable real numbers. The availability of such an implementation is of importance for exact real computations. However proof techniques for the real numbers will require a quite different treatment than polynomial equations.

References

[BCL83] B. Buchberger, G.E. Collins, and R. Loos. *Computer Algebra - Symbolic and Algebraic Computation.* Springer-Verlag, 1983.

[Buc65] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Doctoral Dissertation, Mathematical Institute, University of Innsbruck, Austria, 1965.

[Buc85] B. Buchberger. *Multidimensional Systems Theory*, chapter Gröbner bases: An algorithmic method in polynomial ideal theory, pages 184–232. Reidel, Dordrecht, 1985.

[Bün96] R. Bündgen. Buchberger's algorithm: The term rewriter's point of view. *Theoretical Computer Science*, 159:143–190, 1996.

[BW93] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra.* Number 141 in Graduate Texts in Mathematics. Springer-Verlag, 1993. in Cooperation with H. Kredel.

[EHH+90] H.-D. Ebbinghaus, H. Hermes, F. Hirzebruch, M. Koecher, K. Mainzer, J. Neukirch, A. Prestel, and R. Remmert. *Numbers.* Number 123 in Graduate Texts in Mathematics. Springer-Verlag, 1990.

[GG91] S.J. Garland and J.V. Guttag. *A Guide to LP, The Larch Prover.* Massachusetts Institute of Technology, 1991.

[Hul80] J.-M. Hullot. A Catalogue of Canonical Term Rewriting Systems. Technical Report CSL-113, SRI International, April 1980.

[JM94] J.J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19,20:503–581, 1994.

[Lip81] J.D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley Publishing Company, 1981.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Sla94] J. Slaney. The Crisis in Finite Mathematics: Automated Reasoning as Cause and Cure. In A. Bundy, editor, *12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 1–13. Springer-Verlag, 1994.

This article was processed using the LaTeX macro package with LLNCS style