Symbolic Computation: Computer Algebra and Logic *

Bruno Buchberger
Research Institute for Symbolic Computation
A4232 Schloß Hagenberg, Austria
Bruno.Buchberger@risc.uni-linz.ac.at

September 11, 1996

Abstract

In this paper we present our personal view of what should be the next step in the development of symbolic computation systems. The main point is that future systems should integrate the power of algebra and logic. We identify four gaps between the future ideal and the systems available at present: the logic, the syntax, the mathematics, and the prover gap, respectively. We discuss higher order logic without extensionality and with set theory as a subtheory as a logic frame for future systems and we propose to start from existing computer algebra systems and proceed by adding new facilities for closing the syntax, mathematics, and the prover gaps. Mathematica seems to be a particularly suitable candidate for such an approach. As the main technique for structuring mathematical knowledge, mathematical methods (including algorithms), and also mathematical proofs, we underline the practical importance of functors and show how they can be naturally embedded into Mathematica.

1 The Next Goal for Symbolic Computation

By the work of researchers in various areas, we now have powerful tools available that support various aspects of problem solving by computer:

- numerical libraries
- special purpose systems for simulation, CAD, robotics, neural network design, ...
- "symbolic computation" systems for computer algebra, computer analysis, ...
- rewrite labs, logic programming systems, constraint solvers, ...

*Invited talk at the "Frontiers of Combining Systems" Conference in Munich, March 26-29, 1996. To appear in the Proceedings of this conference (F. Baader, K.U. Schulz eds.). Applied Logic Series, The work described in this paper is partly sponsored by Fujitsu Labs, ISIS Group, Numazu, Japan.

- special and general theorem provers, theorem generators, theorem checkers, ...
- program verification, transformation, and synthesis systems, ...
- advanced software technology tools in all theses systems,
- graphics, animation, sound, typesetting, notebook, and hyperlink tools in these systems,
- links between these systems, access through the web, electronic user communities, ...

The availability of these systems has drastically enhanced our problem solving potential. However, we want more. Who is "we"? In this paper, I address people who

- explore,
- invent,
- apply,
- teach,
- study, and
- publish

mathematics or, in other words, people who are "doing" mathematics. In this paper, I do not address people who are casual users of mathematics as a "black box". I think that the next natural goal "we" should go for is to do all of mathematics in one system. Here, by a "system", I mean both

- a *logical* system, which should be a sound and uniform frame for all of our mathematical activities and
- a *software* system, which supports these activities.

Although there are systems on the market that advertise explicitly that they are systems for "doing" mathematics, see (Wolfram 1988), I think that there are still significant gaps between what we have and what we want. Basically, I see four gaps:

- the logic gap,
- the syntax gap,
- the mathematics gap, and
- the prover gap.

I will analyze these gaps in Section 2. Then I will argue, in Section 3, why Mathematica seems to be a good starting point for developing a rapid prototype system that may overcome these gaps. (I will not argue, however, that Mathematica is "the" ideal future symbolic computation system. In my view, this would only be possible if the designers of Mathematica made some basic changes in the design of their system.) In Sections 4 to 7, I will then sketch my proposal how one may overcome the logic, the syntax, the mathematics, and the prover gap, respectively.

2 The Present Gaps

2.1 The Logic Gap

At present, officially, there is one uniform logic system in use as one uniform frame of mathematics (although, inofficially, this frame is not at all applied uniformly in the every-day practice of mathematicians): The system of "Bourbakism", which is first order predicate logic plus set theory formulated as a first order theory (which is called "Zermelo Fraenkel set theory"). The problem with the Bourbakistic system as a frame for all of mathematics, including "algorithmic" mathematics, is that in this system functions are sets. Hence, by the extensionality axiom for sets, functions defined by different algorithms but having identical input/output behavior are identical. This is not appropriate for the needs of algorithmic mathematics where we do not only want to discuss the input/output behavior of functions but also properties of the definitions of functions (the "algorithms") and properties of their computational behavior, for example their complexity. Of course, we could circumvent this problem by defining a specific programming language in the frame of ZF (Zermelo-Fraenkel set theory). For this, what we had to do is basically to define a binary function I, the "interpreter" of the programming language, within ZF. (The object $I(p, d)$ is the "result of applying the program p of the given programming language to the input data d".) Then, of course, it is well possible that we have distinct programs $p_1 \neq p_2$ with identical input/output behavior, i.e. such that $\forall d(I(p_1, d) = I(p_2, d))$.

However, this solution is not very natural because, for example in a textbook on algorithmic polynomial ideal theory, we would not like to spend an additional chapter on introducing an extra programming language. Rather, we would like to use certain predicate logic formulae, for example equalities, directly for defining certain algorithmic functions and predicates.

As an alternative we could use higher order predicate logic. However, the problem remains as soon as we introduce an extensionality axiom in such a logic as this is done in most of the usual textbooks on higher order predicate logic, see for example (Andrews 1986). Therefore, in recent years, some authors proposed to use higher order predicate logic without extensionality as a frame for mathematics, see for example CIC (Huet 1995) and NuPRL (Constable 1995). This is a very promising approach. The problem with the present implementations of this idea is that it seems to be quite hard to carry over the results from Bourbakistic mathematics into these systems. This is, however, important because these results are not only of aesthetical value but are indispensable for specifying problems, and describing a hierarchy of increasingly powerful solutions to problems, even in algorithmic mathematics. For example, when describing the membership decision problem for polynomial ideals, it is necessary first to define the non-algorithmic notion of "ideal" and "residue class domain". Also, when developing an algorithmic solution for this problem, for example by "Groebner bases" (Buchberger 1985), it is absolutely necessary to carry out proofs for the correctness of the solution that involve non-algorithmic concepts of set theory. Also, there is still a practical problem with the present implementations of higher order predicate logic without extensionality: Their practical potential for algebraic computation is not very high.

2.2 The Syntax Gap

Notation used in "mathematical" software systems is still far from the usual mathematical notation. For example, in Mathematica, the function definition

```
f[ { }] := { }

f[ { x1_, x2___}] := Prepend[ f[ { x2}], { x1, x1}]
```

and the function call

```
Integrate[ Sin[ Sqrt[ y + a^2]], y]
```

look quite different from the corresponding formulae

$$\begin{aligned} f(\langle\rangle) &:= \langle\rangle \\ f(\langle x_1, x_2, \ldots\rangle) &:= \langle x_1, x_1\rangle \frown f(\langle x_2\rangle) \end{aligned}$$

and

$$\int \sin(\sqrt{y + a^2})\, dy$$

as they may appear in an ordinary mathematical text. Syntax, however, *is* quite important for practical problem solving.

In fact, the next version of Mathematica will provide "ordinary" syntax for mathematical input and output of amazing sophistication, see (Soiffer 1995). However, some unpleasant gaps will still remain. For example, still, brackets will be used for function application, braces will denote tuples instead of sets, and underscores will be used in order to declare variables and "sequence variables".

2.3 The Mathematics Gap

For algorithmic mathematics, the traditional algebraic notions are too coarse. For example, the following three algebraic structures are "equal" from the point of view of algebra:

Residue Domain Modulo 3:
Carrier:

$$\{\{0 + 3x \mid x \in \mathbf{I}\}, \{1 + 3x \mid x \in \mathbf{I}\}, \{2 + 3x \mid x \in \mathbf{I}\}\}.$$

Operation:

$$\{y + 3x \mid x \in \mathbf{I}\} \oplus_3 \{z + 3x \mid x \in \mathbf{I}\} := \{y + z + 3x \mid x \in \mathbf{I}\}.$$

Simplified Residue Domain Modulo 3:

Carrier:

$$\{0, 1, 2\}.$$

Operation:

$$y +_3 z := \text{remainder}(y + z, 3).$$

"Smallest" Simplified Residue Domain Modulo 3:
Carrier:

$$\{-1, 0, 1\}.$$

Operation:

$$y \overline{+_3} z := \text{smallest remainder}(y + z, 3).$$

From the algorithmic point of view the three structures are significantly different. The first structure is non-algorithmic. It results from the integers with addition by application of the non-algorithmic "functor" "residue class formation modulo a congruence relation". The resulting carrier contains elements that are infinite sets and the operation operates on infinite sets and produces infinite sets. The second and the third structures are both algorithmic, i.e. the objects of the carrier are finitary and can be stored in a computer and the operations are computer-realizable. Both structures result from the integers with addition by applying the functor "simplification modulo a canonical simplifier". However, the two structures are not identical and, in fact, the complexity of the two operations is (slightly) different.

When we "do" mathematics, we want to live in both worlds:

- the world of (non-algorithmic) theorems and proofs,
- the world of algorithms and computation.

The World of Theorems and Proofs: For example, when doing Groebner bases theory, we will start with the *definition*

$$G \text{ is a Groebner basis} :\Longleftrightarrow \forall f \in \text{Ideal}(G)\ (f \longrightarrow_G 0)$$

and will want to prove the following *theorem*:

$$G \text{ is a Groebner basis} \Longleftrightarrow \forall f, g \in G\ (\text{S–polynomial}(f, g) \longrightarrow_G 0).$$

The proof will involve various non-algorithmic proof techniques of predicate logic, in particular those for handling quantifiers, and non-algorithmic concepts from set theory.
The World of Algorithms and Computation: In the example of Groebner bases theory, the above theorem can be used unchanged as an *algorithm*

$$G \text{ is a Groebner basis} :\Longleftrightarrow \forall f \in \text{Ideal}(G)\ (f \longrightarrow_G 0)$$

The following proposition can now be "evaluated" by using the above theorem/algorithm and a subset of the predicate logic proof techniques, namely equational logic, as "evaluation machine": When we enter

$$\{x^2y - 3x, xy^2 - 2xy + 4\} \text{ is a Groebner basis.}$$

The result of the evaluation will be "no".

In the future, we would like to live in both worlds within one uniform logic and software system!

2.4 The Prover Gap

The available universal theorem provers (e.g. those based on resolution) are general and, therefore, often too inefficient for supporting practical theorem proving in a wide range of mathematical areas. Also, most times, they are available only as stand-alone systems that are not well connected with computer algebra/analysis systems.

The same is true also for special provers (proof checkers, proof generators) that may be quite efficient for particular mathematical theories. However, again, they are rarely integrated with current algebra/analysis software systems.

What we need is the integration of both universal and special computer-supported theorem proving with the current computer algebra/analysis systems so that, when "doing" mathematics, one can switch between computer-supported theorem/algorithm development and algorithm application.

3 Mathematica is a Good Starting Point

For filling the gaps analyzed in the preceding section, i.e. for supporting all aspects of "doing" mathematics or, in other words, for reaching a new level of sophistication in "symbolic computation" by combining computer algebra and logic, one can adopt one of the following strategies:

1. One can start from successful proving systems like CIC, NuPRL, etc. and add the potential of current computer algebra systems like Maple, Mathematica, Macsyma etc.

2. One can start from one of the practically powerful computer algebra systems and add the potential of current provers.

3. One could design a completely new system.

I do not suggest the third possibility, at least not at this moment, because tremendous work would go into repeating the effort for implementing the wonderful man-machine interfaces current algebra and proof systems already have. The construction of completely new systems or, at least, completely new systems kernels may become reasonable and necessary in the future, after we will have experimented with various early prototypes of combined algebra/logic systems. Rather, I suggest that, at the moment, these experiments should be based on either the first or the second possibility.

In this paper, I argue for basing such an experiment for constructing an early prototype for a combined algebra/logic system on Mathematica. The main reasons can be structured according to the analysis of gaps described in Section 2:

1. Although apparently this was not the intention of the Mathematica designers, a close inspection of the Mathematica language reveals that, in fact, the innermost part of *Mathematica can be viewed as nothing else than directed higher order equational logic without extensionality* and, thus, Mathematica can be viewed as a programming language inside logic. This feature is unique among all existing computer algebra systems.

2. Mathematica has an amazing man/machine interface. Its next version will also have wonderful typesetting facilities and, what is more, there will be facilities that *allow the user to define his own syntax* so that higher order logic with quantifiers and the programming part of this language can be presented as executable code to the system.

3. Using Currying and the module construct, *"functors" can be programmed within Mathematica.* This fact is little known but is essential for structuring non-algorithmic and algorithmic mathematics within a uniform system frame.

4. The directed equational logic facilities of Mathematica are available "twice" in the system: First, at the basic level, they constitute the programming language. Second, at the metalevel, these facilities can also be applied to programs (sets of equalities) at the first level. This allows one, within Mathematica, to program "provers" for proving properties of the programs defined on the first level. Together with the functor principle, well structured provers that combine special provers tailored to the various functors can be built up.

Of course, it may turn out that basing a uniform logic frame for all of mathematics on top of Mathematica may result in intolerable loss of speed because some of the more advanced language features must be simulated by the available features in Mathematica. If this is the case, in a later stage, a new and specially designed kernel should replace the present Mathematica kernel. Thus, we repeat, the present proposal is only meant as a proposal for implementing a rapid prototype of an algebra/logic system as quickly as possible in order to be able to study the logical, mathematical, algorithmic, and practical implications and gain the necessary experience in using such a combined system for inventing and presenting mathematics.

In the next sections we describe in more detail how we want to overcome the logic, syntax, mathematics, and prover gap by adding features to higher order logic and implementing them in the frame of Mathematica.

4 Overcoming the Logic Gap

We start from higher order logic without extensionality. The fundamental concept of this logic is "application" of an object (a "function") f to some other object x, denoted by

$$f(x).$$

Since we do not have extensionality, this (logic instead of set-theoretic) notion of a function remains "fine grain" (i.e. functions with the same input/output behavior are not necessarily identical), which is indispensable for the algorithmic aspects of mathematics.

In this paper, we do not discuss the subtle question of using "types" in such a logic. Of course this question is very important for obtaining sound proof rules in such a logic. This question is deemed to be important also for the practical work within such a system when building up the "tower of mathematical domains". However, in Section 6, we show how functors can be built up in such a way that we do not need types for

the practical mathematical work within the system. Rather, we work with one uniform equality all over the system and, in compensation, we give explicit descriptions of the objects in the various domains.

We observe that (the innermost kernel of) Mathematica can be viewed as an (efficient) implementation of the (directed) equational part of higher order logic without extensionality. For example, the induction definition (in the syntax of Version 2.2 of Mathematica)

```
apply[ f_, { x1_}]  :=  x1
apply[ f_, { x1_, x2___}] := f[ x1, apply[ f, { x2}]]
```

can be viewed as two higher-order logic equalities and, at the same time, as an algorithm whose computations for variable-free input terms are nothing else than proofs using substitution and replacement as inference rules and using the equalities in the direction from left to right. (Do not bother about the strange syntax of the above Mathematica equalities. In the next version (3.0) of Mathematica, the user will be able to define his own syntax and may choose to use the usual mathematical syntax. Variables with three underscores are "sequence variables", i.e. variables for which arbitrary finite sequences of terms may be substituted. In principle, sequence variables are dispensable. However, we think that they are practically attractive and useful. Thus, we propose to have them in our system. Appropriate changes must then be made to the usual proof rules of higher order logic.)

Thus, higher order logic without extensionality contains a practical programming language (efficiently implemented as Mathematica) as a sub-language. The question is: How can we retain Bourbakistic mathematics within this system?

I think that, in addition to the fundamental "$\in$" predicate, it suffices to add one more predicate "is set" so that we can restrict all set-theoretic axioms to those objects that satisfy "is set". For example, extensionality can be stipulated for objects that are sets:

$$A \text{ is a set} \wedge B \text{ is a set} \Longrightarrow (A = B \Longleftrightarrow \forall x(x \in A \Longleftrightarrow x \in B).$$

Similarly, the other axioms of set theory guaranteeing the existence of various sets constructed from given sets can be formulated. In the same way, we can also introduce the set braces as special quantifier. From there on, we can develop (carry over) all of Bourbakistic mathematics including the set-theoretic (as opposed to the above logic) notion of "function" and "function application". Of course, for set-theoretic function application, we have to introduce a new notation. For example, we can introduce the binary function constant "`" and may write "$f`x$" for "the set-theoretic function f applied to x". One may then prove for example, within the system, that if f is a set-theoretic function and x is in the domain of f then the pair $(x, f`x)$ is in f. Note again that extensionality will remain to be restricted for the set-theoretic notion of function application whereas, by intention, it is not available for the original, logical, notion of function application. For example, if

$$\begin{aligned} f(x) &:= x + 1 \\ g(x) &:= x + 2 - 1 \end{aligned}$$

then, of course,

$$\forall x\ f(x) = g(x)$$

and also, for example,

$$\{(x, f(x)) \mid x \in \mathbf{N}\} = \{(x, g(x)) \mid x \in \mathbf{N}\}$$

can be proved. However,

$$f = g$$

cannot be proved in the system.

5 Overcoming the Syntax Gap

The syntax used in the present version of Mathematica (Version 2.2) is quite different from the usual mathematical notation. The next version of Mathematica (Version 3.0) will have an amazingly powerful syntax that comes quite close to ordinary mathematical notation. It will have TEX quality, it is wysiwyg *and* at the same time is formal, i.e. the formulae allowed in Mathematica have (formal although not formally defined) semantics as executable code and new formulae can be created whose semantics can be defined. For example, if one does not like that Mathematica uses braces for denoting tuples rather than sets, the following instruction will introduce angle brackets for denoting tuples:

```
MakeExpression[ RowBox[ { "<", x___, ">"}], StandardForm] :=
     MakeExpression[ RowBox[ { "{" x, "}"}], StandardForm]
```

A similar instruction will also let Mathematica print angle brackets instead of braces. Similarly, we can teach the system to take the set braces for denoting set construction instead tuple construction, to use parentheses instead of brackets for denoting function application, and to replace the use of underscores for identifying variables and sequence variables by some other convention, see the examples in Section 6.

6 Overcoming the Mathematics Gap

6.1 Building Up Mathematics by Functors

Both from a structural and a problem solving point of view, I think it is important to build up computer-supported mathematics by "functors" that construct new domains from given ones. Structuring mathematics by functors has also an immediate implication for structuring provers as will be made explicit in Section 7.

The functor view of mathematics can equally well be applied to both non-algorithmic and algorithmic mathematics and, in fact, it is the unifying view for both worlds. As explained above non-algorithmic mathematics and algorithmic mathematics are strongly intermingled when "doing" mathematics and, often, the transition from the non-algorithmic to the algorithmic context is the actual challenge of mathematics and of course is the main challenge of what is called "symbolic computation".

6.2 Representing Domains

We view a domain D to be a function that defines functions for certain "operators". For example, the following object Z is a simple domain:

$$\begin{aligned} Z(o) &:= Plus, \\ Z(\epsilon) &:= 0, \end{aligned}$$

where "Plus" and "0" are the "built-in" addition and the built-in integer zero of our system, which as we said above contains the innermost kernel of Mathematica. For syntactic simplicity, we introduce the convention that "o_D" etc. stands for "$D(o)$" etc. Thus, we could write, for example,

$$2\ o_Z\ 2$$

which, with the above definitions, evaluates to

$$4.$$

Note that such a notation will be perfectly possible in a user-modified syntax of Mathematica 3.0 and the above evaluation can, hence, be done automatically in Mathematica.

Also we will add the following unary operator "ϵ" to any domain with the convention that, for any domain D, $D(\epsilon)$ is a decision function which yields "True" for exactly the objects we consider to be in the "carrier" of D. For example, we could define

$$Z(\epsilon) := \text{IntegerQ}$$

where "IntegerQ" is the built-in decision algorithm for integers. Note that the symbol "ϵ" is different from the symbol "$\in$" that denotes the element predicate of set theory.

6.3 Representing Functors

Now, in our terminology (which is basically the terminology used, for example, also in ML), a functor is just a function that maps domains into domains. Thus, a functor is an object F that takes D as an argument and produces $F(D)$ with the view that $F(D)$ can now be applied to any operation symbol o yielding an operation in the domain $F(D)$. In particular, $F(D)(\epsilon)$ is a decision function for the objects which we want to be in the carrier of $F(D)$. Also, normally, one will think of any operation $F(D)(o)$ to be reasonably applied only to objects in the carrier of $F(D)$, i.e. objects for which $F(D)(\epsilon)$ yields "True".

For clarifying notation, let us first give a trivial example: We define a functor F that, for any given domain D with any operations o defines the Cartesian product. Staying within higher order predicate logic, this functor could be defined as follows:

$$\begin{aligned} &F(D) := \text{the } N \text{ such that} \\ &\qquad \forall d_1, d_2\ \ \langle d_1, d_2\rangle \epsilon_N :\Longleftrightarrow d_1 \epsilon_D \wedge d_2 \epsilon_D, \\ &\qquad \forall o, d_1, d_2, e_1, e_2\ \ \langle d_1, d_2\rangle o_N \langle e_1, e_2\rangle := \langle d_1 o_D e_1, d_2 o_D e_2\rangle. \end{aligned}$$

Instead, we introduce a new quantifier "Functor" that binds N and also the other quantified variables that appear in the above definition so that the following definition can be seen as nothing else than an abbreviation of the above definition:

$F(D_.) :=$
Functor $(\langle N, d_1, d_2, e_1, e_2, o\rangle,$

$\langle d_{1.}, d_{2.}\rangle \epsilon_N :\Longleftrightarrow d_1 \epsilon_D \wedge d_2 \epsilon_D;$
$\langle d_{1.}, d_{2.}\rangle o_{.N} \langle e_{1.}, e_{2.}\rangle := \langle d_1 o_D e_1, d_2 o_D e_2\rangle;$

N
$).$

Note that, at certain places, we use a dot after a symbol as a notation for declaring the symbol to be a variable and not a constant. The respective convention is an adaptation to the syntax of Mathematica and has no deeper relevance. Note, in particular, that "o" is a variable in this definition. Thus, the definition applies to any operator o in the "signature" of D. We could define the signature of a domain explicitly. We omit this in order not to overload the presentation in this expository paper.

Now, as a matter of fact, the construct "Module" of Mathematica, if restricted to algorithmic operations, has exactly the semantics of the "Functor" construct as defined above. Thus, when restricted to algorithmic functors, application of this construct yields executable code in Mathematica.

6.4 An Example of a Non-Algorithmic Functor

We define a functor that "constructs" the residue class domain of a domain D that is equipped with a binary operator "$\sim$" (which normally will be a congruence relation):

Residue-Domain$(D_.) :=$
Functor$(\langle R, r, s, d, e, o, C\rangle,$

$r_. \ \epsilon_R :\Longleftrightarrow \exists d \ \epsilon_D \ (r = C(d));$
$r_. \ o_{.R} \ s_. := C(\text{Choose}(r) \ o_D \ \text{Choose}(s));$

$C(d_.) := \{e \mid e \ \epsilon_D \ \wedge \ e \sim_D d\};$

R
$).$

This functor is non-algorithmic in many respects. First, for defining whether or not an object r is in the residue domain R, we need the existential quantifier: r is in the carrier of R iff there exists a d in the given domain D such that r is the residue class of d. Second, for the definition of the residue class $C(d)$ of a given d we need the set quantifier. And, finally, for defining the result of applying the operation o in R to the two residue classes r and s we need the "Choose" function that chooses one element from the argument set.

In many areas of mathematics, non-algorithmic functors are the best one can do for solving very general problems. For the above functor, one can at least prove that if $D(\sim)$ is a congruence the residue domain of D has "similar" properties as D and, in addition, may have some new desirable properties, namely that certain objects exist that solve some problem at hand.

In a concrete situation it may be a trivial, an easy, a non-trivial, a very difficult, or an impossible task to come up with an *algorithmic* functor that constructs a domain which is isomorphic to the domain constructed by the above *non-algorithmic* functor.

6.5 An Example of an Algorithmic Functor

In contrast to the previous functor, the following functor is algorithmic and, under certain conditions, constructs a domain which is isomorphic to a residue domain. For this functor we suppose that D is equipped with a unary operator σ (which normally will be a "canonical simplifier" for the congruence $D(\sim)$):

Simplified-Domain($D_.$) :=
Functor($\langle S, o, s, t\rangle$,

$s_. \ \epsilon_S :\Longleftrightarrow s\epsilon_D \ \wedge \ \sigma_D(s) = s;$
$s_. \ o_{.S} \ t_. := \sigma_D(s \ o_D \ t);$

S
).

This functor is algorithmic in the sense that, if all the operations of D are algorithmic, then also all the operations in S are algorithmic, i.e. the functor "preserves" computability. First, in order to determine whether a given s is in the carrier of S, we only have to check whether s is in D and whether $D(\sigma)$, the function associated with the operator σ in D, applied to s is identical to s. Hence, under the assumption that $D(\epsilon)$ and $D(\sigma)$ are algorithmic functions, this is an algorithmic process. Similarly, under the assumption that σ_D and o_D are algorithmic, also o_S is algorithmic.

The main theorem which one can prove about the functor "Simplified-Domain" is the fact that, if σ_D is a "canonical simplifier" w.r.t. $\sim_D$ then the Simplified-Domain(D) is isomorphic to Residue-Domain(D). In fact, the function C defined locally in the functor Residue-Domain is an isomorphism.

Of course, these two functors are pervasive in mathematics and the construction of algorithmic canonical simplifiers is one of the main methods for turning an inconstructive part of mathematics into a constructive one. Finding canonical simplifiers can be trivial, easy, non-trivial, difficult, or shown to be non-existent. For example, for the residue domain modulo 3, it is trivial to find a canonical simplifier. Just take

$$\sigma_Z \ z_. := Mod(z, 3).$$

If we now call

$$Z3 = \text{Simplified} - \text{Domain } (Z)$$

then, for example, entering

$$2\ o_{Z3}\ 2$$

evaluates to

$$1.$$

For the residue domain of a multivariate polynomial ring modulo a polynomial ideal, as another example, the construction of a canonical simplifier is non-trivial (and it had been conjectured to be algorithmically unsolvable for quite some years) and needs the theory of Groebner bases. If P is some polynomial domain (over field coefficients or other suitable coefficient domains) and F is a (finite) set of polynomials in P,

$$\sigma_P\ p_. := \text{Normal} - \text{Form}\ (p, \text{Groebner} - \text{Basis}\ (F))$$

is a canonical simplifier for the residue domain of P modulo the ideal generated by F and

$$Q = \text{Simplified} - \text{Domain}\ (P)$$

constructs an algorithmic isomorphic realization of this residue domain.

Note that the above sequences of statements are not only a mathematical description of the construction but are *executable* code in our system based on Mathematica. This is so mainly because of the fact that (the innermost kernel of) Mathematica is nothing else than an implementation of the equational part of higher order logic without extensionality and because of the following two features of Mathematica:

- Mathematica allows Currying. Thus, it is perfectly possible to handle terms like "$D(o)(s,t)$".
- The "Module" construct of Mathematica has exactly the semantics of the above "Functor" construct and, in fact, one can teach Mathematica the Functor construct by just defining "Functor = Module".

Of course, the functor construct is available also in a few other programming languages, notably in ML. However, in the above realization, Functor is much more flexible because it allows one to define (decision predicates for) arbitrary carrier sets. Also, the above realization of the functor concept is smoothly embedded into a full-fledged computer algebra system with its practical computational power. Unfortunately, neither the fact that Mathematica is essentially higher order equational logic nor the fact that Currying is allowed and Module can be used as Functor, is explicitly mentioned or observed anywhere in the manual (Wolfram 89) nor in the rich literature on Mathematica. One may also add that, surprisingly, the above implementation of the functor principle in Mathematica is quite efficient: Generating code for the operations in a domain by functors of the above kind is hardly slower than hand-coding the operations for a particular domain. The only practical disadvantage of the above realization of functors is that the tracing and debugging facilities of Mathematica, of course, are not tailored for this particular way of structuring code for towers of domains. Anyway, with some experience, it is satisfactorily possible to use the existing debugging tools of Mathematica for debugging mathematical software written in the functor style.

7 Overcoming the Proof Gap

At present, nearly no proving capabilities are available in general computer algebra systems. The "directed" equational proving capabilities of Mathematica are normally used only for computation. Sophisticated special provers, as for example Collins' prover for the theory of real closed fields, see (Collins 1975) and the collection of articles on his algorithm in Vol.5/1-2 of the Journal of Symbolic Computation, are stand-alone systems.

As a first step for combining algebra and logic on the system level, one should of course make these special provers available from within general computer algebra systems like Mathematica. This is not really a problem any more because we can use the MathLink facility for accessing independent systems. Thus, if we encounter a formula like

$$\forall x \in \mathbf{R}\ \exists y \in \mathbf{R}\ (y^2 = x)$$

we should be able to call Collins' algorithm from within Mathematica and obtain the answer

False.

Similarly, if we encounter

$$x \in \mathbf{R}\ \wedge\ \exists y \in \mathbf{R}\ (y^2 = x)$$

a call to Collins' algorithm would yield the equivalent quantifier-free formula

$$x \in \mathbf{R} \wedge x \geq 0.$$

Note that Collins' algorithm goes far beyond the present "simplification capabilities" in ordinary computer algebra systems and, hence, the available of this and other sophisticated special provers will greatly expand the potential of these systems.

Similarly, at present, universal theorem provers are not smoothly linked to computer algebra systems. Of course, it is very much desirable to have access to universal theorem provers, notably of the natural deduction type, from within computer algebra systems. For example if, in the development of some piece of mathematics, say in a "computerized" text book, we want to prove that

$$\sim \text{ is an equivalence on } S \Longrightarrow S/\sim \text{ is a partition on } S$$

then, given the present state of proving technology, it should be possible to produce the proof of this proposition automatically and to print the proof out in a way that closely resembles a proof generated by a human. In fact, the proof of such theorems hardly needs any special trick nor intuition and can basically be done by "expanding" the definition using the natural deduction rules for quantifiers and propositional connectives plus some "simplification", i.e. manipulation of quantifier-free terms and atomic formulae. Thus, given a proposition of the above type, we would wish to be able to call a natural deduction prover from within a computer algebra system and would expect that it produces just the answer

True

or, with the option "verbose", the answer

```
Let ~ be an equivalence on S.

By the definition of S\~ we have to show that
...
```

It seems that one major obstacle that still makes this desire unrealistic is the fact that

- mathematical knowledge must be well organized so that, in a given proof situation, only relevant knowledge and all relevant knowledge is available and
- in a given area of mathematics, in addition to the universal natural deduction proof techniques, relevant special proof techniques should be available.

One of the main points I would like to make in this paper is that I think that the functor principle is not only crucial for structuring mathematical knowledge and mathematical methods (including algorithms) but it is also the key for structuring proof techniques. Namely, the first part of any functor in the above realization is a definition of the elements in the carrier by describing its characteristic function (which is algorithmic in the case of algorithmic functors). The structure of the description of this characteristic function naturally suggests a special prover for properties of the objects in the domain generated by the functor. For example, if the characteristic function is described inductively, a corresponding inductive proof technique is naturally connected with the functor. If the characteristic function is defined using set braces then the usual set-theoretic proof techniques will naturally apply, etc.

We give an example which is trivial (sorry for that!) but, nevertheless, is surprisingly powerful: We take the parameterless functor that defines one particular realization of the natural numbers and define addition in the resulting domain:

Natural-Numbers() :=
Functor($\langle N, n, m\rangle$,

$0\ \epsilon_N :\Longleftrightarrow$ True;
$s(n_.)\ \epsilon_N :\Longleftrightarrow n\ \epsilon_N$;
$n_.\epsilon_N :\Longleftrightarrow$ False;

$n_. +_N 0 := n$;
$n_. +_N s(m_.) := s(n +_N m)$;

N
).

Here, "0" and "s" are "arbitrary but fixed constants". The third clause in the inductive definition of the "ϵ_N" predicate is a clause that handles, the default case of any objects not having the format "$s(s(...s(0)))$". This is possible since Mathematica treats the clauses in equational inductive definitions successively.

Now we describe, in Mathematica, a simple inductive prover that can handle formulae of the form

$$\forall x_1 \forall x_2 \ldots \forall x_n \; l(x_1, \ldots, x_n) = r(x_1, \ldots, x_n)$$

where "l" and "r" are terms containing the free variables "x_1", ..., "x_n". The structure of the prover naturally reflects the induction scheme by which the above characteristic function for the natural numbers in the $0/s$-representation is defined. In fact, at a higher level of the system, provers of this kind could be automatically generated from the inductive definition of the respective characteristic function. The prover proceeds by, first, trying to prove the formula by equational simplification. If this fails, "x_1" is taken as the induction variable. The base case for this induction is generated and, again, a proof by simplification is attempted. If it fails, recursively, an induction over "x_2" (with "x_1" replaced by 0) is started. If it succeeds, the induction hypothesis and the formula to be proved in the induction step for (arbitrary but fixed) "x_1" are generated and a proof by simplification is attempted. If it fails, recursively, an induction over "x_2" (with "x_1" arbitrary but fixed) is started. If it succeeds we are done.

In Mathematica (with Version 2.2 syntax), this recursive induction prover can be described in a few lines:

```
ProofByInduction[

  (* of the *) equality_,
  (* w.r.t. the induction variables *) { v1_, v2___},
  (* using the *) equalities_] :=


  ProofByInduction0[ equality, { v1, v2}, { }, equalities]


ProofByInduction0[ equality_, { v1_, v2___},

    (* arbitrary but fixed *) { w1___},

    equalities_] :=
Module[

   { ... (* local variables *)},


    equalitySimplified = equality //. equalities;


    If[ LeftHandSide[ equalitySimplified] ===

          RightHandSide[ equalitySimplified],
       Return[ True]
       ];
```

```
equality0 = equality /. v1 -> 0;

equality0Simplified = equality0 //. equalities;
inductionHypothesis =

     Generalized[ LeftHandSide[ equality], { v1, w1}] ->

         RightHandSide[ equality];
equalitiesForInductionStep =

     Append[ equalities, inductionHypothesis];
equalitySv1 = equality /. v1 -> s[ v1];
equalitySv1Simplified =

     equalitySv1 //. equalitiesForInductionStep;


inductionBasisProved = False;
inductionStepProved = False;


If[  LeftHandSide[ equality0Simplified] ===

         RightHandSide[ equality0Simplified],
     inductionBasisProved = True
     ];
If[  LeftHandSide[ equalitySv1Simplified] ===

          RightHandSide[ equalitySv1Simplified],
     inductionStepProved = True
     ];


If[  ! inductionBasisProved,
     inductionBasisProved =
         ProofByInduction0[

            equality0Simplified,

            { v2}, { v1, w1}, equalities
            ]
          ];


If[  ! inductionStepProved,
      inductionStepProved =
```

```
                ProofByInduction0[

                    equalitySv1Simplified,

                    { v2}, { v1, w1},

                    equalitiesForInductionStep,

                    ]
                ];


          inductionBasisProved && inductionStepProved
          ]


    ProofByInduction0[

       equality_, {}, { w1___},  equalities_, options___
       ] := False
```

In this description of the recursive induction prover we left out all the lines for printing the proof. The prover is so simple that it hardly needs an explanation. "Generalized" is a procedure that replaces a universally quantified variable by an "arbitrary but fixed constant". "/." is the Mathematica operator for applying, once, rewrite rules (i.e. directed equalities) to terms. "//." is repeated application of "/." until the term does not change any more. Correspondingly, as a technical detail, the equalities that constitute the inductive definitions of operations like, for example, "+" must be presented in the form of "rules". For example, if we want to apply the prover to the domain generated by the above functor, we must extract the defining equalities from the functor so that they are available, say, as the value of some constant "NN", which (in Version 2.2 syntax) could be done by an instruction of the kind

```
NN =

  { sum[ x_, 0]] :> 0,

    sum[ x_, s[ y_]] :> s[ sum[ x, y]]}
```

Now, if we enter

```
ProofByInduction[
   sum[ x, sum[ y, z]] == sum[ sum[ x, y], z],
   { x, y, z},
   NN
   ]
```

we obtain the following inductive proof of the associativity of addition over the natural numbers:

```
Simplification proof of:
for all( x, y, z, ( ( x + ( y + z)) = ( ( x + y) + z))).
-----------------------------------


 . .Simplification of left-hand side:
 . .( x0 + ( y0 + z0))


 . .Simplification of right-hand side:
 . .( ( x0 + y0) + z0)
Not proved  by simplification.


Induction proof of:

for all( x, y, z, ( ( x + ( y + z)) = ( ( x + y) + z))).
Induction variable: x.
******************************


 .Prove induction basis (i. e. formula with x -> 0):
 .for all( y, z, ( ( 0 + ( y + z)) = ( ( 0 + y) + z))).
 .Simplification proof of:
 .for all( y, z, ( ( 0 + ( y + z)) = ( ( 0 + y) + z))).
 .-----------------------------------


 . . .Simplification of left-hand side:
 . . .( 0 + ( y0 + z0))


 . . .Simplification of right-hand side:
 . . .( ( 0 + y0) + z0)
 .Not proved  by simplification.


 .Induction proof of:

 .for all( y, z, ( ( 0 + ( y + z)) = ( ( 0 + y) + z))).
 .Induction variable: y.
 .******************************


 . .Prove induction basis (i. e. formula with y -> 0):
 . .for all( z, ( ( 0 + ( 0 + z)) = ( ( 0 + 0) + z))).
 . .Simplification proof of:
```

```
. .for all( z, ( ( 0 + ( 0 + z)) = ( ( 0 + 0) + z))).
. .-----------------------------------


. . . .Simplification of left-hand side:
. . . .( 0 + ( 0 + z0))


. . . .Simplification of right-hand side:
. . . .( ( 0 + 0) + z0)
. . . . = ( 0 + z0)
. .Not proved  by simplification.


. .Induction proof of:

. .for all( z, ( ( 0 + ( 0 + z)) = ( ( 0 + 0) + z))).
. .Induction variable: z.
. .******************************


. . .Prove induction basis (i. e. formula with z -> 0):
. . .for all( ( ( 0 + ( 0 + 0)) = ( ( 0 + 0) + 0))).


. . . .Simplification of left-hand side:
. . . .( 0 + ( 0 + 0))
. . . . = ( 0 + 0)
. . . . = 0


. . . .Simplification of right-hand side:
. . . .( ( 0 + 0) + 0)
. . . . = ( 0 + 0)
. . . . = 0
. . .Proved induction basis with z -> 0.


. . .Let z0 be arbitrary but fixed.
. . .Induction hypothesis (i. e. formula with z -> z0):
. . .for all( ( ( 0 + ( 0 + z0)) = ( ( 0 + 0) + z0))).


. . .Prove induction step formula (i. e. formula with z ->
```

```
 z0'):
. . .for all( ( ( 0 + ( 0 + z0')) = ( ( 0 + 0) + z0'))).


. . . .Simplification of left-hand side:
. . . .( 0 + ( 0 + z0'))
. . . . = ( 0 + ( 0 + z0)')
. . . . = ( 0 + ( 0 + z0))'
. . . . = ( ( 0 + 0) + z0)'
. . . . = ( 0 + z0)'


. . . .Simplification of right-hand side:
. . . .( ( 0 + 0) + z0')
. . . . = ( ( 0 + 0) + z0)'
. . . . = ( 0 + z0)'
. . .Proved induction step formula with z -> z0'.
. .Proved by induction over z.
. .Proved induction basis with y -> 0.


. .Let y0 be arbitrary but fixed.
. .Induction hypothesis (i. e. formula with y -> y0):
. .for all( z, ( ( 0 + ( y0 + z)) = ( ( 0 + y0) + z))).


. .Prove induction step formula (i. e. formula with y -> y0'


 ):
. .for all( z, ( ( 0 + ( y0' + z)) = ( ( 0 + y0') + z))).
. .Simplification proof of:
. .for all( z, ( ( 0 + ( y0' + z)) = ( ( 0 + y0') + z))).
. .-----------------------------------


. . . .Simplification of left-hand side:
. . . .( 0 + ( y0' + z0))


. . . .Simplification of right-hand side:
. . . .( ( 0 + y0') + z0)
. . . . = ( ( 0 + y0)' + z0)
. .Not proved  by simplification.


. .Induction proof of:
```

```
 . .for all( z, ( ( 0 + ( y0' + z)) = ( ( 0 + y0') + z))).
 . .Induction variable: z.
 . .****************************


 . . .Prove induction basis (i. e. formula with z -> 0):
 . . .for all( ( ( 0 + ( y0' + 0)) = ( ( 0 + y0') + 0))).


 . . . .Simplification of left-hand side:
..........
```

In many more lines, the proof works its way completely automatically through the recursion over the variables. Finally, it will return through all levels so that the last few lines look like this:

```
 . . . .Simplification of left-hand side:
 . . . .( x0' + ( y0' + z0'))
 . . . . = ( x0' + ( y0' + z0)')
 . . . . = ( x0' + ( y0' + z0))'
 . . . . = ( ( x0' + y0') + z0)'
 . . . . = ( ( x0' + y0)' + z0)'
 . . . .Simplification of right-hand side:
 . . . .( ( x0' + y0') + z0')
 . . . . = ( ( x0' + y0') + z0)'
 . . . . = ( ( x0' + y0)' + z0)'
 . . .Proved induction step formula with z -> z0'.
 . .Proved by induction over z.
 . .Proved induction step formula with y -> y0'.
 .Proved by induction over y.
 .Proved induction step formula with x -> x0'.
Proved by induction over x.
```

Of course, the crucial simplifications steps in the proof are exactly the ones that would be produced by the well-known test case method, see for example (Kapur *et al.* 1991). However, it is interesting that this prover evolves from a completely natural approach that does nothing more than reflecting the inductive definition of the particular representation of the natural numbers, it also allows to produce a natural language easy verbose presentation of the proof and it gets along without any human interaction. Most importantly, this prover is fully integrated and in fact programmed in the language of a full-fledged computer algebra system and can hence be used in intimate interaction with computation. It also should be mentioned that, of course, the sequence of the universally quantified variables may drastically influence the length of the proof. This is a phenomenon that has often been reported in the literature. In fact, in the above example, considering the variables in the order "z", "y", "x" produces a very short proof that succeeds in the simplest possible way. This is of course supported

by the heuristics that the variable which is the induction variable in the inductive definition of the functions involved should also be treated first in an inductive proof. In more complicated examples, the proof may be stuck at certain points producing an equality which is "not yet known". Such equalities are often the appropriate guess for a lemma that should be proved (by the above prover) before the main proof can be attempted once more. In fact, this side step can be called automatically so that quite impressive proofs can be handled completely automatically. More details can be found in the preliminary report (Buchberger 1995).

8 Conclusion

In this paper we argued that the combination of the potential of computer algebra and logic systems is the next natural step for enhancing the problem solving power of "symbolic computation" systems. For closing the gap between the "ideal" future system and the systems available at present we suggested to start from a computer algebra system and to add logic and, in particular, proving power. For quite a few reasons which we discussed in some detail, Mathematica seems to be particularly appealing as a starting point for such an approach. We tried to illustrate that the functor view seems to be crucial for structuring mathematical knowledge, mathematical methods (including algorithms) and mathematical proofs in future symbolic computation systems. Thus, the implementation of a well designed system of basic and advanced functors (encompassing both the computation and the prover details for each functor) is the essence of building up a future system. Currently, at the RISC institute, we work on a project that elaborates the details of the approach described in this paper.

Acknowledgement: This paper was written in the frame of a research contract with Fujitsu Labs, Numazu, Japan.

References

(Andrews 1986) P.B. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, London.

(Buchberger 1985) B. Buchberger: Groebner Bases: An Algorithmic Method in Polynomial Ideal Theory. Chapter 6 in: *Multidimensional Systems Theory*, (N.K. Bose ed.). D. Reidel Publishing Company, Dordrecht.

(Buchberger 1995) B. Buchberger: Induction Proofs in Equational Logic: A Case Study Using Mathematica. *Internal Technical Report*, The RISC Institute, A4232 Schloss Hagenberg, Austria.

(Collins 1975) G.E. Collins: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. Proceedings of the Second GI Conference on Automata Theory and Formal Languages. *Lecture Notes in Computer Science*, **33**, pp. 515-532, Springer, Heidelberg.

(Constable 1995) R. Constable: *The Nuprl System.* Lecture Notes of the Summer School on *Logic of Computation*, Marktoberndorf 1995. Edited by Institut für Informatik, Technische Universität München.

(Huet 1995) G. Huet: *The CIC System.* Lecture Notes of the Summer School on *Logic of Computation*, Marktoberndorf 1995. Edited by Institut für Informatik, Technische

Universität München.
(Kapur *et al.* 1991) D. Kapur, P. Narendran, H. Zhang: Automating Inductionless Induction using Test Sets, *Journal of Symbolic Computation*, Vol. 11, No. 1&2, February 1991, pp. 83-111.
(Soiffer 1995) N. Soiffer: *Mathematical Typesetting in Mathematica.* Proceedings of the ISSAC 1995 Conference, pp. 140-149.
(Wolfram 1988) S. Wolfram: *Mathematica: A System for Doing Mathematics by Computers.* Addison-Wesley Publishing Company, Redwood.