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

Complex and Hyper-Complex Numbers: A Case Study for the Combination of Algebraic Computation and Deduction

Wolfgang Gehrke

RT-INF-7-96 **1996**

Università degli Studi di Roma Tre
Dipartimento di Informatica
Via della Vasca Navale 84
I - 00146 Roma, Italy
wgehrke@inf.uniroma3.it

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

ABSTRACT

Recent work investigates the combination of Computer Algebra Systems and Automated Theorem Provers. We elaborate a concrete example: the generic construction of complex numbers, quaternions, Cayley numbers on top of real numbers. The implementation is done by a functor in the programming language SML which can be instantiated to perform the algebraic computations. Another instantiation of the same functor with symbolic expressions is used to perform reasoning about the resulting structure with the help of an external equational prover - the Larch Prover and also by symbolic evaluation of examples.

1 Introduction

Computer Algebra Systems (CAS) can perform a large number of algebraic manipulations. Automated Theorem Provers (ATP) support the rigorous development of proofs. Recent efforts concentrate on the combination of the two enhancing the proving capabilities of ATP [HC96] and verifying the calculations of CAS [KKS96].

This paper focuses on a concrete example combining computations and deductions. The example studies a basic building block of mathematics: numbers. Nevertheless the involved computations are interesting and the corresponding deductions are not just of small size.

Starting out from *natural numbers* ($\mathbb{N}$) more complex structures can be built like *integers* ($\mathbb{Z}$), *rational numbers* ($\mathbb{Q}$), *real numbers* ($\mathbb{R}$), *complex numbers* ($\mathbb{C}$). This process involves several interesting constructions like:

congruence relations $(a,b) \sim (c,d) :\Leftrightarrow a \circ d \equiv b \circ c$ used e.g. for defining $\mathbb{Z}$ as $\mathbb{N} \times \mathbb{N}_{/\sim}$ where $\circ$ becomes addition,

Cauchy sequences $\{f : \mathbb{N} \to \mathbb{Q} | \forall \varepsilon > 0 \exists n \forall m, m' > n : |f(m) - f(m')| < \varepsilon\}$ as a way to define $\mathbb{R}$,

duplication $(a,b) * (c,d) := (ac - bd, ad + bc)$ as the key formula for defining the multiplication in $\mathbb{C}$ seen as pairs of elements from $\mathbb{R}$.

Related abstract algebraic notions to classify these and other number systems are for instance *group*, *ring*, and *field* cf. [Lip81].

Our example examines the construction of complex numbers $\mathbb{C}$, quaternions $\mathbb{H}$ and Cayley numbers $\mathbb{O}$ starting from real numbers $\mathbb{R}$. To achieve this we study a generic duplication method `M`. Iterating the application of this method $\rightsquigarrow_M$ results in the following sequence: $\mathbb{R} \rightsquigarrow_M \mathbb{C} \rightsquigarrow_M \mathbb{H} \rightsquigarrow_M \mathbb{O}$.

We implemented this generic construction in the functional programming language SML [MTH90, MT91]. `Functors` of SML allow the implementation of parameterized structures [Pau91]. The functor can be instantiated and iterated by providing the `built-in reals` and thereafter concrete computations can be carried out.

On the other hand we were interested in the study of the resulting structure, i.e. the investigation whether they are a field, a skew field, a division ring or just a ring. Thus the same functor was instantiated with a symbolic structure. All the operations expand in this case to a symbolic expression over the provided symbolic structure.

Term rewriting systems (TRS cf. [DJ90, Klo92]) are used to investigate equational theories [Bac91]. In particular there exist canonical systems (modulo AC, i.e. associativity and commutativity) for groups, rings, and algebras [Hul80, Geh92]. Therefore it was natural to attempt equational reasoning about this symbolic structure.

We have built an interface from the symbolic expressions to the external theorem prover LP - the Larch Prover [GG91]. This is an theorem prover which supports the determination of canonical term rewriting systems and allows the user to provide termination orderings. Besides it allows a detailed trace with various levels and scripting such that computations can externally be repeated.

The main contribution of this paper can be summarized as follows:

- a generic construction of complex and hyper-complex numbers in SML ready for experiments with different instantiations,
- an interesting example for the combination of computation and deduction by instantiating the same functor code,
- a simple method to interface SML with an external theorem prover along with an attempt to handle decision problems related to rings and fields.

In Sec. 2 we briefly summarize related abstract algebraic notions. Sec. 3 presents the generic duplication of real numbers. The relationship with term rewriting is shown in Sec. 4. The two different instantiations of the functor are given in Sec. 5. We discuss the experimental results in Sec. 6 and finally conclude and suggest future work.

2 Some Background

In this section we recall abstract algebraic notions which are suitable for the study of different number systems. We present the definitions of groups, rings, fields in the equational style and mention algebras

cf. [OS77]. These notions will be needed to analyze the real, complex, and hyper-complex numbers and to formulate related rewrite systems.

Definition1 (Group). $(G, \circ, {}^\star, \upsilon)$ is called a *group* if $G \neq \emptyset$ is a non-empty set, $\circ$ a binary operation, * an unary operation and $\upsilon \in G$ an element, such that the following laws hold:

$$x \circ (y \circ z) = (x \circ y) \circ z \tag{1}$$
$$x \circ \upsilon = x = \upsilon \circ x \tag{2}$$
$$x \circ x^\star = \upsilon = x^\star \circ x \tag{3}$$

If furthermore holds

$$x \circ y = y \circ x \tag{4}$$

it is called *Abelian* group or *commutative* group.

Example 1. The natural numbers $\mathbb{N}$ are not a group yet since they lack the *inverse* operation * although they have a *unit* element υ which is zero. The integers $(\mathbb{Z}, +, -, 0)$ form an Abelian group. Also the rational numbers without zero with respect to multiplication $(\mathbb{Q} \setminus \{0\}, *, {}^{-1}, 1)$ form an Abelian group.

Definition2 (Ring). $(R, *, +, -, 0)$ is called a *ring* if $(R, +, -, 0)$ is an Abelian group and furthermore for the binary operation $*$ holds:

$$x * (y + z) = (x * y) + (x * z) \tag{5}$$
$$(x + y) * z = (x * z) + (y * z) \tag{6}$$

Additionally R is called:

associative if $x * (y * z) = (x * y) * z$,
commutative if $x * y = y * x$,
unit ring if there exists $1 \in R$ with $x * 1 = x = 1 * x$.

Example 2. $(\mathbb{Z}, *, +, -, 0)$ is a commutative and associative unit ring where 1 is the usual number one. Also $(\mathbb{Q}, *, +, -, 0)$ is a commutative and associative unit ring again with 1 being the number one. For a given commutative and associative unit ring R the set $R^\star$ denotes all invertible elements of R with respect to the operation $*$, e.g. $\mathbb{Z}^\star = \{1\}$ and $\mathbb{Q}^\star = \mathbb{Q} \setminus \{0\}$.

Definition3 (Field). $(F, *, +, {}^{-1}, -, 1, 0)$ is a *skew field* if:

1. $0 \neq 1$,
2. $(F, *, +, -, 0)$ is an associative unit ring with unit 1,
3. $(F \setminus \{0\}, *, {}^{-1}, 1)$ is a group.

If $(F, *, +, -, 0)$ is also a commutative ring F is simply called a *field*.

Example 3. $(\mathbb{Q}, *, +, {}^{-1}, -, 1, 0)$ and $(\mathbb{R}, *, +, {}^{-1}, -, 1, 0)$ are both fields. Let $\mathbb{M}_n$ be the set of invertible $n \times n$ matrices with $1 < n \in \mathbb{N}$. Then $\mathbb{M}_n$ is a skew field with respect to the usual matrix operations.

Remark. At the end we mention the notion of an *algebra*. Since it will not be needed in the main flow we only sketch the definition. It makes use of the notion of a *vector space* over a field and of *bilinearity*.

Definition4 (Algebra). A structure $(A, *, +)$ which

- is a vector space over a given field F with addition $+$ and
- where $*$ is a bilinear operation

is called an *algebra*.

Remark. Obviously A with $*$ and $+$ is a ring and therefore called associative, commutative, with unit if this holds for the corresponding ring. A is a *division algebra* if $A^\star = A \setminus \{0\}$. For example each field is a division algebra over itself.

3 The Generic Duplication

In this section we present the appropriate method for duplication. We start from the construction of the complex numbers on top of the real numbers. Afterwards we generalize this construction in that it can be reused for the iteration to construct also the hyper-complex numbers.

Remark. New number structures usually arose from the need to solve certain algebraic equations. For example in case of given a and b the equation $a + x = b$ has not always solutions in $\mathbb{N}$ but in $\mathbb{Z}$ and the equation $a * x = b$ for $a \neq 0$ has not always solutions in $\mathbb{Z}$ but in $\mathbb{Q}$. The equation $x^2 - 2 = 0$ cannot be solved over $\mathbb{Q}$ but it has a solution over $\mathbb{R}$.

Problem 5. There are still algebraic equations which cannot be solved over $\mathbb{R}$. The simplest example of such an equation is $x^2 + 1 = 0$. Therefore it is natural to look for a smallest extension of the field of real numbers such that this equation has a solution.

Method (cf. [OS77]). The resulting structure, the complex numbers $\mathbb{C}$, can be constructed as ordered pairs over the real numbers, i.e. $\mathbb{C} \cong \mathbb{R} \times \mathbb{R}$ with the following operations:

$$\begin{aligned}
0_{\mathbb{C}} &:= (0_{\mathbb{R}}, 0_{\mathbb{R}}) \\
1_{\mathbb{C}} &:= (1_{\mathbb{R}}, 0_{\mathbb{R}}) \\
(a,b) +_{\mathbb{C}} (c,d) &:= (a +_{\mathbb{R}} c, b +_{\mathbb{R}} d) \\
-_{\mathbb{C}}(a,b) &:= (-_{\mathbb{R}} a, -_{\mathbb{R}} b) \\
(a,b) *_{\mathbb{C}} (c,d) &:= ((a *_{\mathbb{R}} c) +_{\mathbb{R}} (-_{\mathbb{R}}(b *_{\mathbb{R}} d)), (a *_{\mathbb{R}} d) +_{\mathbb{R}} (b *_{\mathbb{R}} c)) \\
(a,b)^{-1}_{\mathbb{C}} &:= (a *_{\mathbb{R}} ((a *_{\mathbb{R}} a) +_{\mathbb{R}} (b *_{\mathbb{R}} b))^{-1}_{\mathbb{R}}, -_{\mathbb{R}} b *_{\mathbb{R}} ((a *_{\mathbb{R}} a) +_{\mathbb{R}} (b *_{\mathbb{R}} b))^{-1}_{\mathbb{R}})
\end{aligned}$$

For the pair (a,b) with $a, b \in \mathbb{R}$ it is also usual to write $a * 1 + b * i = a + b * i$ which corresponds to a representation of the vector (a,b) over the base $\{1_{\mathbb{C}} = (1_{\mathbb{R}}, 0_{\mathbb{R}}), i_{\mathbb{C}} = (0_{\mathbb{R}}, 1_{\mathbb{R}})\}$. Now i is a solution of the previously mentioned equation.

Lemma 6. *$\mathbb{C}$ is a field.*

Proof (cf. [OS77]). This can be shown by algebraic calculations verifying all necessary equations. Hereby all the calculations can be done over the real numbers. The involved equality is:

$$(a,b) =_{\mathbb{C}} (c,d) :\Leftrightarrow a =_{\mathbb{R}} c \quad \& \quad b =_{\mathbb{R}} d$$

Problem 7. Can this construction given above be generalized such that its iteration leads to the hyper-complex numbers? In its current presentation it is rather specific for the complex numbers. Furthermore some useful auxiliary operations are missing.

Method (cf. [OS77]). There are two possibilities to proceed: either provide some operations from the real numbers in all structures or just refer to the corresponding previous structure. We prefer the latter approach. Anyway $\mathbb{R}$ can always be embed.

The following auxiliary notions are useful:

$$\begin{aligned}
\iota(r \in \mathbb{R}) &:= (r, 0_{\mathbb{R}}) \in \mathbb{C} \\
\overline{(a,b)}_{\mathbb{C}} &:= (a, -_{\mathbb{R}} b) \\
|(a,b)| &:= (a *_{\mathbb{R}} a) +_{\mathbb{R}} (b *_{\mathbb{R}} b)
\end{aligned}$$

ι is the embedding of real numbers into complex numbers. $\bar{z}$ is called the *conjugate* complex number for z and $|z|$ the square of the *norm* of the complex number z.

With the help of these auxiliary operation we reformulate the construction of the complex numbers:

$$\begin{aligned}
(a,b) *_{\mathbb{C}} (c,d) &:= ((a *_{\mathbb{R}} c) +_{\mathbb{R}} (-_{\mathbb{R}}(\bar{d} *_{\mathbb{R}} b)), (d *_{\mathbb{R}} a) +_{\mathbb{R}} (c *_{\mathbb{R}} \bar{b})) \\
(a,b)^{-1}_{\mathbb{C}} &:= (\bar{a} *_{\mathbb{R}} |(a,b)|^{-1}_{\mathbb{R}}, (-_{\mathbb{R}} b) *_{\mathbb{R}} |(a,b)|^{-1}_{\mathbb{R}})
\end{aligned}$$

In this construction $\bar{a}$ is the conjugate of a real number which is the number itself. For the iteration the order of those operations is important.

Remark. The entire generic construction of the new structure in terms of a given one can be found in the appendix coded in the programming language SML. In particular we make use of the state-of-the-art module facility which allows parameterized structures. The given generic constructions corresponds to a `functor` in SML.

We call this construction `Double` since it always forms pairs from the given structure and defines the new operations in terms of the old operations. The equality of elements corresponds to the equality of all components. It should be noted that properties of the defined operations reduce to properties of the operations in the previous structure.

The actual implementation includes some further useful functions which allow an easier input and output of values. Additionally we provide the embedding of the real numbers into all the structures. Since we successively construct higher-dimensional vector spaces over the real numbers also a base of this vector space is provided.

4 Related Term Rewriting Systems

This section reviews term rewriting systems which are related to the characterization of numbers [Hul80, Geh92]. For the algebraic notions of ring and field we mention the way they can be treated with methods from term rewriting. The presented systems have been verified with help of the Larch Prover [GG91].

Remark. The chosen definition for a group and ring are equational theories. Other formulations just state the existence of an inverse element with respect to addition. With the help of the other axioms it can be shown that the inverse is unique and therefore corresponds to a function.

The properties of the real numbers are of main interest. As a first step we present a canonical system for an associative and commutative unit ring. This system can be used to decide the equality of two given terms.

Theorem 8. *The following is a canonical term rewriting system for the equational theory of an associative, commutative unit ring (i.e. modulo AC-rewriting for $+$ and $*$):*

$$x + 0 \longrightarrow x \tag{1}$$
$$x + (-x) \longrightarrow 0 \tag{2}$$
$$x * 1 \longrightarrow x \tag{3}$$
$$x * (y + z) \longrightarrow (x * y) + (x * z) \tag{4}$$
$$-0 \longrightarrow 0 \tag{5}$$
$$-(-x) \longrightarrow x \tag{6}$$
$$x + (-(y + x)) \longrightarrow -y \tag{7}$$
$$-(x + (-y)) \longrightarrow (-x) + y \tag{8}$$
$$(-x) + (-y) \longrightarrow -(x + y) \tag{9}$$
$$x * 0 \longrightarrow 0 \tag{10}$$
$$(-x) * y \longrightarrow -(x * y) \tag{11}$$

Proof (cf. [Hul80, Geh92]). The given system is produced by the Larch prover with the input:

```
declare sort Ring
declare variables x, y, z: Ring
declare operators
        __+__: Ring, Ring -> Ring
        __*__: Ring, Ring -> Ring
        -__: Ring -> Ring
        0: -> Ring
        1: -> Ring
..
set ordering polynomial
register polynomial     +       x + y + 1
register polynomial     *       x * y
register polynomial     -       x + 1
register polynomial     0       1
register polynomial     1       2
assert
        ac +;
        x + 0 = x;
        x + (-x) = 0;
        ac *;
        x * 1 = x;
        x * (y + z) = (x * y) + (x * z)
..
complete
display
```

The equational theory is preceded by a simple polynomial interpretation for the operators. This type of interpretation can also establish termination in case of AC operators. □

Remark. With the system for an associative, commutative unit ring most of the properties for the complex and hyper-complex numbers can be shown. It remains to consider the multiplicative inverse. This is the only missing component for the notion of a field.

Problem 9. Is there a relatively easy way to prove certain properties of the real numbers which involve also multiplicative inverses? One problem is the possibility of divisions by zero. On the other hand we do not want to consider the full theory of real closed fields even though it is decidable [Col75].

Method . One way out is a construction already mentioned in the introduction. Equations involving the division can be handled with pairs of ring elements like numerator and denominator (i.e. two real numbers) together with a congruence relation:

$$(a, b) \sim (c, d) :\Leftrightarrow a * d \equiv b * c$$

At the moment we do not consider the problem of the division by zero which could be handled by a constraint solving mechanism.

Example 4. We show this method coded with the help of the Larch prover including termination. It would be easy to add some code which propagates constraints, which in our case could collect all denominators.

```
declare sort Field
declare operators
        pair: Ring, Ring -> Field
        plus: Field, Field -> Field
        times: Field, Field -> Field
        neg: Field -> Field
        inv: Field -> Field
        zero: -> Field
        one: -> Field
        eq: Field, Field -> Ring
..
register polynomial     pair    x + y
register polynomial     plus    x * y
register polynomial     times   x * y
register polynomial     neg     x + 2
register polynomial     inv     x + 1
register polynomial     zero    4
register polynomial     one     5
register polynomial     eq      x * y
declare variables a,b,c,d : Ring
assert
        plus(pair(a,b), pair(c,d)) = pair((a*d) + (b*c) , b * d);
        times(pair(a,b), pair(c,d)) = pair(a * c, b * d);
        neg(pair(a,b)) = pair(- a, b);
        inv(pair(a,b)) = pair(b, a);
        zero = pair(0, 1);
        one = pair(1, 1);
        eq(pair(a,b),pair(c,d)) = (a * d) + (- (b * c))
..
complete
display
```

In any case we want to apply this method only for checking the inverse element with respect to multiplication.

Lemma 10. *The following hold:*
i) $r = 0 \iff |r| = 0$ *for all* $r \in \mathbb{R}$,
ii) $z = 0 \iff |z| = 0$ *for all* $z \in \mathbb{C}$,
iii) $\sum_i x_i^2 = 0 \iff \forall i : x_i = 0$ *for all* $x_i \in \mathbb{R}$.

Remark. These properties are obvious. They guarantee that for all the constructed structures on top of the real number the function `reci` is well defined for all elements different from zero. Since we will not consider other questions we drop a constraint propagation scheme.

5 Combining Computation and Deduction

This section shows how the same functor is instantiated twice. One instantiation results in a structure to be used for computations. Another instantiation with symbolic terms allows to check certain properties of the resulting structures which is done with the help of a simple interface to the externally running Larch prover.

We are interested in both:

- in computations in the resulting structures,
- in deductions about the resulting structures.

Therefore it is important that both work with the same code. On the other hand we start out from a base structure - here the real numbers - where we just assume certain properties to hold.

Firstly we show the instantiation of the functor shown in the appendix with a structure to allow computations. To do this we simply supply it with the built-in data type of real numbers:

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

        exception Dimension
        fun gl2f([r]) = r
          | gl2f(_) = raise Dimension
        fun f2gl(r) = [r]
        fun f2s(r) = Makestring.realToSciStr(r, 3)

        val plus = Real.+
        val zero = 0.0
        val neg = Real.~
        val times = Real.*
        val one = 1.0

        exception ZeroInverse
        val sprod = Real.*
        fun conj r = r
        fun norm r = Real.*(r,r)
        fun reci(r) = ((one / r) handle Div => raise ZeroInverse)
    end

structure Rc = Reals

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

The functions `gl2f` and `f2gl` simplify the input and output of values and the functions `f2s` produces strings which can be printed.

With the help of these instantiations computations over the corresponding domain can be performed. For example:

```
- val t1 = Cc.times(Cc.gl2f([0.0,1.0]),
=                  Cc.gl2f([0.0,1.0]));
val t1 = P (~1.0,0.0) : Cc.$
- val t2 = Qc.times(Qc.gl2f([0.0,1.0,0.0,0.0]),
=                  Qc.gl2f([0.0,0.0,1.0,0.0]));
val t2 = P (P (0.0,0.0),P (0.0,1.0)) : Qc.$
- print(Oc.f2s(Oc.times(Oc.gl2f([1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0]),
=                      Oc.gl2f([1.0,~1.0,~1.0,~1.0,~1.0,~1.0,~1.0,~1.0]))));

= (((8.000E0,0.000E0),(0.000E0,0.000E0)),((0.000E0,0.000E0),(0.000E0,0.000E0)))
```

With the iterated application of the same functor all the necessary structures have been created.

Secondly we are interested in deductions about the resulting structures. To do this we define a type of symbolic terms as shown in the following signatures:

```
signature AbstractRing =
    sig
        datatype term = Zero | One | Var of string |
            Neg of term | Plus of term * term | Times of term * term
        val t2s : term -> string
        (* ... *)
        datatype sem_eq = EnoughPoints | NotIn of string -> BigInt.bigint
            | VarConflict of string
        val equal : term * term -> sem_eq
        (* ... *)
        val lpEqual : term * term -> bool
    end

signature AbstractField =
    sig
        datatype term = Zero | One | Var of string | 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 *)
        (* ... *)
        type sem_eq
        val equal : term * term -> sem_eq * ring list

        val lpEqual : term * term -> bool * ring list
    end
```

The main idea is to encode all operations with a corresponding symbolic term either as a ring element or a field element depending on the deduction to be performed.

Furthermore the function `t2s` transforms symbolic terms into strings. These strings can be used for the communication with the external Larch Prover. The coding of field elements as described in the previous section is already done inside SML for efficiency reasons.

These symbolic terms are exploited in the definition of a new structure of completely symbolic terms:

```
structure SymbolicReals =
    struct
        (* ... *)
        structure F =
            struct
                type ground = abstractField.term
                val dimension = 1
                type $ = abstractField.term
                fun g2f x = x
                val base = [abstractField.One]

                exception Dimension
                fun gl2f([x]) = x
                  | gl2f(_) = raise Dimension
                fun f2gl(x) = [x]
                val f2s = abstractField.t2s

                fun plus(l,r) = abstractField.Plus(l,r)
                val zero = abstractField.Zero
                fun neg(t) = abstractField.Neg(t)
                fun times(l,r) = abstractField.Times(l,r)
                val one = abstractField.One

                exception ZeroInverse
                fun conj(t) = t
                fun norm(t) = abstractField.Times(t,t)
                fun reci(t) = abstractField.Reci(t)
            end
        (* ... *)
    end
```

The substructure `F` as a symbolic structure for a field can now be used as parameter for the functor `Double`. The so created symbolic structures are used for reasoning. All constructions are generic in the base structure of a symbolic field.

In order to prove properties for the complex and hyper-complex numbers queries to the Larch prover are generated automatically. Therefore it was necessary to establish a communication between SML and the Larch Prover which can be seen in the following signature:

```
signature Lp =
    sig
        exception LpRunning
        exception LpNotRunning
        val start : unit -> unit
        val terminate : unit -> unit
```

```
        val send : string -> unit           (* without a newline *)
        val receive : unit -> string list   (* in reversed order *)
        val reset : unit -> unit

        val format : string * int * char list -> string
        val normalize : string * char list -> string      (* last output *)
        val equal : string * string * char list -> bool   (* only normal *)

        val setProgram : string -> unit
        val exeProgram : string -> unit
    end
```

This interface was kept as simple as possible.

For the concrete implementation we have been using:

- the SML/NJ compiler [1] version 108 which includes also a compiler manager,
- `IO.execute : string * string list -> instream * outstream` which allows the creation of and communication with an external process,
- the datatype `string` for sending and receiving information.

Furthermore it is possible to log all actions of the Larch prover to a file and set a trace level. This together with the possibility to create script files allows a sufficient control of the ongoing interaction.

With the help of the symbolic terms queries for the Larch Prover are generated automatically. The output is transformed into an answer within SML but also the log file always can be inspected. In this way the following proposition can be verified:

Proposition 11. *The following hold:*
i) The complex numbers $\mathbb{C}$ are a field.
ii) The quaternions $\mathbb{H}$ are a skew field.
iii) The Cayley numbers $\mathbb{O}$ are a ring with division.

Since all the constructed structures of complex and hypercomplex numbers are algebras it proves to be useful to provide a base. This base can be used to perform a symbolic evaluation of a test set generated from this base. In particular it is useful to construct counterexamples.

It should be mentioned that the generated queries get rather large on the higher dimensional structures. A main problem became the necessity of AC unification. Therefore we additionally implemented a semantic decision procedure for multivariate polynomials over an integral domain.

The first observation is that terms over a ring are just polynomials. Since we are interested in the real numbers we always have the integers as a substructure. Then the following theorem can be applied:

Theorem 12 (cf. [OS77]). *Let I be an* integral domain, *$\alpha, \beta \in I[x]$ and furthermore $max(deg(\alpha), deg(\beta)) < m \in \mathbb{N}$. If there exist m different elements $r_i \in I$ such that $\forall i \in \{1 \ldots m\} : \alpha(r_i) = \beta(r_i)$ then $\alpha = \beta$.*

$\mathbb{R}$ and $\mathbb{Z}$ are also integral domains. Furthermore from I integral domain follows that also $I[x]$ is an integral domain. Thus we have a bound for a finite set of points such that the semantic test of equality in these points solves the decision problem (also in the multivariate case).

We conclude by showing a part of the final interface for the user which is described by the signature `Symbolic`:

```
signature Symbolic =
    sig
        structure R :
            sig
                include FieldLike sharing type ground = abstractRing.term
            end
        structure F :
            sig
                include FieldLike sharing type ground = abstractField.term
            end

        type sem_eq
        type provedBy
        val combine : provedBy * (unit -> provedBy) -> provedBy

        val SemR : (R.ground * R.ground) -> provedBy
```

[1] Copyright 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996 by AT&T Bell Laboratories

```
    val LpR : (R.ground * R.ground ) -> provedBy
    val SemF : (F.ground * F.ground) -> provedBy
    val LpF : (F.ground * F.ground) -> provedBy

    val proveR : int * (R.$ list -> (R.$ * R.$)) *
        ((R.ground * R.ground) -> provedBy) -> provedBy
    val proveF : int * (F.$ list -> (F.$ * F.$)) *
        ((F.ground * F.ground) -> provedBy) -> provedBy
    (* ... *)
end
```

The constants `SemR` and `LpR` are the two methods for reasoning about ring elements, `SemF` and `LpF` reason over field elements. The functions `proveR(n, f, m)` and `proveF(n, f, m)` check the equality of two elements in n unknowns where the elements can be produced by providing a list of lenght n to f and the comparison is done by method m.

6 Discussion

As benefits of the suggested framework wee see the following:

1. it allows fast experiments with new structures by examining examples,
2. it offers the possibility to perform reasoning in a restricted form but general enough to characterize the number system,
3. it is transparent in that the external proofs can be verified from the log files.

Again it should be noted that the functor code for computations and for the deduction was the same. The proofs proceed with respect to properties valid for real numbers which are hard coded.

We also want to point out that the involved deductions are not trivial. For the step from $\mathbb{R}$ to $\mathbb{C}$ we need to perform 9 operations over $\mathbb{R}$ for the multiplication in $\mathbb{C}$. This number gets accumulated by the repeated application of the `Double` functor.

The communication overhead with the external prover does not play an important role since the performed deductions are rather time consuming. Surely it makes sense to contrast this pure syntactic approach with semantic methods which provide means for deciding equality for a given structure. In our particular case the inspection of the symbolic evaluation of terms at a generated point set or at points from the base was useful.

Even though in our example there is an obvious decision method, on the other hand this way can be very costly. *Meta proofs* become a real necessity. One direction here could be to find a parallel to the functor `Double` on the reasoning side to simplify deductions in the higher-dimensional structure.

A simple example that certain properties are propagated by the construction is the commutativity of addition. This can be seen just from the definition of addition. Another time it has to do with the fact that all the structures are indeed algebras.

7 Conclusions and Future Work

We have presented an interesting non-trivial example combining computations and deductions. The example investigates complex numbers, quaternions, Cayley numbers starting from real numbers with the help of a single generic construction. This construction can be used either for computation in the new structure or to produce a symbolic instantiation for reasoning.

The provided functor can also be reused for computations in several ways. Here we have shown its iteration starting from the built-in real numbers or starting from a symbolic structure. Instead the input could be the field of rational numbers leading to rational complex and hyper-complex numbers or the input could be computational real numbers leading to computational complex and hyper-complex numbers.

The studied number systems can be classified with the help of mainly equational theories like those of rings and fields. Since these notions allow the use of term rewriting systems and canonical systems are known for them it was very natural to incorporate an equational prover - here the Larch prover. SML is not a pure functional language and allows imperative IO - therefore it was easy to communicate with the external prover via streams using strings as the common data structure.

The Larch Prover was chosen because of its functionality and the easy interface. This prover supports several levels of trace and can report all given commands to a file. Polynomial interpretations are provided as a means to prove termination of term rewriting systems (also AC).

The suggested framework is well suitable to study different number systems, to determine their properties expressible by equations, and to perform test computations. Though the actual insight consists of finding the underlying general constructions like the duplication method in our example. This part heavily depends on the correct human intuition.

As future work we plan to work on further examples of general methods used in the construction of numbers as outlined in the introduction. They can be bundled into a SML library. Furthermore these computational structures are accompanied by tools to support related reasoning mechanisms suitable for various experiments but also for the teaching to students and newcomers in this area.

References

[Bac91] L. Bachmair. *Canonical Equational Proofs.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

[Col75] G. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Proceedings of the Second Conference on Automata Theory and Formal Languages*, number 33 in Lecture Notes in Computer Science, pages 134–183, 1975.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier, MIT Press, 1990.

[Geh92] W. Gehrke. Detailed Catalogue of Canonical Term Rewriting Systems Generated Automatically. Technical Report 92–64, Research Institute for Symbolic Computation, Linz, Austria, 1992.

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

[HC96] K. Homann and J. Calmet. Structures for Symbolic Mathematical Reasoning and Computation. In *Proceedings of the Fourth International Symposium on Design and Implementation of Symbolic Computation Systems, DISCO-96*, Lecture Notes in Computer Science, 1996. forthcoming.

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

[KKS96] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra with Proof Planning. In *Proceedings of the Fourth International Symposium on Design and Implementation of Symbolic Computation Systems, DISCO-96*, Lecture Notes in Computer Science, 1996. forthcoming.

[Klo92] J.W. Klop. *Handbook of Logic in Computer Science*, volume 2, chapter Term Rewriting Systems, pages 2–116. Oxford Science Publications, 1992.

[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.

[OS77] A.L. Oniščik and R. Sulanke. *Algebra und Geometrie.* Studienbücherei. Deutscher Verlag der Wissenschaften, Berlin, 1977.

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

A Code for Duplication in SML

```
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 plus : $ * $ -> $
        val zero : $
        val neg : $ -> $
        val times : $ * $ -> $
        val one : $

        exception ZeroInverse
        val sprod : ground * $ -> $ and norm : $ -> ground
        val conj : $ -> $ and reci : $ -> $
    end

functor Double(structure G : FieldLike
               structure F : FieldLike
               sharing type F.ground = G.$) :
    sig
        include FieldLike sharing type ground = G.$
    end =
    struct
        type ground = G.$
        val dimension = 2 * F.dimension
        datatype $ = P of F.$ * F.$
        fun g2f g = P(F.g2f(g), F.zero)
        val base = let fun aux [] = []
                       | aux (h::t) = P(h, F.zero)::P(F.zero, h)::(aux t)
                   in   aux F.base   end

        exception Dimension
        fun gl2f gl =
            let val l = List.length(gl)
                fun aux rl 0 ll = P(F.gl2f(List.rev(ll)), F.gl2f(rl))
                  | aux (h::t) n ll = aux t (n - 1) (h::ll)
            in
                 if l = dimension then aux gl (F.dimension) []
                 else raise Dimension
            end
        fun f2gl (P(l, r)) = F.f2gl(l) @ F.f2gl(r)
        fun f2s (P(l, r)) = "(" ^ F.f2s(l) ^ "," ^ F.f2s(r) ^ ")"

        fun plus((P(a,b)), (P(c,d))) = P(F.plus(a,c), F.plus(b,d))
        val zero = P(F.zero, F.zero)
        fun neg(P(a,b)) = P(F.neg(a), F.neg(b))
        fun times((P(a,b)), (P(c,d))) =
            P(F.plus(F.times(a,c),F.neg(F.times(F.conj(d),b))),
              F.plus(F.times(d,a),F.times(b,F.conj(c))))
        val one = P(F.one, F.zero)

        exception ZeroInverse = F.ZeroInverse
        fun sprod(s, (P(a,b))) = P(F.sprod(s,a),F.sprod(s,b))
        fun conj(P(a,b)) = P(F.conj(a), F.neg(b))
        fun norm(P(a,b)) = G.plus(F.norm(a), F.norm(b))
        fun reci(v) = sprod(G.reci(norm(v)), conj(v))
    end
```

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