MATHEMATICA AS A REWRITE LANGUAGE *

BRUNO BUCHBERGER
Research Institute for Symbolic Computation, University of Linz
Linz, A 4040, Austria
E-mail: buchberger@risc.uni-linz.ac.at

ABSTRACT

The kernel of the Mathematica language is a higher-order conditional rewrite language with sequence variables. This fact is little known. We derive some conclusions from this for the use of Mathematica as a research tool in the area of rewriting and related areas.

1. The Objective of this Paper

This paper contains a simple message which is almost trivial but little known:

> The innermost kernel of the Mathematica language is essentially nothing else than a higher-order, conditional rewrite language, efficiently and professionally implemented.

This message may be interesting and useful for two communities which, at present, are nearly disjoint but whose future interaction, in my view, is desirable:

- *The community of Mathematica developers and users*: Knowing that what they are basically doing is implementing and using conditional rewriting may motivate them to look at the results of the field of rewriting, which over the past twenty years has matured into a solid and rich science, see for example [2].
- *The community of researchers in the field of rewriting*: Knowing that (a part of) Mathematica is an efficient implementation of conditional rewriting with many practically useful extras in the front end may provide them with an easily accessible and powerful research tool.

2. What is Mathematica?

Mathematica is a "mathematical software system". In fact, it is one of the most advanced and comprehensive systems in this category. In addition, it is one of the few mathematical software systems that are professionally produced and marketed by a company. This is a feature that may have disadvantages for a research community

*Invited paper at "The Second Fuji International Workshop on Functional and Logic Programming", November 1-4, 1996, Shonan Village, Japan, Sponsored by Japan Society for Software Science and Technology. To appear in the Proceedings (T. Ida ed.).

because the code of the kernel of professional systems is normally not open for the user. On the other hand, it also provides some definite advantages as, for example, professional maintenance, high performance, professional software production tools, and - in the case of Mathematica - a fantastic front end.

The present version of Mathematica was designed and implemented about ten years ago by Stephen Wolfram with the essential idea of basing computing on "pattern matching". A new version, 3.0., is just to be released, see [3]. Mathematica 3.0 has a number of important new features some of which are also relevant for the practical part of this paper. The innermost part of the language, however, has been retained and proved to be quite stable over the years. I think that this fact can be explained by the very reason that Wolfram's pattern matching is essentially the natural concept of conditional rewriting,

In a view that is motivated by the objectives of this paper, the structure of Mathematica can be described in the following layers:

- An evaluator for higher-order *conditional rewriting* of terms modulo equalities which forms the innermost part of the Mathematica language.
- An interpreter for (a big arsenal of) other language constructs in Mathematica including all common constructs of *procedural programming* languages.
- A huge *library* of numerical, algebraic, and symbolic "built-in" algorithms written in C and implementing the presently best mathematical methods in the various fields. This algorithms are called from within Mathematica programs by appropriate function calls.
- A *front-end* that supports wordprocessing, graphics, animation, sound, indexing, outlining, hyper-links, interactive buttons, mathematical typesetting, and also extensible syntax.

Mathematica is comprehensive both as a modern programming language and as a mathematical algorithms library and this is the reason for the majority of the Mathematica users (engineers, physicists, applied mathematicians, math researchers, math teachers) for working with this system. However, the distinguishing feature that may make Mathematica interesting as a research tool for people working in rewriting, unification, narrowing, constraint solving and related areas is its design based on (conditional) rewriting. In addition, the following features may be attractive for researchers in these areas:

- The availability of an arsenal of built-in algorithms for manipulating symbolic expressions and, in particular, rewrite rules, which makes it possible to write one's own evaluators for rewrite rule systems and other logical systems.
- The possibility for defining one's own notation.

3. The Rewrite Language Constructs of Mathematica

3.1. Overview

The innermost language constructs of Mathematica, which constitute a universal programming language based on the concept of conditional rewriting, are the following:

- constants,
- ordinary variables,
- sequence variables,
- expressions,
- conditional equalities (rewrite rules).

Let us call the version of Mathematica that is restricted to these language constructs "*Mathematica/R*" (i.e. "Mathematica restricted to Rewriting"). We give the syntax and (operational) semantics of these constructs informally, mostly by examples, in the subsequent subsections.

The only one of the above concepts that may be uncommon is the concept of "sequence variables". In principle, the usage of these variables could be avoided. However, they constitute an elegant and natural programming facility to which Mathematica owes some of its attractiveness. We think that they deserve to be used and studied in more depth by the rewrite research community.

3.2. Constants

We start from an infinite supply of constants. In the concrete syntax of Mathematica, identifiers (certain sequences of characters) as, for example, "$x23$", "Sin", etc. are used as constants. (In Mathematica, certain constants as, for example, the numerals "0", "1","2", $\cdots$, and many standard function names like "If","And", "Sin", etc. have built-in "meaning". However, this is of no relevance in the context of this paper, i.e. we can consider a "pure" version of Mathematica in which only constants are used whose meaning is defined by the user by explicit conditional rewrite rules of the form described below.)

3.3. Ordinary Variables

Identifiers (with the exception of certain special ones like the numerals) can also be used as variables. For indicating that an identifier is used as a variable, an underscore is written immediately after the identifier. For example, "$x\_$" and "object3_" are variables. Note that Mathematica does not "declare" the usage of an identifier as a variable for some global scope but declares this usage right at the place where the variable appears.

3.4. Sequence Variables

Identifiers can also be used as "sequence variables". For indicating that an identifier is used as a seqeunce variable, three underscores are written immediately after the identifier. For example, "$x\_\_\_$" and "$object3\_\_\_$" are sequence variables.

We prefer to declare sequence variables by decorating identifiers with an overbar and one underscore because this separates more clearly the declaration of using an identifier for sequences (declared by the overbar) and as a variable (declared by the underscore). In our notation, the above sequence variables look like this: "$\overline{x}\_$" and "$\overline{object3}\_$".

In fact, in the new version 3.0 of Mathematica, there is a wide range of possibilities for introducing one's own notation. Thus, if the reader is not happy with the above notation, he may choose a different way of distinguishing the usage of identifiers as constants, variables, and sequence variables, respectively.

Anticipating the description of the evaluation process, sequence variables can be replaced by any finite sequence (including the empty sequence) of expressions whereas ordinary variables can by replaced by only one expression.

3.5. Expressions

Any constant and any ordinary variable is a Mathematica expression.

If F is an expression that is not a variable and T_1, $\cdots$, T_n are expressions or sequence variables then

$$F[T_1, \cdots, T_n]$$

is also an expression. (Such an expression is called a "compound expression" or "application expression".)

For example, "23", "$Sin[Plus[2, x\_]]$" and "$Map[f, List[a, b, \overline{objects}\_]]$" are compound expressions.

Note that any expression F can be combined with any number n of expressions T_1, $\cdots$, T_n for forming a compound expression. Thus, for example, all of the following are syntactically correct expressions: "$List[]$","$List[3]$", "$List[a, b]$", "$List[2, 1, 4, 2, 1]$". (In fact, these expressions are used for representing lists of arbitrary length.)

Also "Currying" is possible. For example, "$P[D][Plus][x, y]$" is also a syntactically correct compound expression.

3.6. Special Notation

In the new version 3.0 of Mathematica, there are many ways of defining and using nice notation for expressions, e.g. special mathematical symbols, Greek and other special alphabets, infix operators, subscripts, superscripts, symbol decorations, various types of braces etc.

For example, we may decide to use "$S \cap T$" and "x_i" instead of "$Intersection[S, T]$" and "$Subscript[x, i]$", respectively. If we use such notation then, of course, the use of parentheses may become necessary.

Special notation is quite relevant for enhancing readability. Note, however, that expressions containing special notation are considered just as abbreviations for the corresponding expressions in the above standard notation, which in Mathematica jargon is called the "full form" notation. Thus, the language is not really extended by special notation. Rather, when an expression is entered and before it is processed by the evaluation procedure to be described below, it is first translated into full form.

3.7. Conditional Equalities (Rewrite Rules)

Rewrite rules and conditional rewrite rules (called "function definitions" in Mathematica) have the structure

$$lhsExpression \;:=\; rhsExpression$$

and

$$lhsExpression \;:=\; rhsExpression \;/;\; condition,$$

respectively. Here, $lhsExpression$ is an arbitrary expression (but not a variable) and $rhsExpression$ and $condition$ are arbitrary expressions. For saving writing effort, the underscores and overlines in the notation of ordinary and sequence variables can be (in fact, must be) omitted in $rhsExpression$ and $condition$.

Mathematica programs are just finite sequences of such (conditional) rewrite rules separated by semicolons. Here is a first example of a Mathematica program consisting of only two (unconditional) rewrite rules:

$$\begin{aligned} sum[m\_, 0] := &\quad m; \\ sum[m\_, s[n\_]] := &\quad s[sum[m, n]]. \end{aligned}$$

It is clear that what these two rewrite rules define is addition over the natural numbers represented by the expressions "0", "$s[0]$", "$s[s[0]]$",$\cdots$.

Here is an example of a Mathematica program containing a sequence variable:

$$\begin{aligned} map[f\_, List[]] :=&\quad List[]; \\ map[f\_, List[object\_, \overline{objects\_}]] :=&\quad prepended[f[object], \\ &\quad map[f, List[objects]]]. \end{aligned}$$

Note that the overbars and underscores are omitted on the right-hand side of the rewrite rules. Also, note that, semantically, "$f\_$" is a function variable here.

The following example demonstrates Currying:

$$\begin{aligned} \mathcal{N}[plus][x\_, y\_] &:= plus1[x, y]; \\ \mathcal{T}[plus][x\_, y\_] &:= plus2[x, y]. \end{aligned}$$

The following example uses conditions:

$$\begin{aligned} substituted[v\_, v\_, term\_] &:= term \quad /;\ isSymbol[v]; \\ substituted[w\_, v\_, term\_] &:= w \quad /;\ isSymbol[w] \wedge isSymbol[v]. \end{aligned}$$

Here are notational variants of all the definitions above. Note that, after adding appropriate notational definitions, these variants are executable code in Mathematica 3.0:

$$\begin{aligned} m\_ + 0 :=&\quad m; \\ m\_ + n\_^{+} :=&\quad (m+n)^{+}. \end{aligned}$$

$$\begin{aligned} f\_\Diamond\langle\rangle :=&\quad \langle\rangle; \\ f\_\Diamond\langle object\_, \overline{objects\_}\rangle :=&\quad f[object] \cdot (f\Diamond\langle objects\rangle). \end{aligned}$$

$$\begin{aligned} x\_ +_{\mathcal{N}} y\_ :=&\quad plus1[x, y]; \\ x\_ +_{\mathcal{T}} y\_ :=&\quad plus2[x, y]. \end{aligned}$$

$$\begin{aligned} v_{\_v\_\to term\_} :=&\quad term \quad /; v \in \mathcal{V}; \\ w_{\_v\_\to term\_} :=&\quad w \quad /; w \in \mathcal{V} \wedge v \in \mathcal{V}. \end{aligned}$$

3.8. The Evaluator of Mathematica

After "entering" a finite sequence of conditional rewrite rules R and an expression e, the runtime evaluator of Mathematica starts to rewrite e with respect to the rewrite rules in R, in the traditional sense of rewriting, until no more rewriting is possible. The particular strategy of rewriting is described informally on pp. 991 of [3]. An important peculiarity of this strategy is that rules in R are taken in the order in which they are given in R. (However, automatic re-ordering may take place in such a way that "more special" rules are arranged in front of "more general" rules. This is most times quite reasonable. Sometimes, however, the user might want to have more individual control over the arrangement of rules.) Knowledge about the specific order in which rewrite rules are arranged in R may safe some programming effort because, for example,

$$\begin{array}{rr} f[x\_] := & g1[x] \ /; \ c1[x]; \\ f[x\_] := & g2[x] \ /; \ \mathit{not}\ c1[x] \wedge d2[x] \end{array}$$

has the same effect as

$$\begin{array}{rl} f[x\_] := & g1[x] \ /; \ c1[x]; \\ f[x\_] := & g2[x] \ /; \ d2[x]. \end{array}$$

Also, it should be mentioned that Mathematica does provide a couple of possibilities for the user to influence the basic evaluation mechanism in order to improve efficiency and the structuredness of programs. For example, some of the arguments in function definitions can be specified to by "held", i.e. not evaluated immediately at call-time. This facility can be used for defining quantifiers with bound variables, see below. As another example, the user may decide "at which identifier" a particular rule is stored. For example, one may decide whether a rule of the form

$$f[g[x\_]] := h[x]$$

is stored as a "rule for f" or a "rule for g". We do not go into these technical details here. However, it should be clear that these mechanisms are of utmost importance for making (conditional) rewriting a practically attractive programming paradigm. For example, the mechanism for associating rules with identifiers opens an immediate possibility for realizing "object oriented programming" in Mathematica, see Section 2.4.13 of [3].

The only point that needs some explanation here is the use of the sequence variables in this evaluation process. For example, if R is the above pair of rewrite rules that defines "map", the first step in rewriting the expression

$$g\Diamond\langle 2,1,3,2\rangle$$

matches the ordinary variable "$object\_$" with the constant "2" and the sequence variable "$objects\_\_$" with the finite sequence "1","3","2" (and *not* with the tuple "$\langle 1,3,2\rangle$"!). Thus, after the first rewriting step the expression becomes

$$g[2]\cdot(g\Diamond\langle 1,3,2\rangle).$$

3.9. Lambda Expressions in Mathematica

Although, in practical programming, lambda expressions are rarely used it may be interesting to note that Mathematica provides them also. In Mathematica full form, "$Function[x, expression]$" is the notation for "$\lambda x.expression$". Again, the user may define another notation for lambda expressions according to his personal preference. The evaluation process for lambda expressions works correctly (with the necessary renaming of bound variables) also in the subtle cases in which free variables in expressions substituted for a lambda-quantified variable happen to get into the scope of another lambda-quantifier. For example,

$$Function[x, Function[y, x^2+y]][f[y]]$$

is correctly evaluated to

$$Function[y\$, f[y]^2+y\$]$$

with "y" renamed to the new variable "$y\$$".

Thus, summarizing, Mathematica implements - as part of its basic language interpreter - an evaluator for higher order conditional rewriting.

4. Manipulating Symbolic Objects and Rewrite Rules in Mathematica

Mathematica, however, provides rewriting not only as its basic language evaluator but also on a "second level". It makes the evaluator available to the user in the form of the built-in functions "*ReplaceAll*" and "*ReplaceRepeated*", respectively: "$ReplaceAll[e, R]$" (or, in the usual Mathematica syntax, "e /. R") yields the expression that results from applying one rewrite step to e using the rules in R and "$ReplaceRepeatedl[e, R]$" (or, in the usual Mathematica syntax, "e //. R") produces the normal form of e w.r.t. R. Also, since rewrite rules themselves are stored as compound expressions, one can use all the (built-in) functions for decomposing and re-composing rewrite rules and similar objects. Thus, we can, for example, easily write our own evaluator or trace functions that show the essential information on evaluation processes. (One particular trace function is also built in.)

For example, after entering

$$R := \langle m_{\_} + 0 :> m, m_{\_} + n_{\_}{}^{+} :> (m+n)^{+} \rangle$$

and

$$0^{+++} + 0^{++} \; //. \; R$$

one obtains

$$0^{+++++}.$$

(Note that, here, ":>" instead of ":=" must be used as the separator in rewrite rules.)

5. A Programming Example in Mathematica/R: Functors

Functors are an important means for building up towers of domains in a generic way. However, functors are available only in very few general programming languages as, for example, in ML but not in any of the mathematical software systems. In this section we illustrate how functors can be implemented easily (and efficiently) in Mathematica/R. For this, Currying is essential.

We adopt the common view that functors are functions that map domains into domains. Hence, for implementing functors, we must first agree on a representation of domains. We view a domain D to be a function that defines functions for certain "operators". For example, the following object $\mathcal{Z}$ is a simple domain:

$$\begin{aligned} \mathcal{Z}[\triangle] :=& \quad Plus, \\ \mathcal{Z}[1] :=& \quad 0, \end{aligned}$$

where "$Plus$" and "0" are the "built-in" addition and the built-in integer zero of Mathematica. We introduce the notation "o_D" for "$D[o]$" etc., which can also be implemented by the syntax extension facilities in Mathematica. Thus, we can write, for example,

$$2 \triangle_{\mathcal{Z}} 3$$

which, with the above definitions, evaluates to

$$5.$$

Also we will extend the meaning of "$\in$", with the common "abuse of notation", in such a way that "$x \in \mathcal{Z}$" yields "$True$" for exactly those objects x that we want to be in the "carrier" of $\mathcal{Z}$. For example, we could define

$$z\_ \in \mathcal{Z} := IntegerQ[z]$$

where "$IntegerQ$" is the built-in decision algorithm for integers. We may also wish to add a special notation for the "signature" of a domain but we don't do this here in order not to overload this introductory presentation.

With this representation of domains, a functor is now an object F that takes any domain D as an argument and produces $F[D]$ such that $F[D]$ can then be applied to any operation symbol o (in the signature of $F[D]$) yielding an operation in the domain $F[D]$. Also, "$x \in F[D]$" must be appropriately defined. Normally, one will think of any operation $F[D][o]$ to be applied reasonably only to objects in the carrier of $F[D]$, i.e. objects x for which "$x \in F[D]$" yields "$True$".

We can only give a trivial example here which should, however, be sufficient for illustrating the functor programming style that is possible in Mathematica: We define a functor F that, for any given domain D with any operations o, defines the Cartesian product. Using Currying, this functor could be defined as follows:

$$\begin{aligned} F[D] :=& \quad \textit{the } N \textit{ such that} \\ \\ \forall d1, d2 \ \langle d1, d2\rangle \in N :=& \quad d1 \in D \wedge d2 \in D; \\ \forall o, d1, d2, e1, e2 \ \langle d1, d2\rangle \, o_N \, \langle e1, e2\rangle :=& \quad \langle d1 \, o_D \, e1, d2 \, o_D \, e2\rangle. \end{aligned}$$

Now, the "such that" quantifier with exactly the above semantics is available in Mathematica in the form of the "Module" construct that binds N and also the other quantified variables that appear in the above definition so that the following definition can be seen as just an abbreviation of the above definition:

$$\begin{aligned} F[D\_] :=& \quad Module[\langle N, o, d1, d2, e1, e2\rangle, \\ \\ \langle d1\_, d2\_\rangle \in N :=& \qquad d1 \in D \wedge d2 \in D; \\ \langle d1\_, d2\_\rangle \, o\_N \, \langle e1\_, e2\_\rangle :=& \qquad \langle d1 \, o_D \, e1, d2 \, o_D \, e2\rangle; \\ \\ N] & \end{aligned}$$

Note that "$\_o$" is a variable in this definition. Thus, the definition applies to any operator o in the "signature" of $F[D]$ (which reasonably should be identical with the signature of D). The above definition is executable code in Mathematica 3.0 and, thus, the functor can now be applied, for example, to $\mathcal{Z}$:

$$\mathcal{C} := F[\mathcal{Z}].$$

After having evaluated this,

$$\langle 2, 3\rangle \triangle_{\mathcal{C}} \langle 5, 7\rangle$$

yields

$$\langle 7, 10\rangle.$$

6. A Programming Example in Mathematica/R: A Simple Inductive Prover

We now display the basic structure of a simple Mathematica/R program for automatically proving multivariate equalities over the natural numbers in the representation "0","0^{+}","0^{++}", $\cdots$, using a knowledge base of multivariate equalities in the form of rewrite rules (for example, the inductive definition of addition given above). This example shows the possibility of manipulating rewrite rules and related symbolic expressions from within Mathematica. The subroutine "$SimplificationProof$" is basically a call to the evaluator of Mathematica. "$RewriteRule$" turns a universally quantified equality into a Mathematica rewrite rule. "$ArbitraryButFixed$" generates new constants for universally quantified variables and "$Information$" is supposed to pretty-print information on a proof. With these explanations, the meaning of the program below should be evident.

$InductionProof[\forall_{\langle v1\_,\overline{v2\_}\rangle} lhs\_ \equiv rhs\_, \langle \overline{rules\_}\rangle] :=$

$Block[\langle\cdots\rangle,$

$simplificationProof = SimplificationProof[$
$\quad ArbitraryButFixed[\langle v1, v2\rangle, lhs \equiv rhs], \langle rules\rangle];$
$If[Successfull[simplificationProof],$
$\quad Return[Information[simplificationProof]]];$

$inductionBaseProof = InductionProof[\forall_{\langle v2\rangle} lhs \equiv rhs /. v1 \to 0, \langle rules\rangle];$
$If[notSuccessfull[inductionBaseProof],$
$\quad Return[Information[simplificationProof, inductionBaseProof]]];$

$inductionVariable = ArbitraryButFixed[v1];$

$$
\begin{aligned}
&inductionHypothesis = \\
&\qquad RewriteRule[\langle v2\rangle, lhs \equiv rhs/.v1 \rightarrow inductionVariable]; \\
&inductionStepProof = InductionProof[\\
&\qquad \forall_{\langle v2\rangle} lhs \equiv rhs/.v1 \rightarrow inductionVariable^{+}, \\
&\qquad Append[\langle rules\rangle, inductionHypothesis]]; \\
&Return[Information[\\
&\qquad simplificationProof, inductionBaseProof, inductionStepProof]]; \\
&]
\end{aligned}
$$

7. A Programming Example in Mathematica/R: Quantifiers

It is clear how expressions involving various quantifiers with bounded ranges can be conceived as abbreviations for certain terms involving the lambda quantifier. However, it is normally not possible to consider the definitions of these abbreviations as rewrite rules in the rewrite rule language itself. Mathematica provides the possibility to "hold" all or some of the arguments in a function definition, i.e. to prevent them from being evaluated at call-time. With this facility it is possible to define arbitrary quantifiers by rewrite rules *within* Mathematica/R. Together with the facilities for user-defined notation this gives a versatile and practically attractive potential for providing all the quantifiers one would want to see in a truely mathematical language. In particular, one can also define quantifiers that range over the carrier sets of arbitrary (algorithmically enumerable) domains. We illustrate quantifier introduction in the case of a simple variant of the universal quantifier over bounded integer ranges:

$$
\begin{aligned}
&Attributes[ForAll] = \langle HoldAll\rangle; \\
&\forall_{lowerBound\_\leq boundVariable\_\leq upperBound\_} expression\_ := \\
&\qquad If[lowerBound > upperBound, True, \\
&\qquad\qquad (\lambda\, boundVariable.\, expression)[lowerBound] \\
&\qquad\qquad \wedge \forall_{lowerBound+1 \leq boundVariable \leq upperBound} expression]
\end{aligned}
$$

"$Attributes[ForAll] = \langle HoldAll\rangle$" defines the arguments of "$ForAll$" (which is the full form name of "$\forall$") to be "*held*" unevaluated at call-time. Here is an example of an expression involving this quantifier:

$$\forall_{1\leq j\leq 10}\, 0 < j^2 < 101,$$

which evaluates to "$True$".

8. Conclusion: Research Topics and the Theorema Project

We hope that we were able to demonstrate in this paper that Mathematica is a theoretically and practically attractive implementation of the higher-order conditional rewriting programming paradigm. Addressing the two research communities mentioned in the introduction, we believe that the following two research lines should be pursued:

- The evaluation semantics and the denotational semantics of Mathematica should be defined and studied in more detail so that programming in Mathematica and verifying properties of Mathematica programs can be based on a more profound basis. This will be particularly important if Mathematica is extended to incorporate the unbounded quantifiers of pure mathematics. So far, only the evaluation semantics of Mathematica is defined and, in fact, it is only defined by the informal description given on pp. 991 of [3].
- Mathematica should and can profitably be used as a test environment for research in conditional rewriting and related subjects like narrowing, unification in specific theories, constraint solving, inductive proving etc.

These goals are also two of the goals in the author's Theorema project, see [1]. The overall goal of the Theorema project is the definition and implementation of a common language environment (starting from and based on Mathematica) for both the non-algorithmic and the algorithmic part of mathematics with the functor construct as the main structuring concept and special provers and solvers associated with each functor.

9. Acknowledgements and references

This paper was written while the author stayed as a visiting research fellow at the University of Tsukuba, Chair of Professor Tetsuo Ida, in the frame of the TARA (Tsukuba Advanced Research Alliance) Project. The work was also supported by the Advanced Information Technology Program (AITP) of the Japanese Information-Technology Promotion Agency (IPA) in the frame of the project "Coordination Programming in Open Computing Environments".

References

[1] B. Buchberger. Proving, Solving, Computing. In: *Proceedings of the Workshop on Multiparadigm Logic Programming*, GMD Bonn, September 5-6, 1996, (ed. T. Ida and Y. Guo), to appear.

[2] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In: *Handbook of Theoretical Computer Science*, Vol. B (ed.J. van Leeuwen), (North Holland,1990), pp. 243-320.

[3] S. Wolfram. *The Mathematica Book* (Wolfram Media, Champaign, Illinois, USA,1996), Preliminary edition for beta testers.