A ‘Calculemus–approach’ to high-school math?

Walther A. Neuper

HTBLA, Höhere Technische Bundeslehranstalt
A-5020 Salzburg, Inzlinger Hauptstr. 30

Abstract A ‘Calculemus-approach’ in constructing software for high-school math is considered, i.e. the question how to encompass logical rigor as well as calculational power by combining concepts and components of theorem provers and of algebra systems for educational use.

The requirements for student-users are formulated following recent work done at IST, TU Graz, in close contact to RISC Hagenberg, and based on a prototype originating from this work.

The paper presents work in progress towards these requirements w.r.t. four concepts: the logical base, stepwise problem solving, reactive user-guidance, and automated generation of explanations.

1 Introduction

‘Both Deduction Systems and Computer Algebra Systems are receiving growing attention from industry and academia. ... Mathematical Software Systems have been commercially very successful. Their use is now wide-spread in industry, education, and scientific contexts: there are now literally millions of installations of computer algebra programs.’

The author teaches math in ‘notebook-classes’ at an Austrian high-school, where each student has his or her own notebook, and uses it (i.e. an algebra system, mainly) in math, too. The experience with the algebra systems is doubtful; they are not designed for education 2: ‘...there is still need for improvement as many application domains still fall outside the scope of existing Deduction Systems and Computer Algebra Systems.’

In this paper we will discuss a mathematics-engine for tutoring, which necessarily falls outside the scope of existing systems, if the following requirements should be met:

1 http://www.cs.unitn.it/~rseba/calculemus2001/meeting.html#generalities
2 There is one important and successful exception, mathXpert [Bee84]; and because there is no space for related work, the the main difference is the following: mathXpert does not provide explicit specification and thus doesn’t handle subproblems, ... etc. as proposed in sect.3.2.
1. solve problems of 'applied math' automatically from a given (prepared by an author, and hidden) 'formalization' and specification; if nothing is given, support interactive formalization and specification
2. present calculations close to paper and pencil work, and additionally show their structure, and relations to underlying knowledge
3. allow one to trace a calculation down to elementary steps, i.e. the application of theorems to a proof-state
4. allow the user to do all the steps himself or herself, give feedback, and return to a demonstration-mode on request
5. generate explanations automatically on request by the student, by a kind of 'reflection' (and not by an author trying to foresee all possible requests)

The feasibility of the requirements w.r.t. interactivity has been shown by a prototype \(^3\), developed at the Institute of Software Technology, TU Graz in close contact with the Research Institute of Symbolic Computation, Linz \(^5\). The interactivity relies on a math-engine; the latter is the concern of this paper. Two concepts of the math-engine seem to be novel ones, the hierarchy of problem-types (see 3.2) and the 'reverse rewriting' (see 3.3). In order to motivate the overall design, we want to present all essential concepts, and we want to mention parts still open within this work in progress; thus, in order to respect space limits, some of the presentations are sketchy.

2 Logical foundations

If we want to give feedback to a student whether a step in a calculation is admissible or not, we have to maintain some kind of proof-state, and thus are concerned with the logical framework. As a consequence, the implementation of the prototype is based on a theorem prover, and not on an algebra system. The choice between Reduce/Redlog [DS96] and Isabelle [Pau94] was made in favor of the latter. The main reason was, Isabelle has already implemented most of the knowledge for high-school math in high-order logic (HOL), and the respective theories are being developed rapidly.

\(^3\) http://www.ist.tu-graz.ac.at/research/edm/isac/
\(^4\) http://www.ist.tu-graz.ac.at/
\(^5\) http://www.risc.uni-linz.ac.at/
2.1 Foundations in constructive mathematics

Isabelle aims at proving theorems, whereas traditional high-school math is ‘applied math’, mainly concerned with ‘example construction problems’ [Buc94], where some objects \( x \) are given, meeting some precondition \( \eta(x) \), and some objects \( y \) are to be constructed, meeting the postcondition \( \rho(x,y) \).

Combining Isabelle’s non-constructive concept with the constructive one necessary for high-school math is likely to be neither harmless nor trivial: The foundations for the educational system math.Ypert [Bee84] have been researched carefully - [Bee88] demonstrated, that carelessly combining the axiom of extensionality, the recursion theorem and the separation axiom leads to an inconsistent system. The author is not aware of research, indicating what problems may arise from the combination of HOL with set construction as described subsequently.

2.2 Implementation and manipulation of a proof-state

A proof-tree represents a (partially) completed proof. In order to have a simple mapping between the proof-tree and the external representation of a calculation, the safe LCF-grounds have been left, implementing the following (where \([\ ]\) denote lists):

**Definition 1.** The set \( \mathcal{P} \) of proof-trees is inductively defined on nodes \( N = (O,Bs) \) where \( O \) is called a **proof-object** and \( Bs \) is a list called the branches of \( O \):

\[
\begin{align*}
(O,[\ ]) & \in \mathcal{P} \\
(O,Bs) & \in \mathcal{P} \text{ where } Bs = [N_1,\ldots,N_n] \text{ with } N_i \in \mathcal{P}
\end{align*}
\]

There are two types of proof-objects:

- **problem-objects** are records containing all data concerning the specification of an example
- **solve-objects** are records representing the deductive steps, i.e. steps of logical deduction and application of algebraic laws.

*These two kinds of objects are called proof-objects; their respective fields will be introduced as soon as needed. The trees root is a problem-object, called the root-problem.*

The types of branches model the structures of high-school math, the most specific ones concern set construction, e.g. the calculation

\[
\{1,2,3,4,5,6,7\} \cap \{m \in \mathcal{N}. m/7\} = \{1,7\}
\]
where \( \divides \) denotes 'divides' in the natural numbers, is represented by a so-called \textit{Collect}-branch in a solve-object as follows

\[
\begin{align*}
O.\text{expr} & = \{1, 2, 3, 4, 5, 6, 7\} \cap \{m \in \mathbb{N} : m \divides 7\} \\
O.\text{branch} & = \text{Collect} \\
[O_1.\text{expr} = 1 \in \{m \in \mathbb{N} : m \divides 7\} & \land O_1.\text{result} = \text{true} \\
O_2.\text{expr} = 2 \in \{m \in \mathbb{N} : m \divides 7\} & \land O_2.\text{result} = \text{false} \\
& \ldots \\
O.\text{result} & = \{1, 7\}
\end{align*}
\]

where \( O.x \) selects the field \( x \) (i.e. \textit{expr}, \textit{branch}, \textit{result}) from the proof-object \( O \), and where the fields of one and the same \( O \) have the same indent level. The formal definition of this \textit{branch-type} and others is straightforward, and gives the basis of the systems semantics.

The purpose of splitting a simple calculation like that is to get steps, each of which is justified by an elementary theorem proven in Isabelle/HOL. In the example given these theorems could be

\[
\begin{align*}
def\text{divisors} & \quad \text{divisors } n = \{m \in \mathbb{N} : m \divides n\} \\
\text{divisor}\_\text{leq} & \quad m \divides n = m \leq n \land m \divides n \\
\text{inter}\_\text{def} & \quad \{x. \ P x\} \cap \{x. \ Q x\} = \{x. \ P x \land Q x\} \\
\text{mem}\_\text{Collect}\_\text{leq} & \quad (a \in \{x. \ P(x)\}) = P(a)
\end{align*}
\]

Then the calculation could be represented on a so-called \textbf{work-sheet}, the structure being connected with, and justified by the theorems,

\[
\begin{align*}
\text{divisors } 7 & = \\
& = \{m \in \mathbb{N} : m \divides 7\} = \\
& = \{m \in \mathbb{N} : m \leq 7, \land m \divides 7\} = \\
& = \{m \in \mathbb{N} : m \leq 7\} \cap \{m \in \mathbb{N} : m \divides 7\} = \\
& = \{1, 2, 3, 4, 5, 6, 7\} \cap \{m \in \mathbb{N} : m \divides 7\} = \\
& = \{m \in \mathbb{N} : m \divides 7\} = \\
& = 1/7 = \\
& = \text{true} = \\
& = 2 \in \{m \in \mathbb{N} : m \divides 7\} = \\
& = \ldots \\
& = \{1, 7\}
\end{align*}
\]

where the theorems are applied by the \textbf{tactic Rewrite}\textsuperscript{6}, shown flushed right. The labels on the right margin relate to those in the knowledge-base, while the tactics (5), (7), (9) correspond to the meta-logic, represented by the branch-type \textit{Collect}. These tactics are designed for input by the user and do not indicate the underlying theorems (which is a questionable design decision). The above example exceeds Isabelle's meta-logic.

\textsuperscript{6} \textit{Rewrite} applies a rewrite rule once, whereas \textit{Rewrite Set} applies a (terminating) set of rules
A survey on high-school math showed [Neu01a] that virtually all examples can be modeled by three special branch-types like Collect.

The proof-tree $P \in \mathcal{P}$ is extended by applying a tactic (e.g., Rewrite, Calculate, Check, Collect) to a given formula $f$ in $P$, representing a proof-state $(P, f)$. The tactics semantics is based on the transition relation from $(P, f)$ to $(P', f')$, defined for each tactic respectively. These definitions are straightforward, leading to the notions of applicable tactics.

3 Autonomous and interactive problem solving

Having indicated the essential prerequisites for the requirements (2.) and (3.) on p.2, we turn to the requirement (1.) ‘solve problems automatically, or support interactive formalization and specification’.

3.1 The phases of problem solving

are model, specify, solve, and have particular input-data and output-data:

\[ \text{description} \rightarrow \text{model} \rightarrow \text{formalization} \rightarrow \ldots \]
\[ \ldots \rightarrow \text{specify} \rightarrow \text{specification} \rightarrow \ldots \]
\[ \ldots \rightarrow \text{solve} \rightarrow \text{solution} \]

To illustrate we shall use a typical example in high-school math. This example, referred to as ‘maximum-example’ in the sequel, is given by the following description:

*Given a circle with radius $r = 7$, inscribe a rectangle with length $u$ and width $v$. Determine $u$ and $v$ such that the rectangles area $A$ is a maximum.*

The model phases output is a formalization, potentially in one of the variants

\[ F_1 \equiv ( \{ (r, 7) \}, \{ A, \{ u, v \} \}, \{ 0 \leq \frac{u}{r} \leq 1, \{ A = uv, (\frac{u}{r})^2 + (\frac{v}{r})^2 = r^2 \} \} ) \]
\[ F_{II} \equiv ( \{ (r, 7) \}, \{ A, \{ u, v \} \}, \{ 0 \leq \frac{v}{r} \leq 1, \{ A = uv, (\frac{u}{r})^2 + (\frac{v}{r})^2 = r^2 \} \} ) \]
\[ F_{III} \equiv ( \{ (r, 7) \}, \{ A, \{ u, v \} \}, \{ 0 \leq \alpha \leq \frac{\pi}{2}, \{ A = uv, \frac{u}{r} = \sin \alpha, \frac{v}{r} = r \cos \alpha \} \} ) \]

This may motivate the definition

**Definition 2.** Given a set $I$ of substitutions, called input-items, a set $O \neq \emptyset$ of output-variables, and a set $R$ of relations, the triple $F = (I, O, R)$ is a formalization iff $(\text{Vars} I) \cap O = \emptyset$. 
where $Vars$ extracts the first element of each pair in a substitution \(^7\)
(and in later use also: $Vars$ extracts the variables from a term). The sets
within the triple contain different types of objects, in particular sets for
the purpose of grouping; a formalization will ‘instantiate’ a problem-type,
see Def.5.

The formalization must be given (by an author, and eventually hidden
from the student) in order for the problem to be ‘solved automatically’.

\(\text{3.2 Interactive and automated specification of problems}\)

In algebra systems specification is done by selecting the function (say
\textit{solve} an equation), and supplying the appropriate arguments; the domain
is specified by some kind of switch (e.g. real or complex solutions for
equations).

In order to allow for interactive specification, we need the underlying
knowledge in an explicit form, which can be browsed by the user, and
which allows for interactive selection. The universe of this knowledge is
organized along three axes (see Fig.1 on p.15), one of which concerns
problem-types.

Let us begin with the maximum-example, structured as an (example
construction) problem, which may be the result of automated modeling
from the hidden formalization, or which may be the result of the user’s
input: \(^9\)

\[
\text{problem } \left[\text{"maximum"}\right] \\
I \equiv \{ (r,7) \} \\
\eta(r) \equiv 0 \leq r \\
O \equiv \{ A, \{ u, v \} \} \\
\rho(u, v, r) \equiv A = u \cdot v \wedge \left(\frac{u}{r}\right)^2 + \left(\frac{v}{r}\right)^2 = r^2 \wedge \\
\forall A', u' \cdot v'. A' = u' \cdot v' \wedge \left(\frac{u'}{r}\right)^2 + \left(\frac{v'}{r}\right)^2 = r^2 \implies A' \leq A \\
R \equiv \{ A = u \cdot v, \left(\frac{u}{r}\right)^2 + \left(\frac{v}{r}\right)^2 = r^2 \} 
\]

The post-condition $\rho(a, b, r)$ is, besides $I$, the characteristic of a prob-
lem (-type). Unfortunately, the above post-condition is very hard to ver-
ify: this could be done for simpler ones like the post-condition for the
solution-set $L$ of an equation $a = b$, which may be $\forall l \in L. \left(\lambda v. a\right)l = \frac{b}{2}$.

\(^7\) We exclude the important case of substitutions with a value \textit{arbitrary, but fixed}
from this paper.

\(^8\) This decision could have alternatives: one could push ahead the modeling to the
domain of elementary geometry (comprising formal notions of circle, rectangle, in-
scribe, etc., or one could employ techniques from artificial intelligence like some
geometry tutors.

\(^9\) Note that the intervals like $0 \leq \frac{u}{r} \leq r$ from the formalization do not show up in this
problem; but it is passed to a subproblem, see the script \textit{Maximum, value} on p.12.
\[(\lambda v. b) l | \leq \epsilon, \text{ but not for the maximum-example} - \text{we will come back to this issue in section 4. The definition for the notion of a problem is}

**Definition 3.** Given a substitution \(I\) and a set \(O\) of variables with \((\text{Vars } I) \cap O = \emptyset\), and a set \(R\) of predicates, some predicates \(\eta(\text{Vars } I)\) and \(\rho(\text{Vars } I, \text{Vars } O, \text{Vars } R)\), with \(\rho\) quantifying all free variables by the such-quantifier, then \(L = (I, \eta, O, \rho, R)\) is a **problem**.

The elements of \(I\) are called **input-items** and those of \(O\) are called **output-variables**. \(\eta\) is the **pre-condition** and the predicate \(\rho\) is the **post-condition**, relating input and output.

\(R\) consists of sub-terms of \(\rho\), it is redundant for pedagogical reasons.

A problem is **suitable** iff \(\eta(I)\) evaluates to true, and a problem is **solved** iff there exists a set \(V\) of values for all output-variables, \(\mathcal{O} = O \times V\) such that \(\rho(I, \mathcal{O})\) evaluates to true. The set \(V\) is called the **solution** of \(L\).

\(\eta(I)\) denotes substitution of the \(\text{Vars } I\) by their respective values. There are restrictions on the (substitution and evaluation) of \(\rho(I, \mathcal{O})\), the post-condition - see 4.3.

Now, what we need for automated specification is the following: given a formalization, find the appropriate problem-**type**. For instance, the dozens of problems, found along with the maximum-example in textbooks on calculus, shall belong to one single problem-type.

A preliminary attempt to describe the problem-type, the maximum-example may belong to, is the following:

\[
\begin{align*}
\text{problemtype ["maximum"]} \\
I' &\equiv \{a_{\_} \} \\
\eta' &\equiv \text{map } (\lambda x. \leq ) \ a_{\_} \\
O' &\equiv \{ m_{\_}\ vs_{\_}\} \\
\rho' &\equiv \text{let } x_1 = \{ m_{\_}\} \cup \{ vs_{\_}\} \cup (\text{Vars } rs_{\_}); \\
&\quad \text{in } \text{map } (\text{op\_}) \ rs_{\_}\ &\& \$ \\
&\quad \forall \$ x_2 \$. (\lambda x_1\. \$. \text{map } (\text{op\_}) \ rs_{\_}\ &\& \$ ) \& \& \$ x_2 \& \& \$ \implies \$ \text{primed } m_{\_}\ &\& \$ \& \& \$ m_{\_}\ &\& \$ \\
R' &\equiv \{ rs_{\_}\}
\end{align*}
\]

where \$ denotes a term-constructor. \textit{primed} attaches a ‘ to a variable. \textit{map} takes sets instead of lists, and, for instance, creates inequalities for the pairs \textit{fix}. The underscores define identifiers to belong to the meta-language of problem-types (as opposed to the object-language of math).

The design of the language describing how to generate pre- and post-condition from the problem-type is an open question, and the author is not aware of related work. Presently the syntax of \(\eta'\) and \(\rho'\) (let us call them p-templates) is unclear, as well as the details of their generation, i.e. some function \(\mathcal{X}\).
\[ \mathcal{X} : \mathcal{P}' \times \mathcal{S} \rightarrow \mathcal{P} \]

where \( \mathcal{P}' \) is a set of p-templates, \( \mathcal{S} \) is a set of substitutions, and \( \mathcal{P} \) a set of predicates. This function works for the example as follows

\[ \mathcal{X} : (\text{map } (\{0 \leq r\} \ 	ext{fix}_-, \{(fix_-, \{(r, l)\})\}) \rightarrow 0 \leq r \]

The function \( \mathcal{X} \) is necessary to proceed from Def.4 to Def.5.

**Definition 4.** Let \( X_1, X_2 \) be sets of variables, \( P_1 \) a set of predicates. Given the sets \( I', O' \) and \( R' \) of variables, and given two p-templates \( \eta' \) and \( \rho' \), then \( Y = (I', \eta', O', \rho', R') \) is a **problem-type**. \( I', O', R' \) are called the input-components of \( Y \), shortly IOR'.

A problem-type is a kind of a general template to be instantiated by a particular formalization:

**Definition 5.** Let \( F_0, F \) be formalizations, \( F \equiv (I, O, R) \), \( Y \equiv (I', \eta', O', \rho', R') \) a problem-type with input-components IOR'. Let further be \( L \) a problem, \( P \) a set of predicates, \( S \) a set of sets of input-items, and \( V \) a set of sets of variables.

Then we say \( F \) *instantiates* \( Y \) given \( F_0 \) yielding \( L \) iff

(i) \( F_0 = \emptyset \ \land \ \text{matching IOR'} F \) while generating \( \sigma_Y = \text{match IOR'} F \)

(ii) \( \exists \eta \in P, I \in S, \eta = \mathcal{X}(\eta', \sigma_Y) \) and \( \eta(I) \) holds

(iii) \( \exists \rho \in P, \rho = \mathcal{X}(\rho', \sigma_Y) \)

(iv) \( \exists O \in V, L = (I, \eta, O, \rho, R) \)

where \text{match} yields a substitution \( \sigma_Y \), and \text{matching} is true for \( \sigma_Y = \emptyset \).

Condition (i) contains a case-distinction concerning whether there is a hidden formalization \( F_0 \) prepared or not. This is the answer to requirement (1.) on p.2 w.r.t. specification.

With the latter definition we have come to the point, because instantiates can be used to construct a quasi-order, which in turn allows one to construct an acyclic graph.

**Definition 6.** Given two problem-types \( Y_1 = (I_1', \eta_1', O_1', \rho_1', R_1') \) and \( Y_2 = (I_2', \eta_2', O_2', \rho_2', R_2') \), and a set \( \mathcal{F} \) of formalizations, we say \( Y_1 \) *refines* \( Y_2 \) iff \( \forall F \in \mathcal{F}. F \text{ instantiates } Y_2 \Rightarrow F \text{ instantiates } Y_1 \).

The acyclic graph constructed by the quasi-order on problem-types induced by refines, leads to the hierarchy of problem-types, called the problem-tree:
Definition 7. Let $\mathcal{ID}$ be a set of strings, and $id_i \in \mathcal{ID}$ some elements (for $0 \leq i \leq n, n \leq 2$), called labels, and $\mathcal{Y}$ a set of problem-types with some elements $Y_i \in \mathcal{Y}$. Then we call the acyclic graph "problemtree" with constructor "Join" and nodes in $(\mathcal{ID} \times \mathcal{Y})$

\[
\text{datatype problemtree} = \text{Join of } ((\mathcal{ID} \times \mathcal{Y}) \times \text{(problemtree list)})
\]

a problem-tree iff
(i) for all parallel nodes $(id_i, Y_i)$ the labels $id_i$ are pairwise disjoint
(ii) $Y_i$ below $Y_j$ iff $Y_i$ refines $Y_j$
(iii) $Y_i$ parallel $Y_j$ iff $\neg(Y_i$ refines $Y_j) \land \neg(Y_j$ refines $Y_i)$

where below and parallel are relations on the acyclic graph (equivalent to the respective notions on terms). The list of labels $[id_{i_1}, \cdots, id_{i_k}]$ along the path from the problem-trees root to a problem-type $Y_k$ is called a problem-ID.

Given a problem-tree, we can automatically refine a vaguely formulated problem to a stronger formulated one! Let us look at the part of a problem-tree concerning the maximum-example:

```
\text{:}
\text{Join (("make\_fun", }Y_k),
\text{ [ Join (("by\_elimination", }Y_k1), []),
\text{ Join (("by\_new\_variable", }Y_k2), []), \cdots ]})
\text{:}
```

Given any one of the formalizations of p.5 and the problem-ID ["make\_fun"], automated refinement can be done due to the matching of input-items and the evaluation of the pre-condition of $Y_{k1}$ and $Y_{k2}$ respectively: If the formalization is $F_I$ or $F_{II}$, $Y_{k1}$ would be chosen, and the method attached would yield $A(a) = 2a \sqrt{7^2 - \left(\frac{a}{2}\right)^2}$ or $A(b) = 2b \sqrt{7^2 - \left(\frac{b}{2}\right)^2}$, and if the formalization is $F_{III}$, $Y_{k2}$ would yield something like $A(a) = 2 \cdot 7 \sin \alpha \cdot 2 \cdot 7 \cos \alpha$. $Y_{k1}$ is addressed by the problem-ID ["make\_fun", "by\_elimination"] and $Y_{k2}$ is addressed by ["make\_fun", "by\_new\_variable"]). The script Maximum_value on p.12 employs the mechanism of refinement.

To the author's best knowledge, the automated refinement on explicit problem-types seems to be a novel mechanism. We eagerly look forward to experiences of how larger portions of knowledge, e.g. all types of equations in the high-school syllabus are to be mapped to a problem-tree, to the respective patterns of input-items and pre-conditions, and how the performance of automated refinement will be.
3.3 Solve stepwise by rewriting

A major part of high-school math can be done by rewriting, [Neu01a] gives a survey on the respective topics within the syllabus. Rewriting is very close to what a human mathematician does when applying a certain theorem to a certain formula. Mechanical rewriting, the step by step application of theorems (like \( l = r \), as directed rules \( l \rightarrow r \)) tends to be verbose; [Buc97] showed how to master this verbosity by a ‘nested cells representation’.

Re-engineering the simplifiers for \( \mathcal{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R} \) and the complex numbers \( \mathcal{C} \), all together with the related function constants, is apparently the most straight forward task. Several simplifiers are trivial, such as differentiation or expanding logarithms, and just require basic knowledge on termination and confluence in rewriting.

Many other tasks are not trivial; for instance (a comprehensive survey is given in [Neu01a]), the calculation of the canonical polynomial form over \( \mathbb{Z} \) with numerical constants involves conditional rewriting \(^{10}\), and thus the basic knowledge as presented e.g. in [BN98] is not sufficient. This is even more the case with rationals or radicals.

Also rewrite orders are of concern, for instance if a special term (say \( \sqrt{\cdot} \)) needs to be shifted to the root of a term (Knuth-Bendix order !) - see the tactic \texttt{Rewrite\_Set\_isolate\_root} in the script \texttt{square\_equation} on p.12.

There is a wealth of very specialized literature on rewriting, but much is 25 years old and thus not easily accessible. In fact, some specific knowledge, buried in any algebra system, may need to be reinvented. But all together, this task is more suited for industrious students of computer mathematics, than for challenging R&D.

Implementation of ‘reverse rewriting’ follows a novel idea, exploiting a suggestion of [Har97] on the combination of algebra systems and theorem provers. The idea is the following:

There are topics in high-school mathematics, not really suitable for rewriting, but taught as such. A typical example is factorization while calculating in \( \mathbb{Q}[x] \):

\[
\frac{x - 2}{2x^2 - 2} = \quad \text{Calculate } 8 = 2 \cdot 1^2 \quad (1)
\]

\[
\frac{x - 2}{2x^2 - 2} = \quad \text{Rewrite } 2a - 2b = 2(a - b) \quad (2)
\]

\(^{10}\) While distributing towards polynomial form by \((a + b) \cdot c \rightarrow a \cdot c + b \cdot c\), numeral constants must be contracted by \( a \text{ is Const} \land b \text{ is Const} \Rightarrow a \cdot c + b \cdot c \rightarrow (a + b) \cdot c\); both rules together would obstruct termination, if there would not be the condition.
\[
\frac{x - 2}{2(x^2 - 1^2)} = \quad \text{Rewrite} \quad a^2 - b^2 = (a + b)(a - b) \quad (3)
\]
\[
\frac{x - 2}{2(x + 1)(x - 1)} = \quad \text{Rewrite} \quad \frac{b}{ab} = \frac{1}{a} \quad (4)
\]

Indeed, factorizing and canceling in (2) \ldots (4) of the above example are taught as an application of the (inverse) law of distributivity and of laws on binoms, while (1) is skipped as ‘obvious’.

The automated process of ‘reverse rewriting’ is proposed to factorize yielding (4), but without showing the result to the student; rather, certain factors are multiplied again in order to present steps (1) \ldots (3) exactly in this sequence. This is an answer to the requirements (2.) and (3.) on p.2. The technique of ‘reverse rewriting’ is intended to implement step by step calculation of canonical forms in all domains taught at high-schools.

4 Scripts for reactive user-guidance

4.1 Describing methods

A method of solving a specified and instantiated problem is described by a so-called script. Scripts are formulated in Isabelle/HOL; their syntax, given in Backus normal-form, is

\[
\text{script} ::= \text{Script id arg}^* = \text{body} \\
\text{arg} ::= \text{id} | ( ( \text{id} :: \text{type} ) ) \\
\text{body} ::= \text{expr} \\
\text{expr} ::= \% \text{id} . \text{expr} \ |
\begin{array}{l}
\text{let id = expr ( ; id = expr)* in expr} \\
\text{if prop then expr else expr} \\
\text{while prop expr id} \\
\text{repeat expr id} \\
\text{try expr id} \\
\text{(expr or expr) id} \\
\text{tac ( id | listexpr )* | listexpr | id}
\end{array}
\]

\[
\text{type} ::= \text{id} \\
\text{tac} ::= \text{id}
\]

where (/) belongs to the object-language, id is an identifier with the usual syntax, prop is a proposition constructed by logical operators of Isabelle/HOL, listexpr (called list-expression) is constructed by Isabelle’s list functions like hd, tl, nth, and type are (virtually) all types declared
in Isabelle's version 99. tac stands for tactics, tacticals are written in typewriter font.

% (for λ-abstraction), let...in and if...then...else are already defined in HOL, while checks the given formula, repeat, try, or depend on whether their subexpressions are applicable (see p.5). Tactics to be done in parallel can be modeled by two tacticals, by let...in or by or.

A script solving the maximum-example is mainly concerned with data-transfer to subproblems, and could look like

```
Script Maximum_value (fix : bool list) (m~ : real) (rs~ : bool list)
  (v~ : real) (itv~ : real set) (err~ : bool) =
    (let
      e~ = (bd o (filter (Testvar m~)) rs~);
      l~ = (if #1 < Length rs~
      then (Subproblem (Reals, ["make_fun", no_met]) [m~, v~, rs~])
      else (bd rs~));
      m~ = Subproblem (Reals, ["function", "max_of", "on_interval"],
          maximum_on_interval) [ l~, v~, itv~ ]
    in (Subproblem (Reals, ["tool", "find_values", ] find_values)
        [ m~, (Rhs l~, v~, m~, (dropWhile (ident e~, rs~)))))
```

Testvar filters that relation, which contains the variable describing the value to be maximized. The tactic Subproblem (Reals, [make_fun], no_met) [m~, v~, rs~] deserves special attention: it takes the arguments domain Reals, problem-type [make_fun], and the formalization [m~, v~, rs~], but it does not specify a method (no_met). This can be done because of the mechanism of refinement, Def.6: dependent on the modeling of the rootproblem, one of the formalizations is passed in by the formal parameters of the script; when calling the subproblem, the problem-type is refined to ["make_fun", "by_elimination"] or ["make_fun", "by_new_variable"]. This mechanism definitely generalizes the call of subproblems in comparison to function calls in other program languages.

Another script, typical for rewriting, describes a method for solving an equation containing square-roots, where the equation can be solved by isolating the roots and squaring the whole equation, is as follows

```
Script square_equation (eq~ : bool) (v~ : real) (err~ : real) =
    (let e~ =
      (while (not o is_root_free)
        %e~ (let
          e~ = try (Rewrite_Set simplify False) eq~;
          e~ = try (repeat (Rewrite_assoc_plus_inv False)) e~;
          e~ = try (repeat (Rewrite_assoc_mul_inv False)) e~;
          e~ = try (Rewrite_Set isolate_root False) e~;
      e~ = try (Rewrite_Set simplify True) e~;
      e~ = try (repeat (Rewrite_assoc_plus_inv True)) e~;
      e~ = try (repeat (Rewrite_assoc_mul_inv True)) e~;
      e~ = try (Rewrite_Set isolate_root True) e~;
```

\[ \text{in } ((\text{Rewrite \ square\_equation\_left \ True}) \text{ or } \text{(Rewrite \ square\_equation\_right \ True)}) \text{ e}_ wrench) \]

\[ e_{\text{wrench}} = \text{try (Rewrite\_Set\_Inst \{bdv, v_{\text{wrench}}\} \text{ norm\_equation False}) e_{\text{wrench}}} \]

\[ L_{\text{wrench}} = \text{Subproblem (Reals, ["equation", "univariate"], no\_met) [ e_{\text{wrench}}, v_{\text{wrench}} \text{ err}]} \]

\[ \text{in Check\_elementwise } L_{\text{wrench}} \text{ Assumptions} \]

where Rewrite has a boolean argument pushing the assumptions of conditional rewrite-rules into the global assumptions or not.

### 4.2 Reactive user-guidance

What we call ‘reactive user-guidance’ here is a consequence of the math-engine's design: the math-engine is capable of automated modeling (given a formalization) and of automated specification and refinement, as already discussed. The user, on the other hand, is free to supply a step of his or her own choice, and the math-engine will give feedback. The feedback during modeling concerns unknown or missing items and the suitability of problem-types during specification. If the user gets stuck, the math-engine is able (in principle) to supply the next step—in reaction to previous steps of the user.

This concept is easy to implement for modeling and specifying, but it had turned out to be hard for solving (which is guided by scripts). At each tactic, Subproblem, Rewrite\_Set\_Inst, Rewrite\_Set, Rewrite etc., control is passed to the user. The interpreter of the scripts is concerned with the following tasks:

1. beginning with the last tactic done (or the root of the scripts body)
   find the next tactic to do; this may fail due to a ‘misleading’ tactic
2. present this tactic, i.e. the user-tactic associated to the script-tactic found, to the student as a suggestion for the next step
3. receive the student’s input (assumed to be a tactic here for reasons of simplicity, and not an expression; a tactic, however, which may be different from the one suggested by the script) and check if it is applicable at the present proof-state; if so, apply and promote the proof-state, otherwise notify the student
4. locate the tactic associated with the input tactic, which may be classified as misleading; continue with 1.

This kind of interpretation (the details of which belong to the domain of compiler construction, see [Neu01a], and which exceed space limits) is a major advance towards a flexible dialog, where the system and the user are partners on an equal basis: the user can be active and propose tactics,
and the system checks them for applicability at the present proof-state, or the user passes activity to the system and requests a proposal for the next step.

The reader may have noted that the scripts do not contain any hint for the dialogue; this is done in a separated module.

4.3 On the correctness of scripts

The application of a method, i.e. the evaluation of a script, is guarded by a specification of the problem containing a post-condition. It has been mentioned on p.6, that some postconditions cannot easily be verified for a particular example, e.g. for the maximum-example.

In this case it would be an interesting challenge for automated deduction to verify the correctness of the script w.r.t. the post-condition. It is not the example constructed, which is of concern, but rather the algorithm constructing such examples is of concern (one level of abstraction higher)!

Such a proof of correctness would involve the semantics of scripts, which are formulated in Isabelle/HOL, i.e. meta-language and object-language could be within the same logic. [Nil98] could be a model for this task. This proof would also concern the semantics of tactics involving the proof-tree and its branch-types; this is an even harder task. Altogether, it’s an issue, but not considered urgent for the practical usage of the prototype.

5 Explanations by reflection

Reflection is a term, which recently has become folklore with the program-language Java; it denotes facilities of the software-system to inspect its own language-constructs. Reflection in context with explanations means, that requests for explanations should not somehow be foreseen by an author of the system; but rather the system should just reflect its state and its knowledge and only on user request will the system exhibit its knowledge and mechanisms in reasonable portions and steps towards more and more details.

5.1 A separated layer of explicit knowledge

Math knowledge is separated from the math-engine, which interprets the knowledge. Due to verbal advice from the creator of Theorema©[BJ98])
Figure 1. The three-dimensional universe of mathematics

the knowledge is organized along three axes (see Fig.1): the axis of problem-types as presented in 3.2, the axis of methods from 4.1, and the axis of domains, which has already be given by Isabelle’s hierarchy of theories [Pau94].

The user will access this knowledge in leisure browsing, but also with particular requests arising from special situations. This raises issues related to the high structure of the knowledge; the details are not yet designed.

5.2 The representation of the proof-state

is done on a work-sheet, one of which can be found on p.4. A work-sheet produced by the script square_equation on p.12 could look like

\[ L = \text{solve\_root\_equ} \ (\sqrt{9 + 4x} = \sqrt{9 + 5 + x}) \ (bdv = x) \ (c = 0) \]

1. \[ \sqrt{9 + 4x} = \sqrt{9 + 5 + x} \]
   \[ \text{Rewrite \ square\_equation\_left}, \ a \geq 0 \land b \geq 0 \Rightarrow (a = b) = (a^2 = b^2) \]

   1.1. \[ (\sqrt{9 + 4x})^2 = (\sqrt{9 + 5 + x})^2 \]
       \[ \text{Rewrite\_Set\ simplify} \]
   1.2. \[ 9 + 4x = 5 + 2x + 2\sqrt{5x + x^2} \]
       \[ \text{Rewrite\_Set\ isolate\_root} \]
   1.3. \[ \sqrt{5x + x^2} = \frac{(a + 4x) - (b + 2x)}{2} \]
       \[ \text{Rewrite \ square\_equation\_left}, \ a \geq 0 \land b \geq 0 \Rightarrow (a = b) = (a^2 = b^2) \]

11 The theorems use the syntax of Isabelle/HOL, which uses = in a strange looking way: \( (a = b) = (a^2 = b^2) \) etc.
1.4. \((\sqrt{5x} + x)^2 = \left(\frac{9+4x}{(5+2x)}\right)^2\) \hspace{1cm} \text{Rewrite, Set simplify}

1.5. 5x + x^2 = 4 + \frac{1}{2}x + x^2 \hspace{1cm} \text{Rewrite, Set, Inst \([bdv,x])\} \text{ Normalize, equation}

2. \(x - 4 = 0\) \hspace{1cm} \text{Subproblem \((R, \text{equation, univar}, \epsilon)\)}

3. \(L_1 = \text{solve, univar} \ (x - 4 = 0) \ (bdv = x)\) \hspace{1cm} \text{Apply, Method \((R, \text{solve, linear})\)}

3'. \(L_1 = \{4\}\)

Check elementwise \(0 \leq \sqrt{x} + \sqrt{5 + x} \land 0 \leq 9 + 4x \land 0 \leq x^2 + 5x \land 0 \leq 2 + x\)

\[L = \{4\}\]

Reflecting the proof-state, such a work-sheet cannot be edited arbitrarily by the user; there are various solutions possible to this requirement, see for instance [Asp00]. Another model is [Wen99].

For the user-group envisaged, i.e. high-school students, the work-sheet should look like a calculation done by paper and pencil. On the other hand, many technical details need to be displayed, as soon as the user starts to inquire. In the example above, for instance, at line (1.) and (1.3.) the applied theorem contains a condition, which needs to be instantiated and stored with the assumptions of this (sub-)problem. How should this be displayed? Many similaR design decisions have still to be made!

6 Conclusions

The ‘Calculusus-approach’, attempted in this paper for the design and the development of educational software for mathematics, lead to concepts and a prototype, very different from existing algebra systems and presumably simpler to use than deduction systems. This prototype demonstrates the feasibility of the concepts concerning interactivity, and it is used to make teachers aware of what they can request from state-of-the-art math tools [Neu01b].

6.1 Conceptual achievements

Conceptual achievements, established by merging techniques from algebra systems and from deductive systems and shown as feasible by the prototype, are the following.

- A proof-state justifying each step as correct or incorrect (i.e. checking a tactic applicable or not applicable w.r.t. the current proof-state, see 2.2), or eventually misleading
- The knowledge in a separate language layer, i.e. the 3D-universe of math in human readable format (5.1); this can be inspected by the user and interpreted by the math-engine as well
- Interactive and automated specification of a domain (i.e. an Isabelle theory), a problem(-type, see sect.3.2), and a method
– Step-by-step execution of methods, given by scripts whose interpreter passes control to the user at each tactic and resumes guidance (4.2)
– The prototype’s math-engine is ready for implementing the math-knowledge (about 70% of the syllabi are appropriate).
– The math-engine as implemented is an appropriate basis for establishing a novel kind of dialog, where the system and the user interact as partners on an equal base.

6.2 Open questions (?), and future work (!)

– Formal underpinning of the semantics of tactics (2,2) and tacticals (4.1), in order to prove a script w.r.t. the postcondition of the respective problem?
– Completeness of the set of tactics (2,2), and their usability for students (to be investigated in field-studies)?
– Problem-hierarchy appropriate to incorporate all math-knowledge for high-school and to provide for adequately efficient problem refinement?
– Re-engineering of the basic functions of algebra systems (solve, simplify, differentiate etc.) and implementation along the respective axes in the 3D-universe (5.1)!
– Enhanced interactivity in the prototype: dynamic search in the 3D-universe, go back to previous proof-states, high-level description of dialogs and user-model!

The approval, which the prototype has already gained from teachers, seems to make worth the effort to continue clarification of theoretical foundations, to implement the math knowledge according to the syllabus, and to extend the prototypes functionality.

References