The *Theorema* Language: Implementing Object- and Meta-Level Usage of Symbols*

Bruno Buchberger Wolfgang Windsteiger

RISC-Linz
Schloß Hagenberg, Austria

Abstract

Interactive software systems that are designed to offer proving and computing facilities at the same time face the problem of evaluation of formulae: In the situation of *computing*, a formula given to the system should be evaluated whereas in the situation of *proving* the formula should be kept unevaluated. Also, in the *Theorema* project we use the same language (Mathematica 3.0) as the *working language* for defining new concepts, stating properties of these concepts, proving the properties, computing values using the new knowledge, etc. and as the *programming language* for implementing the system's provers, evaluators, etc. For this, a similar conflict of evaluating symbols in one situation and keeping them unevaluated in another situation has to be resolved.

In order to cope with these two problems, when an expression is input to the system, *Theorema* clearly distinguishes between the meta- and the object-level (formula level) of the language. On the formula level *no evaluations* whatsoever happen whereas on the meta-level both *Mathematica built-in rules* as well as *Theorema-defined rules* are used for evaluation of expressions. Moreover, a mechanism is provided that decides automatically whether a symbol is to be interpreted in its object-level- or in its meta-level context.

We will show how the capabilities of Mathematica 3.0 are used to implement this system behavior in such a fashion that new functions added to the system can easily be integrated into this concept.

Bibliographic Data

To be presented at the *Calculemus* Workshop, Eindhoven, July 13-15, 1998.

*) This reasearch work is supported by the Austrian Science Foundation (FWF) — project FO-1302 (SFB).

Introduction

Typically, a *Theorema* session proceeds as follows: After defining a new notion by something like

```
Definition["Greatest Common Divisor", any[m, n],
    gcd[m, n] = max_t (t | m ∧ t | n)]
```

one will want to formulate and prove a proposition like

```
Prove[
    Proposition["Euclid",
        gcd[m, n] = gcd[m − n, n]],
    KB → Definition["Greatest Common Divisor"]]
```

(in this command, the first argument is the proposition to be proved and the second is the "knowledge base" of assumptions that can be used in this proof.)

After having proved such a proposition, subsequent computation of concrete values of the function "gcd" may be much faster if, instead of the original definition, we use the proposition proved, i.e.

```
Compute[gcd[12, 18], KB → Proposition["Euclid"]]
```

results in a much faster computation than

```
Compute[gcd[12, 18], KB → Definition["Greatest Common Divisor"]]
```

(see [Buchberger, 1993]).

(Note that this can also be expressed by saying that *Theorema* considers a part of predicate logic as a universal programming language, which is "internal" to predicate logic, i.e. computation proceeds by applying a certain subset of the predicate logic inference rules - namely basically substitution and replacement - as the interpreter of the programming language. All function definitions are hence "internal" in the sense of [Barendregt, 1997] - referred to as "Poincaré Principle" (see [Poincaré, 1902])- and program verification reduces to proving the correctness of propositions like Proposition["Euclid"].)

In this paper, we do not address this general philosophy of embedding computation into logic. Rather, we deal with a very practical problem of the usage of symbols that arises from the fact that we use Mathematica as the programming language for implementing our provers, solvers, and evaluators. Summarizing, these are programs that take an expression *E* and a "knowledge base" *K* of expressions as an input and prove, solve, or evaluate *E* using the knowledge *K*. Now, some of the symbols occurring in *E* or *K* (for example, the symbols "∧", "<" etc.) may already have a "built-in" meaning in Mathematica. Thus, before *E* and *K* are actually passed to the provers, solvers, and evaluators of *Theorema*, they may already be transformed by the application of the built-in Mathematica rules for these symbols, which of course completely distorts the intended functionality. For example, **Prove[1<0⇒X>0, K]**, by the built-in rules of Mathematica, would be immediately transformed into **Prove[True,K]**. Any reasonable prover "Prove" would now yield the answer "proved" independent of the knowledge *K*. However, what we want is that the prover "Prove" uses the knowledge *K* (which, typically, will include knowledge on the concepts denoted by "0", "1", "<", ">") for deciding whether or not "1<0⇒X>0" is true and, in fact, for certain *K*, may become false. Hence, we must make sure that "1<0⇒X>0" (and the expressions in *K*) remain unevaluated as an input to "Prove".

Similarly, the Mathematica programs implementing the *Theorema* provers, solvers, and evaluators, typically will contain clauses that refer to the structure of expressions in terms of symbols used and again, as part of these clauses, these symbols must not be evaluated even if, in Mathematica, built-in rules for some of the symbols are available. This is tricky because, on the other hand, it may well be desirable and necessary to use certain symbols, at certain places in these programs, with their built-in Mathematica meaning. For example, in the clause

```
Prove[E1_ ⟺ E2_, K_] := Prove[E1 ⇒ E2, K] ∧ Prove[E2 ⇒ E1, K]
```

which might be a reasonable part of a prover programmed in Mathematica, the symbol "⇒" must stay unevaluated whereas the symbol "∧" must have the built-in meaning of Mathematica (i.e. the usual "and" operator as a programming construct).

A straightforward solution would be to assign different names to symbols in formulae for proving, solving, and computing, and to use different symbols in formulae and in programming. Different notations for different symbols must be invented then of course. This, however, would contradict to the final vision of the *Theorema* system to use *one language* for proving, solving, and computing (evaluating) on the one hand, and programming on the other hand (see [Buchberger, 1996]). Hence, a mechanism for detecting the context, in which a formula is given, must be developed and, depending on the context, evaluation must be prevented or not.

For evaluating formulae, knowledge built into Mathematica as well as knowledge defined as part of the *Theorema* system is used. The extent to which *Theorema* knowledge is applied during evaluation depends on which parts of the system are loaded into the running session. For instance, the formula "3∈ℕ" can not be evaluated further by Mathematica because neither "∈" nor "ℕ" have a pre-defined meaning. In this stage, natural numbers can be built up from the scratch and the introduced knowledge can then be used for evaluating "3∈ℕ". However, knowledge on the symbols "∈" and "ℕ" is already available in the *Theorema*-package on natural numbers and, after loading the natural number package, the above formula will immediately evaluate to "True". By this mechanism, each user can choose the level, from where to start defining new theories.

In the sequel we will describe in detail a solution using Mathematica (in fact, whenever we mention Mathematica, we mean Mathematica version 3.0 or higher). Features like the possibility to modify the default input behavior by defining *preprocessing functions* and to create individual components (with separate name spaces) by means of *packages* are heavily used in this solution.

The Problem

The logical "implies-operator" is available in Mathematica in one of the forms "⇒" (infix) or "**Implies**" (prefix).

```
{A ⇒ B, Implies[A, B]}
```

```
{Implies[A, B], Implies[A, B]}
```

```
?⇒
```

```
An infix operator , x ⇒ y is by default interpreted as Implies [x, y].
```

```
?Implies
```

```
Implies [p, q] represents the logical implication p -> q.
```

As one can see from the information about "Implies", Mathematica is aware of the meaning of an implication in mathematical logics and it immediately tries to evaluate to "True" or "False" if possible, i.e.

```
A ⇒ B
```

```
Implies[A, B]
```

```
False ⇒ B
```

```
True
```

```
1 < 0 ⇒ X > 0
```

```
True
```

Fine, this is what we expect when we want to calculate the formula's value but, by the Mathematica evaluation mechanism, this happens even when the formula is passed as an argument to a function, like e.g. a prover. (Note, however, that evaluation of function arguments *before* passing them to the function is in most of the cases exactly what one wants (composition of functions!), but not in this case!)

```
Prove[1 < 0 ⇒ X > 0]
```

```
Prove[True]
```

No matter how the function "Prove" is actually implemented, it will get the formula "True" to be proven with respect to a "default knowledge base" that contains the knowledge collected during this session. In this case — presupposing the knowledge base does not introduce "special interpretations" for the symbols occuring in the formula — the result is still correct, but we definitely *do not want to do "proving by evaluating Mathematica built-in rules"*. Now, suppose we want to explicitely specify the knowledge base used for proving, we get

```
Prove[1 < 0 ⇒ X > 0, KB → {n < m ⟺ (∃_{z∈ℤ} (n + z == m))}]

Prove[True, KB → {n < m ⟺ ∃_{z∈ℤ} n + z == m}]
```

Again, the goal formula is immediately evaluated, disregarding the interpretation of the symbol "<" as specified in the knowledge base.

We can prevent argument evaluation by giving the function "Prove" an attribute like

```
SetAttributes[Prove, HoldFirst]

Prove[1 < 0 ⇒ X > 0, KB → {n < m ⟺ (∃_{z∈ℤ} (n + z == m))}]

Prove[Implies[1 < 0, X > 0], KB → {n < m ⟺ ∃_{z∈ℤ} n + z == m}]
```

which has exactly the desired effect, but it would make the implementation of "Prove" and its subroutines pretty cumbersome, because in each program step and in each subroutine call it must be guaranteed that the formula stays unevaluated.

```
Prove[E1_ ⟺ E2_, K_] := Module[{lr = E1 ⇒ E2, rl = E2 ⇒ E1},
                        Print[StringForm["We prove: `` and ``", lr, rl]];
      ProveImplication[lr, K] ⋀ ProveImplication[rl, K]]

Prove[(1 < 0) ⟺ (X > 0), K]

We prove : True and Implies[X > 0, False]

ProveImplication[True, K] && ProveImplication[Implies[X > 0, False], K]
```

Hence, many of the involved subroutines will also have to carry the attribute "HoldFirst" (or "HoldAll" or "HoldRest" depending on the circumstances), which introduces new inconveniences in combination with pattern matching:

```
SetAttributes[ProveImplication, HoldFirst];
ProveImplication[Implies[E1_, E2_], K] := Prove[E2, K ⋃ {E1}]

Prove[(1 < 0) ⟺ (X > 0), K]

We prove : True and Implies[X > 0, False]

ProveImplication[lr$2, K] && ProveImplication[rl$2, K]

%[[2, 1]]

Implies[X > 0, False]
```

Although the head of the argument is "Implies" the pattern in the definition of the function does not match due to the argument "HoldFirst", so programming using pattern matching needs special care in combination with "Hold"-attributes. In the example the troubles stem from the fact that "Implies" has builtin Mathematica evaluation rules. However, the same effects can be observed for new symbols that are introduced in *Theorema* as soon as they get evaluation rules attached, i.e. when the package containing the knowledge about the symbol is loaded.

Summarizing, it would be possible to solve the problem by consequent use of "Hold"-like attributes in all function definitions. As a consequence, considerable parts of the implementation would loose the elegance that was gained by making use of the pattern matching mechanism of Mathematica and programming would be much more effort.

For the case of computing, an additional aspect must be regarded: we want to compute results using different knowledge bases. On the one hand in order to compare different algorithms, on the other hand to demonstrate the influence of "more knowledge" on the efficiency of computation. For instance, using the (non-algorithmic) definition of "divisibility of natural numbers" in our knowledge base the computation

$$\mathbf{Compute}\Big[\mathbf{divides[3,7]},\ \mathbf{KB}\rightarrow\Big\{\mathbf{divides[n,m]}\Longleftrightarrow\underset{q\in\mathbb{N}}{\exists}(\mathbf{m}==\mathbf{n}*\mathbf{q})\Big\}\Big]$$

will *not terminate*. Of course, using "better" knowledge (in the sense of "algorithmic" knowledge) about divisibility will *immediately result* in

$$\mathbf{Compute}\Big[\mathbf{divides[3,7]},\ \mathbf{KB}\rightarrow\Big\{\mathbf{divides[n,m]}\Longleftrightarrow\underset{\substack{q\in\mathbb{N}\\ q<m}}{\exists}(\mathbf{m}==\mathbf{n}*\mathbf{q})\Big\}\Big]$$

```
False
```

We can now investigate how "more knowledge" (e.g. using continued subtraction, using division, using knowledge about prime numbers, etc.) influences the complexity of the computation. Again, whatever is known about "divides" in the system must not be used to evaluate the arguments of "Compute".

The Solution Used in Theorema

We tried to find a more satisfying solution than the heavy use of "Hold"-Attributes described in the previous section. Such a solution must fulfill at least the following criteria:

• On the user level it must be detected which symbols must be evaluated and which have to stay unevaluated (e.g. as an argument for a prover).
• On the programming level the implementation of the provers, solvers, and evaluators should be as easy as possible at the same time preserving as much of Mathematica's elegant programming style as possible.

In the following sections we will describe a solution, which meets these criteria. It is based on the following key ideas:

• Inside a formula, certain symbols (namely those having a meaning in either Mathematica or *Theorema*) will be renamed into *fresh symbols* without a meaning.
• A mechanism is provided to automatically select the *appropriate context* of a symbol (proving - computing - programming).

The Use of the Mathematica Package Concept to Organize Language Levels

Before we explain the implementation issues of this solution, we will describe the underlying system design from a software-technological point of view.

Mathematica provides the concept of *packages* in order to introduce different name spaces for symbols. On the one hand, this minimizes the danger of naming conflicts between distinct parts of the system, on the other hand it gives total control on the capabilities available in the system at any moment through the possibility of loading individual packages separately. Moreover, having the whole system split up into many small blocks increases the system's efficiency, because a user can, at any moment, have only those packages loaded, which are really necessary.

Packages implementing parts of the *Theorema* language represent *Theorema language levels*, which are hierarchically structured according to inherent dependencies between language components. The most basic level is the *core* of the language, higher levels are e.g. logical connectives, numbers, sets, tuples, or quantifiers, the top level will be the *Theorema* user language, in which all features of the language are available. Consequently, for each component its syntax and its semantics are *different language levels* and therefore separate packages are available containing the syntax and the semantics, respectively. By this, it is possible to load, for instance, the language level "syntax of logical connectives" only, which would have the effect that formulae containing any logical connectives ("$\wedge$", "$\vee$", "$\Rightarrow$", "$\Leftrightarrow$", etc.) are understood correctly by the system. However, no evaluations will happen at that level, since no semantics of the connectives is known. This alone would solve part of the evaluation problems, if, when it comes to proving formulae, only the syntax level needed for input of the formulae is loaded. Still, as soon as proving and computing must be available at the same time, some more ideas are necessary, but, emphasizing again the software-technologigcal aspect, splitting up in small pieces is valuable on its own.

How Theorema Input is Processed

In this section we describe how *Theorema* input is processed. Input to the system is given through the notebook interface of Mathematica 3.0, so basically everything that is allowed as Mathematica input is potential *Theorema* input. In particular, we will make use of available special symbols (like "$\Rightarrow$", "$\Leftrightarrow$", "$\forall$", "$\exists$", etc.) and of the possibility of structured twodimensional input (like "$\frac{a}{b}$", "$\overline{x}$", "$\underset{x=1,\ldots,n}{\forall} P[x]$", etc.). Still, the *Theorema language* is defined in terms of Mathematica expressions, i.e. expressions of the form **h[a$_1$, ..., a$_n$]** consisting of a head h with an arbitrary number of arguments a_1, ..., a_n. In this sense the *Theorema* language is embedded into the set of Mathematica expressions. We will call those expressions that make up the *Theorema* language from now on *Theorema expressions*. For most of the *Theorema* expressions, *nice notation variants for in- and output* will be defined in the respective syntax levels using the facilities of Mathematica (we will not go into details on notation here), for evaluation purposes some of the expressions will in addition have a *meaning* defined in the respective semantics levels.

We will describe in the following sections how the interplay between syntax and semantics is organized in order to provide the desired functionality.

Separating Syntax- and Semantics-Processing

As described above there is a clear separation between the implementation of the *syntax* and the *semantics* of the language.

$$\text{Nice input} \underset{\textbf{syntax}}{\overset{\text{by means of}}{\Longrightarrow}} \textit{Theorema}\text{ expression} \underset{\textbf{semantics}}{\overset{\text{by means of}}{\Longrightarrow}} \text{Value}$$

As an example, the formula $\underset{x \in \mathbb{N}}{\forall} x > 0$ is translated by means of syntax into the *Theorema* expression "ForAll[•range[•var[x], $\mathbb{N}$], •var[x] > 0]" (note that this example is simplified). By means of semantics, this *Theorema* expression can be evaluated to "True".

The translation from user input to *Theorema* expressions, i.e. the *syntax part*, must happen *before* Mathematica starts evaluation. Otherwise, a function having an attribute like HoldAll would prevent its arguments to be translated into *Theorema* expressions, which in turn would lead to unpredictable behavior. Mathematica provides several possibilities for placing actions before usual evaluation. The input string/boxes are first fed through the function stored in the global variable **$PreRead** followed by **MakeExpression** (a Mathematica system function, which translates the box-form representing the input structure into Mathematica expressions) and finally through the function stored in the global variable **$Pre**. The result of $Pre is then evaluated using built-in as well as user-defined rules known for the symbols occuring in the expression, so all the syntax translations need to be packed into those 3 functions. In a standard Mathematica session, $PreRead and $Pre are undefined, whereas many rules for MakeExpression are available by default. However, it is allowed to extend (or even overwrite) the standard definition of MakeExpression.

$$\begin{array}{c}
\text{Nice input} \\
\Downarrow \textbf{\$PreRead} \\
\text{Rearranged input} \\
\Downarrow \textbf{MakeExpression} \\
\textit{Theorema}\text{ Pre–Expression} \\
\Downarrow \textbf{\$Pre} \\
\textit{Theorema}\text{ Expression}
\end{array}$$

The input given by the user is first split into boxes using built-in operator precedence rules. For some input forms that we want to allow the resulting box representation is unsatisfactory, i.e. it would need complicated MakeExpression definitions to process the input correctly. Instead it is much easier to rearrange the box structure *before* sending the boxes to MakeExpression. This is done in **$PreRead**.

In fact, most of the work is done by **MakeExpression**, however, some translations (namely those with a "global nature", see the example below) would need very sophisticated MakeExpression rules, which would be fairly complicated to implement and to maintain. In those cases it is preferable to produce an (intermediate) *Theorema pre-expression* using MakeExpression and shift the final translation into **$Pre**. In all other cases the *Theorema* pre–expression and the *Theorema* expression are identical and $Pre leaves the expression unchanged. We will call the translation from a pre-expression into a *Theorema* expression *Theorema preprocessing.*

Example: The *Theorema* expression representing a tuple $\langle a, b, c\rangle$ is the expression **List[a,b,c]**. A very simple MakeExpression rule can translate every anglebracket expression into the corresponding List expression.

On the other hand, a variable x in *Theorema* is represented by **•var[x]** at every occurrence in a formula but there is no special input syntax for variables since it can be detected from the formula context which symbols are variables. This means that in a quantified formula like $\underset{x}{\forall} P[x]$ every occurence of the symbol x in $P[x]$ has to be replaced by •var[x]. However, a single MakeExpression rule can hardly achieve this. Instead, we leave the x unchanged for the moment and wrap the entire expression into a pre–expression tagged **•withVariables** that indicates x as a variable. In the preprocessing step then every x inside the •withVariables-expression is substituted by •var[x].

Input	***Theorema* Pre–Expression**	***Theorema* Expression**
⟨a, b, c⟩	List[a, b, c]	List[a, b, c]
$g[x] \Leftrightarrow \underset{x}{\forall} P[x]$	Iff[g[x], •withVariables[x, ForAll[x, P[x]]]]	Iff[g[x], ForAll[•var[x], P[•var[x]]]]

Theorema expressions are processed further using rules introduced in the semantics of the language. In this example, the semantics would typically contain a Mathematica definition for equivalence, e.g. **Iff[A_,B_]:=Implies[A,B]∧Implies[B,A]**. Now, as soon as the semantics is loaded, this definition would then be applied to the *Theorema* expression in order to compute the formula's value. As already explained in the previous sections, this is not always desired.

Introducing Fresh Names for Functions

The crucial point in our approach is that we prevent automatic evaluation of formulae by a mechanism to introduce *fresh names* for functions occuring in *Theorema* expressions that would otherwise be evaluated. By convention, the fresh name is always produced by prepending the special symbol "™" (like *Theorema* ☺!) to the original name, so e.g. the Mathematica builtin **And** is translated into **™And**, sometimes we use this opportunity to assign more telling names, so e.g. the operator **LeftRightArrow** representing an equivalence goes into **™Iff**. Based on translation tables these translations can be applied at appropriate places.

Fresh names must occur in the *Theorema* expression already, thus, the translation into fresh names must happen as a part of the syntax processing. A first approach to achieve this would be to define new MakeExpression rules in such a way that the respective fresh names - different from the names used in the implementation of the semantics - are introduced immediately. Still, in order to avoid evaluation conflicts with Mathematica built-in functions, redefinition of most of the default MakeExpression rules for Mathematica functions (∧, +, *, etc.) would be necessary, and, even worse, the built-in functions would not be available anymore, except in their FullForm. Therefore, expressions containing evaluatable symbols are viewed as *Theorema* pre-expressions that are replaced by the respective fresh symbols in the preprocessing step.

Distinguishing Proving, Computing, and Programming

In programming in particular, we want to use some symbols both as parts of formulae and with their Mathematica built-in meaning. Recall the example of the implementation of a simple proof rule

Prove[E1_ ⟺ E2_, K_] := Prove[E1 ⇒ E2, K] ⋀ Prove[E2 ⇒ E1, K]

In this line of input, the symbols "⇔" and "⇒" are used on the object-level of formulae and, therefore, should stay unevaluated, whereas the symbol "∧" is used on the meta-level as a programming construct (note that, at the same time, "∧" can also occur on the object-level as logical connective in a formula). Since, in *Theorema* expressions, *tags* are used to indicate the "type" of an expression, these tags can be used as a means to distinguish between the object- and the meta-level usage of symbols. For instance, a formula in *Theorema* is always represented as the *Theorema* expression **•lf[label,formula]** (a "labeled formula" containing a string used as label and the actual formula), which tells the preprocessor to substitute fresh names inside such an expression. At any time, the preprocessor can "learn" additional keywords indicating object-level in a *Theorema* expression. In addition to this, we provide one special tag ("•") to explicitely switch to object-level inside the expression.

In the *Theorema user language*, which is the language used on the top-level by a *Theorema* user, we provide *environments* for entering definitions, theorems, algorithms, etc. by which we again can distinguish between object-level and meta-level of the language automatically. In order to do computations in the *Theorema proving (standard) session*, the top level function **Compute** has to be called, which evaluates an expression w.r.t. a given knowledge base. If no knowledge base is specified, the evaluator uses a default knowledge base that contains knowledge available from the semantics of the currently loaded language level. In case one is mainly interested in computing, it is then probably more comfortable to switch to a *Theorema compuational session*, in which every input given to the system is evaluated automatically just as if it was wrapped in a call to "Compute". The default knowledge base can be extended by knowledge available in the standard session.

From Theory to Practice

In this section, we want to demonstrate the features of the top-level *Theorema* user language for giving definitions, stating and proving theorems, computing values, and switch between different *Theorema* sessions. After having loaded *Theorema*, we first enter the language level "syntax of logical connectives".

```
Needs["Theorema`Language`Syntax`Connectives"];
```

At this level, we can now enter a labeled formula like

```
•lf["1", 1 < 0 ⇔ X > 0]
```

```
•lf[1, 1 < 0 ⇔ X > 0]
```

No evaluations happen due to two reasons: fresh names were substituted for the symbols, in particular for the Mathematica built-in "<" and ">", and, secondly, the "⇔" does not yet have a meaning if the syntax is loaded only. In the semantics, we typically have definitions in the following fashion:

```
A_ ⇔ B_ := (A ⇒ B) ⋀ (B ⇒ A)
```

Still, unless "Compute" is used explicitely, no evaluations happen inside a formula. We now switch to the user language level, in order to show the possibilities available there.

```
Needs["Theorema`Language`Semantics`UserLanguage"];
```

```
•lf["1", 1 < 0 ⇔ X > 0]
```

```
•lf[1, 1 < 0 ⇔ X > 0]
```

```
Compute[•lf["1", 1 < 0 ⇔ X > 0]]
```

```
X > 0 ⇒ False
```

Let us now consider a quantified formula

$$\bullet\mathrm{lf}\left[\text{"2"},\ g[x] \Leftrightarrow \underset{x}{\forall} P[x]\right] \mathbin{//} \mathrm{InputForm}$$

```
•lf["2", ™Iff[g[x],
  ™ForAll[•range[•simpleRange[
    •var[x]]], True, P[•var[x]]]]]
```

Note that, on the left-hand side of the formula the *x* is regarded as a constant, whereas inside the quantifier the *x* is marked as a variable. We will now use *environments* to build up (a very small part of) mathematics. Furthermore, from now on, we do not show variables in their internal representation anymore, we rather italicize them.

```
FormatVariables[OutputForm → "Standard", FontSlant → "Italic"];
Needs["Theorema`Language`Syntax`Number"];
Needs["Theorema`Language`Semantics`Quantifier"];
```

Definition["divides", any[n, m],

$$\mathrm{div}[n, m] \Leftrightarrow \left(\underset{q\in\mathbb{N}}{\exists}(m = n\, q)\right) \text{ "non-algorithmic"}$$
]

The **any[n,m]** is used instead of a universal quantifier in the formula, which is particualrly useful, if an environment contains more than one formula with the same variables universally quantified. Such an environment has the effect that from now on we can refer to the formula(e) given in the environment by

```
Definition["divides"]
```

•def[divides, •range[•simpleRange[n], •simpleRange[m]],
True, •flist[•lf[non-algorithmic, div[n, m] ⇔ $\underset{q\in\mathbb{N}}{\exists}(m = n\, q)$]]]

```
Compute[div[3, 6], KB → Definition["divides"]]
```

```
divides_non -algorithmic
```

$\underset{q\in\mathbb{N}}{\exists}(6 == 3\, q)$

By application of clause "non-algorithmic" of definition "divides" the expression "div[3,6]" was evaluated to "$\underset{q\in\mathbb{N}}{\exists}(6 == 3\, q)$", which was not evaluated further, because, in the current implementation, a quantifier ranging over an infinite set is not evaluated, due to the risk of an infinite loop. Actually, the quantifier does not range over an infinite set!

Proposition["divides", any[n, m],

$$\mathrm{div}[n, m] \Leftrightarrow \left(\underset{q=1,\ldots,m}{\exists}(m = n\, q)\right) \text{ "algorithmic"}$$
]

This proposition, of course, has to be proven before it can be used for computation.

```
Prove[Proposition["divides"],
        KB → Definition["divides"], Prover → NatNumPredLogProver]
```

NatNumPredLogProver[•prop[divides, •range[•simpleRange[n], •simpleRange[m]],
True, •flist[•lf[algorithmic, div[n, m] ⇔ $\underset{q=1,\ldots,m}{\exists}(m = n\, q)$]]],
•def[divides, •range[•simpleRange[n], •simpleRange[m]], True,
•flist[•lf[non-algorithmic, div[n, m] ⇔ $\underset{q\in\mathbb{N}}{\exists}(m = n\, q)$]]]]

The proving method specified in the option "Prover→method" is used to prove the goal w.r.t. the knowledge base given in the option "KB→knowledge base" (in this example, we specified a nonexistent method just to show what happens!). Suppose the prover had managed to prove the proposition, we can then use the "new knowledge" for computing.

```
Compute[div[3, 6], KB → Proposition["divides"]]

divides_algorithmic

True

Compute[div[4, 999], KB → Proposition["divides"]]

divides_algorithmic

False
```

Now, if we intend to do some computations with the knowledge collected during the standard session, we enter a computational session, initialize the default knowledge base (so that it is aware of the semantics of the currently loaded language level), and then update the knowledge base with the proposition proved in the standard session.

```
ComputationalSession["Div"]; InitializeDefaultKB[]; UpdateKB[Proposition["divides"]]

•[ShowKB[]]
```

$$\left\{\text{div}[\text{n\_?IsNotSequenceAtom}, \text{m\_?IsNotSequenceAtom}] :\to \left(\text{PrintTrace}[\text{divides}, \text{algorithmic}];\ \underset{q=1,\ldots,m}{\exists}(m = n\,q)\right)\right\}$$

```
div[4, 17]

divides_algorithmic

False

div[7, 49]

divides_algorithmic

True

EndComputationalSession[]
```

We can also compare the influence of a "better algorithm" on the computing time needed. Consider

Proposition["divides–improved", any[n, m],

$$\begin{array}{ll} \text{div}[1, m] = \text{True} & \text{"1|m"} \\ \text{div}[n, m] \Leftrightarrow \left(\underset{q=1,\ldots,\left\lfloor \frac{m}{2} \right\rfloor}{\exists}(m = n\,q)\right) & \text{"n|m"} \end{array}$$

]

```
Timing[Compute[div[13, 6000], KB → Proposition["divides–improved"]]]

divides–improved_n|m

{1.71 Second, False}
```

```
Timing[Compute[div[13, 6000], KB → Proposition["divides"]]]

divides_algorithmic

{2.76 Second, False}

Timing[Compute[div[1, 600], KB → Proposition["divides–improved"]]]

divides –improved_ 1|m

{0.03 Second, True}

Timing[Compute[div[1, 600], KB → Proposition["divides"]]]

divides_algorithmic

{0.28 Second, True}
```

Prospects for Future Development

Based on the feature of having different *sessions* and facility to *switch between sessions* and to *transfer knowledge from one session to another*, we will now increase the power and the comfort in the individual sessions. In the standard as well as in the computational session, we will introduce *typed variables*, in the computational session it will be possible also to give function definitions in a Mathematica style using ":=", etc.

By this, the *Theorema* system should become a valuable tool for "doing mathematics" in many fields of algorithmic mathematics.

Bibliography

[Barendregt, 1997] H. Barendregt, Erik Barendsen: *Autarkic Computations in Formal Proofs*. Technical report, Univ. of Nijmegen, 1997.

[Buchberger, 1993] B. Buchberger: *Mathematica: A System for Doing Mathematics by Computer?* In: Advances in the Design of Symbolic Computation Systems. A. Miola (ed.), Springer New York-Vienna, 1997. Written version of an invited talk at DISCO 93, Gmunden, Austria.

[Buchberger, 1996] B. Buchberger: *Proving, Solving, Computing. A Language Environment Based on Mathematica*. Invited talk at the "Multi-Paradigm Logic Programming" Workshop, Bonn, Sept. 5-6, 1996.

[Poincaré, 1902] H. Poincaré: *La Science et l'Hypothèse*. Flammarion, Paris.