Reactive User-Guidance by an Autonomous Engine Doing High-School Math

Walther A. Neuper

Dissertation

vorgelegt zur Erreichung des akademischen Grades

Doktor der Technischen Wissenschaften

an der

Technischen Universität Graz

Graz, April 2001

ABSTRACT

Computer algebra systems (CAS) have been widely introduced to math education at high-schools during the last decade. These modern tools, indispensable in engineering and science, greatly motivate students and teachers as well. However, didactics of math also exhibits harmful consequences of CAS: (1) basic math skills are skipped by 'pressing a button', and (2) CAS introduce the necessity to *specify* problems which is considered an important, but hard task.
This thesis aims at the development of a software component, called 'the tutor', which copes with both harms, by (1) reestablishing a step-mode towards solutions as principal mode, and by (2) supporting all phases, modeling, specifying and solving a problem. The tutor is specified with the requirements, to autonomously model-specify-solve a problem, to assist a student in independent attempts to do so, to work with logical rigor like computer theorem prover do, to keep math knowledge separated from the deduction component, and thus to make the tutor extensible to the majority of mathematics problems.

The thesis shows that the construction of such a tutor can rely on well established concepts and techniques of formal mathematics and software technology. There are, however, three decisive original contributions to be made by the thesis: (1) problem-types related to formal methods, (2) proof-scripts belonging to compiler construction, and (3) dialog-atoms from human-computer-interaction. These three concepts accomplish three novel tasks: (1) allow for mechanical search on hierarchies of problems, (2) provide for user guidance resuming after input of variants of a calculational proof, and (3) lay the foundation for implementing highly flexible and structured dialogs.
These three novel concepts and components make the standard techniques work together as required. This is shown by a prototype implementation, which also roughly illustrates the usability of such a tutor.

ZUSAMMENFASSUNG

Computer Algebra Systeme (CAS) sind im Verlaufe der letzten Dekade in beträchtlichem Ausmaß in den Mathematik-Unterricht an Höheren Schulen eingeführt worden. Diese modernen Werkzeuge, unverzichtbar in Technik und Wissenschaft, motivieren Studenten und auch Lehrer. Die Didaktik der Mathematik vermeldet jedoch auch Bedenken zum Einsatz von CAS: (1) grundlegende Fertigkeiten werden durch 'Knöpfe drücken' übergangen, und (2) CAS verlagen mehr Aufmerksamkeit bei der Problemspezifikation, was schwierig zu unterrichten ist.
Diese Dissertation zielt auf die Entwicklung einer Software-Komponente, genannt 'der Tutor', die beide Bedenken zu entschärfen sucht, durch (1) Einführen einer schrittweisen Ausführung, und durch (2) Unterstüzen aller Phasen, des Modellierens, des Spezifizierens und des Lösens von Problemen. Der Tutor ist auf die Anforderung festgelegt, Spezifikation und Problemlösung autonom durchführen zu können, den Lernenden in seinen eigenen Versuchen zu unterstützen, darüberhinaus logisch exakt zu arbeiten wie Theorem-Prover, das Wissen getrennt von der Rechenmaschine zu halten, und so den Tutor erweiterbar zu machen auf einen Großteil des Mathematikstoffes an Höheren Schulen.
Im Rahmen der Dissertation wird zu zeigen versucht, daß sich die Entwicklung solch eines Tutors weithin auf bekannt Konzepte und Techniken aus Computer-Mathematik und Software-Technologie stützen kann. Drei originale Beiträge jedoch sind zum Erreichen der Ziele notwendig: (1) Problem-Typen in Anlehnung an das Gebiet der Formalen Methoden, (2) Beweis-Skripts aus dem Bereich des Compilerbaus, und (3) Dialog-Atome aus Mensch-Maschine-Kommunikation. Diese drei Konzepte bewältigen neuartige Aufgaben: Punkt (1) erlaubt mechanische Suche in Hierarchien von Problemen, (2) ermöglicht die Benutzerführung nach alternativer Benutzereingabe wieder aufzunehmen, und (3) legt die Grundlage für flexible und strukturierte Dialog-Steuerung.
Diese drei neuartigen Konzepte zusammen erlauben den hochgesteckten Zielen wesentlich näher zu kommen. Eine Prototyp-Implementation zeigt, worin die Nützlichkeit des Tutors liegen kann.

ACKNOWLEDGEMENTS

This work would never had started if Dines Bjørner, at that time director of UNU/IIST - Institute of Software Technology of The United Nations University, would not have pursued the idea of a new kind of educational software for mathematics, and if he would not had involved me as a teacher into a respective projekt initiative in 1993.

Peter Lucas guided my way towards an interactive math software with great circumspection and awareness, consistently all over the years. Without his advice I could not have tackled the issues of interactivity. In 1999/2000 he gave me the tremendous opportunity to work at his institute for a sabbatical year, where he joined my efforts for implementing a prototype with those of other students. Many thanks also to the staff of his institute !

Bruno Buchberger inspired me with his keen views of mathematics, and gave me a completely new perspective of teaching mathematics (after having been a teacher myself for more than two decades !). The challenging lessons he gave enabled me to tackle the formalization of mathematics knowledge with the goal of mechanical interpretation.

The Department of Scientific Computing at the University of Salzburg generously approved to use its computer system for major parts of the practical work.

Salzburg, April 2, 2001

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION: MOTIVATION BY DIDACTICS

The introductory chapter explains the relevance of and the demand for the new component of an 'artificial mathematician', called 'the tutor' in the sequel, specifies the user requirements, and compares with similar existing software tools.

The relevance of software tools in mathematics education is made evident by a survey on the huge efforts of school administrations to provide computer equipment as well as teacher training, and by documenting the dominating role of discussions on computer algebra systems (CAS) in didactic of mathematics.

The discussion shows much enthusiasm about the advances of the software tools for math education, but also raises seriously argued warnings. From these warnings the demand for a new software tool is being derived, which merely will be a complementary component to existing systems.

The tutor is specified with exciting new kinds of functionality. The specification is given as user requirements specification, and a fictive sample session with a student illustrates the features to be introduced.

Last but not least the topic is approached from the technical side, and the lush scenery of math software tools is searched for those which best meet the stated requirements: The first investigation concerns their features, techniques and components for appropriateness to the requirements, and the usability for constructing the specified new component. The second investigation is the other way round and asks for what is missing in the most appropriate tools.

Based on this twofold investigation the choice for one existing software system, to start prototyping the tutor with, is justified.

Finally the user-requirements are compiled into four more technical requirements, a mathematics tutoring system of the state of the art should meet. These requirements circumscribe the scope of the thesis. After the presentation of the concepts developed within the thesis, these concepts and their implementation in a prototype are checked w.r.t. the four requirements.

1.1 Software-tools in mathematics education

The dream of an 'artificial mathematician' has first been articulated seriously by [Bun83]. Much of the dream has become reality in the form of CAS, theorem provers and other software-tools. CAS have been used by scientists and engineers for almost two decades, and during the last decade they have become a matter of course in classrooms at highschools.

1.1.1 The impact of software-tools

Less than ten years after the 'pocket-calculator revolution' another development in technology lead to a wave of changes in mathematics education. Spreadsheets and computer-algebra systems (CAS) with their respective graphing capabilities opened new possibilities. The didactic discussion immediately took over this topic; it is surveyed in [Fuc98] (with bias to Europe). School-administration and institutions responsible for teacher training reacted quickly in countries like Austria and United Kingdom. Other countries and school districts followed this example [1].

In the United States the Teachers Training with Technology (T^3) started in 1993, and since then reaches several thousand teachers per year. This program has been taken over by a commercial firm and is offered worldwide now [2]. Austria belonged to the pioneers among the European countries and was one of the first to provide all students at high-school with a general license for a computer algebra system, derive [Sof94], in 1990.

Although the impact of several kinds of software-tools is well recognized in the didactics literature [TJ98], and in particular the usefulness of spreadsheets in mathematics education is thoroughly discussed [Neu98], in Austria (and in most other countries, too) CAS drew all the attention upon them.

An 'Austrian Center for Didactics of Computer Algebra' (ACDCA) was founded by the federal institution for teacher training [3]. ACDCA organizes yearly national and international conferences, and systematically collects and distributes teaching examples and examinations involving CAS.

In 1993/94 the Austria federal ministry for education launched a field-research project ([AFHK94] and [AF96]) developing, testing and evaluating example lessons with CAS ([Gro95] and [Noc96]). Eight hundred students were involved in the first phase, thousand-five-hundred in the follow-up project. In 1999/2000 it will be continued involving classes all over the country, while investigating learning-situations with CAS, developing a teachers handbook for mathematics-education using CAS, CAS in examinations, and new ways of students interaction and classrom activity using CAS [4]. Spe-

[1] http://www.derive-europe.com/deutsch/Referenzen.htm
[2] see http://www.t3ww.org/t3/t3info.htm
[3] see http://www.acdca.ac.at/german/index.htm
[4] see http://www.acdca.ac.at/projekt3/index.htm

cial considerations for technical high-schools are in [Sch91], and an interest group, called AMMU, has been established for this type of schools in Austria [5].

Both, ACDCA and AMMU, regularly distribute teaching materials on CAS to teachers in the respective types of schools. Also textbooks for mathematics education one by one have begun to include examples dedicated to CAS application and sections on the handling of particular CAS ([S+94] and [R+92]). The usage of CAS in examinations has been discussed and actually tried from the very beginning ([Wur96] for Austria and [WW91] for Bavaria)

1.1.2 Expectations and warnings on the use of CAS

The remarkable efforts concentrating on increasing usage of CAS are backed by high expectations, which had been developed in discussions on didactics in mathematics. These discussions were really vivid all around the world beginning with the early rise of CAS, which, in the authors opinion, reflects the fact that in the meanwhile computers had gained much in acceptance, and that symbolic manipulations are more essential for mathematics than numeric capabilities.

Expectations on advances for mathematics education by using CAS are mentioned in the literature in a large number, which may be compiled as follows.

Concentration on the essentials of mathematics is supported as CAS may be used as a work-horse for the 'drudgery of performing long and tedious mathematical calculations' [Kut97]. This coincides with rather old requests [Len96], not to emphasize algorithmically oriented examples so much, and a very recent study [BL+97] compromising particularily math education in German speaking countries not to teach students practical problem solving. Using CAS is expected to help for better ratings in the future [Wei97]. Several studies suggest direct bearings on 'process-object encapsulation', i.e. support for the student to develop concepts of abstract objects by encapsulating (calculational) processes [HMJR93], [MST94] [6].

Experimentation may be supported by the same kind of work-horse and the graphics features, fostering a wellknown pedagogic issue: proceeding in a psychologically natural way from observation of individual cases to general and formalized notions [TKW97]. This way may rerun the historical

[5] http://members.ccc.at/ ammu/

[6] An example for process-object encapsulation is a notion of function as an object having certain properties (continuous, ascending, differentiable etc). A natural predecessor of the concept as an object a function is the experience of a special kind of process – evaluating a given term for many, many values (of the bound variable).

development of mathematics [Wit81], and experience helps to strengthen intuitive prototypes of abstract concepts [Dör91].

Modelling is considered an important, but still neglected part of maths education [Hey96]. The transformation of real world problems into models may be supported by CAS, because they do not complain about hughe numers with many digits behind the comma, and they can handle and analyse long lists of empirical data. CAS also allow to recalculate several variants of a model with a keystroke, thus fostering an intuitive validity-check by the student [Fuc98]

Students motivation for mathematics may improve by more pleasant classroom activities like discovery-, independent- and socal-learning. CAS provide for a lot of new opporunities to establish such classroom activities (and in fact, most of the articles in the dedicated journals stress this aspect). Improvements in the students (and teachers) motivation have been investigated by [Gro95] (and [Sve95]) in Austria, [Bow97] in England and [AL97] in France.

Promote unsettled issues is another expectation in the didactics community: CAS are supposed to be vehicles to promote issues like structuring mathematics education by 'fundamental ideas' [AFS96] or multiperspectively (graphics, data, formal) representing problems and emphasize describing, arguing, (preformal) proving [Fuc98]. While these changes are meant to occur without changes in the syllabi, development of maths curriculi is being discussed in relation to CAS, too.

Warnings of a slash back on an unapproriate use of software-tools in mathematics education and some technically inherent deficiencies of CAS establish the other side of the medal. They are identified in the literature on didactics in mathematics as follows.

Formula manipulation doesn't receive enough attention in CAS-based education. Very early doubts were stated, that CAS would emphasize the 'high-level' structures and neglect the 'low-level' structures in maths and thus the confidential base would be lost [Sch91]. A detailed analysis [Rec98] identifies a 'key stroke paradigm', justifying maths objects by key stroke, e.g. a real number by $\sqrt{6}$, where the notion is neither backed by procedural skills ($\sqrt{6} = \sqrt{3} \cdot \sqrt{12} = \cdots$) nor by a clear concept (a real number as a limit, or as a Dedekind cut etc.). At this reference special examples can be found which show that 'meaningless' (syntactically oriented) manipulations are the best way to a solution, claiming that the expression 'less computation, more conceptual comprehension' is not yet well understood.

Ability differences are being stretched by CAS biased maths education. This may be not only because some students more likely go ahead in exploring a CAS on their own, but because 'niches' for 'weak students' vanish. A study on final exams (Abitur) in Bavaria [WW91] shows, that a greater percentage of the examples for the low-level course than for the high-level course can be solved by a CAS – 60% for the low-level course and 50% of the high-level course. And it is concluded that exams, CAS allowed, would stress such examples, which are considered more appropriate for high-level courses in traditional written examinations. A more recent opinion [Mon97a] calls the question still open, but demonstrates, that quickly invented examples usually become harder, if they are of a kind not giving an advance to the use of CAS to solve them.

Too powerful a tool for beginners the CAS are being considered. [Sta97] draws a comparison to software filters (e.g. '*NetNanny*') for the internet, protecting children from material that is not considered suitable for them. Should second-graders be protected from complex numbers when typing *solve* $x^2 + 2 = 0$? Another case where a student may get a question answered by notions much more complex than in the question itself, is solving a well known real world problem by *solve* $x(L - 2x)^2 = 0.05$ and obtaining results with trigonometric functions ?

Handling of a CAS is an issue of its own, unlike using paper and pencil. This causes a real dilemma for teachers: Is the CAS introduced before doing the mathematics, in which case it is somewhat vacuous, or is it introduced alongside the learning of mathematics, in which case the technicalities may detract from the learning of mathematics [Mon97a]. Another argument is found in [Mon97b] where considerable differences in operations are stated between solving a problem by hand or by a CAS, headed by the question 'what are they doing ?'.

Validating results becomes a must, and often this concerns very technical aspects. Not only the input of formulae in a 1-dimensional format (and using parentheses) causes problems to beginners (which my be an important occasion for learning [Kut97]), also the output [Fuc98]:

$$\textit{expand}\ (\frac{2x+3}{x-2}, x)$$

results in

$$\frac{3}{-2+x} + \frac{2x}{-2+x} \quad \text{or} \quad 2\frac{x}{x-2} + 3\frac{1}{x-2} \quad \text{or} \quad \frac{7}{x-2} + 2$$

depending on the CAS employed, Mathematica, Maple or Derive. Comparatively different formulae may result within one and the same system, if some switches are set differently.

Unstructured presentation of calculations in CAS is a step back wrt. paper and pencil. The calculation written by hand in an examination or in a homework was a mirror for the students thinking, and experienced teachers know to use that mirror. [WW91] regards the representation of a calculation in a CAS (the DOS-version of Derive at that time) as not structured enough to be accepted as a printout for an examination. The graphic representation on the screen has become better with the windows-version, and it is comparable with other CAS now. But still, the logic consistency between subproblems is not captured in any way in the commercially available CAS.

1.1.3 Why not *adapt the* tools *?*

The discussion in didactics of maths as described above could be interpreted as suggesting efforts to adapt the lessons to the tool used in the lessons. Why not vice versa adapt the tools to the requirements of maths lessons ?

Interestingly enough, the latter almost never has been claimed in the literature. One exception is [MT94] discussing general user interface design issues, another [Age92] who asks for how to 'naturalize computer algebra for mathematical reasoning'.

Now, why not claim for a tool which ancitipates the warnings mentioned above while maintaining the advances of CAS and other systems — and thus adapt those tools which initially have not been designed for educational use !

In particular, why not construct a tool which assists the student in formula manipulation, and which guides him or her step by step through a calculation ?

If modelling is considered so important, why not provide for guidance in that task by a system ? Why not make crucial parts of modelling explicit, formalize the problems taught at schools and structure them in a way that a student can identify a given example as belonging to a particular (type of) problem, just as a technician searches the store for an appropriate spare part ? Isn't it an actual issue in contemporary maths education ?

And explicitly (and understandably) specifying the domain and the method, wouldn't that control the power of the mathematics engine ? Why not such a type of tool ?

Why not make the dialog modi so flexible that a student can learn (e.g. by watching the system solving an example, instead watching the teacher writing to the blackboard who cannot be interrupted for questions any time) and do written exams within the same environment ?

If handling a CAS is an issue, why not provide for a system which performs calculations just as done by hand, and which *gradually* may lead into specific CAS functionality ?

If checking results is a problem, why not incorporate the basic technique (basic at least in software technology; in maths this still seems to be the teachers task when assigning marks) of checking the postcondition into the

system ?

These questions are specified in the form of a narrative user requirements definition in the following section.

1.2 User requirements for a new tool

The dream of an 'artificial mathematician' has first been articulated seriously by [Bun83]. As old as this dream is the issue of 'teaching children to be mathematicians versus teaching about mathematics' [Pap72]. It is time to join these two ideas !

1.2.1 Some termini technici

In order to make the narrative version of the user-requirements specification better comprehensible, let us introduce the following termini (where many of them will be formally defined in chapter 2):

formula: formal text describing mathematical objects. Formulae evaluating to 'true' or 'false' are called *predicates*, all others are called *expressions*.

rule: a theorem or an axiom. If the rule can derive a given formula into another formula (in a given context), the rule is called *applicable* (to the given formula).

step: the derivation of a given formula into another one by a rule, or the proposition of another formula which can be derived from the given one by some rules.

example: a part of a textbook on mathematics in high-schools supplying some data together with raising a question the student should answer by applying mathematical methods.

description (of an example): text, formulae and eventually drawings which an author of a textbook supplies in the context of an example.

formalization: The description of an example refined to formulae only. This process of refinement is called *modeling*.

domain: a separate part of mathematics knowledge containing definitions, axioms and (proved) theorems.

problem: a structure of formulae capturing what is considered common to a collection of examples: the objects given (called the *prerequisites*), the objects to find (called the *goal*) and predicates over these objects.

method: a structured sequence of rules deriving the objects given in an example (belonging to some problem) into the objects belonging to the goal, and eventually subproblems. The objects belonging to the goal are called the *solution* of the example.

specification: the mapping of a domain, a problem and a method on an example. The process of establishing such an appropriate mapping is called *specify* the example.

calculational proof (or simply *calculation*): the sequence for steps deriving the solution of an example from the given objects.

Remark: According to the above termini we say 'solve an example' instead of 'solve a problem', in order to make clear that an exmple needs to be specified to become a problem, before a method can be applied appropriately. However, 'problem-solving' is such a fluent expression, that we will use it instead of 'example-solving', if there is no danger of confusion.

Several of these termini concern a special view of problem solving, which may be regarded as a part of the creativity-spiral [Buc93]. There are three phases, each of which starts with a particular kind of formulation of the example under consideration, and each of them is geared towards generating a well defined (intermediate) result.

Fig. 1.1: The three phases of problem solving

The three phases have been defined verbally above, they are shown in Fig.1.1 together with their input and output, and an example is given on page 11.

1.2.2 As a high-school student I would like to ...

Given an example in a textbook on mathematics in high-school, as a student I would like to

- see the calculation of an example presented on the screen as I am used to have it in paper and pencil work
  - get the calculation without remembering special CAS-commands or searching them in menus
  - see the calculation in steps neither too wide for understanding them nor to narrow to get bored

  - edit a calculation by deleting, inserting a formula, and continuing at an earlier step with another variant (all supervised by the system for logical consistency)
  - get a printout of the calculation of an example and have my homework or written examination done

- get support in all phases, in modeling and specifying an example, as well as in solving it
- get both, a demonstration of how a given example can be solved, as well as get help in my own attempts to solve the problem
- get a complete demonstration of how to stepwise solve the example just by pushing a button for each step
  - get information (on demand only !) about the prerequisites and (sub)goals
    * see problems similar to the one the example belongs to
    * see the definitions, axioms and theorems in the domain involved
    * see alternative methods I could use to solve the example
  - get the calculation in steps as I would do by hand
    * zoom in for details and zoom out for a survey in a long calculation
    * see alternative rules applicable at the current formula
    * see (in an animation) how a rule is applicable (at a sub-term)
- get support in solving the example on my own
  - get hints while modelling and specifying the example
    * which objects are missing
    * which objects are wrong
    * which predicates on which objects are not true, indicating that the example may not belong to the type of problem
    * which method is applicable to the example
  - get support by the system in providing an applicable rule to input
    * by a list of actually applicable rules
    * by an applicable rule shown partially
  - get support by the system in continuing the calculation by input of the next formula
    * by presentation of the rule to be applied
    * by a partial presentation of the next formula

  - redo a calculation from a previous step, if I want to explore an alternative or if I have gone astray

- get all my input checked such that no faulty calculation is possible
- get my questions answered (*only !*) if I ask
  - which rule derived a formula at any place in the calculation
  - where the current formula comes from
  - what the current goal is
- solve an example *unknown to the system*, and nevertheless get support after having specified the example (no help during modeling in this case)

Last but not least two requirements concerning teachers and course designers:

- Anybody willing to provide his or her students with special examples or example collections, should be enabled to do so without low-level programming (using C^{++}, Lisp or ML). Rather the example collections and the math knowledge should be extensible in a representation close to the traditional language of mathematics.
- Various kinds of course organisations, of learning strategies and of classroom activities should be supported by different dialog modi, which should operate on one and the same representation of math knowledge.

All the requirements and choices between them should be available at any time, and the student should be able to switch from passively consuming the tutoring-systems presentation to a more active involvement. A full fledged dialog-guide should enable a course-designer to predefine the dialog-mode (this issue exceeds the scope of this thesis, but is considered crucial for future development), and for instance to enforce a certain amount of active involvement of the student (which could provide for applicability of the tutor in written examinations).

A more elaborated and technical specification of the requirements above has been done in preparing a graphical user interface for the tutor [Fin00a].

1.2.3 An example session

According to the user requirements above a 'reactive user-guidance component' (abbreviated as 'the tutor') will be developed in the subsequent chapter. What follows is the description of how a session of a high-school student with such a tutor could go. The tutors *output* is labeled by 'tutor', the students *input* is labeled by 'student'. In order not to produce the

impression of a 'magic session' footnotes briefly explain how the dialog is guided by the tutor.

Model a formalized representation from the description given in some kind of electronic textbook is the first task:

> *A coil with a circle-shaped section and radius R should get a cross-shaped kernel (two equal bars with length b and width a) of iron, see Fig.1.2. The area A of the kernels section should be maximal for a given R.*

Fig. 1.2: Coil with cross-shaped kernel to be maximized

The student first will try to transfer the information given by the text and the drawing into some formulae. This task is called the model-phase. If the dialog is in a mode engaging the student, it will present the following frame for input:

given:	*constant_values* : [ ]
find:	*maximum* :
	other_values : [ ]
relate:	*relations* : [ ]

where [ ] indicate that lists of formulae are to be input [7]. Let us assume the student isn't sure how to start the calculation and pushes the `yourTurn` button (as button denoted by angles ⟨`yourTurn`⟩):

[7] The suggestively named keywords *constant_values, maximum,* etc. stem from hidden information prepared with each example, called the (type of) problem. Another part of hidden information concerns the formalization in all reasonable variants:

variant I
$[\,[R = ArbFix],\ \ 0.0 \leq a \leq \frac{R}{2},\ \ [A = 2ab - a^2,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = R^2]\ \,]$
variant II
$[\,[R = ArbFix],\ \ 0.0 \leq b \leq \frac{R}{2},\ \ [A = 2ab - a^2,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = R^2]\ \,]$
variant III
$[\,[R = ArbFix],\ \ 0.0 \leq \alpha \leq \frac{\pi}{2},\ \ [A = 2ab - a^2,\ \frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha]\ \,]$

This hidden information enables the tutoring-system to support the student in the model- and specify-phase

student: ⟨yourTurn⟩
tutor: constant_values $[R =$Arbfix$]$
student: ⟨accept⟩
tutor: maximum A
...

Now let us assume the student has become self confident enough to proceed on her or his own, and wants to input which other values must be 'found' capturing the shape of the kernel:

...
student: other_values $[a, b]$
tutor: accept
student: relations $[A = ab + ab]$
(... forgets to subtract the square in the middle of the kernel)
(1) tutor: $A = ab + ab$ does not belong to this example !
student: $A = ab + ab - a^2$
tutor: accept
student: apply method 'maximum_by_calculus'
(... tries to finish the specification too early)
(2) tutor: relations $[A = ab + ab - a^2]$ is incomplete !
student: yourTurn
tutor: $[A = ab + ab - a^2,\ \frac{a}{2} = R \sin \alpha]$
student: howComes
(3) tutor: definition of sin !
...

[8]

The model-phase may result in the following:

given: *constant_values* : $[R = ArbFix]$
find: *maximum* : A
other_values : $[a, b]$
relate: *relations* : $[A = ab + ab - a^2,\ \frac{a}{2} = R \sin \alpha,\ \frac{b}{2} = R \cos \alpha]$

This is the input to the next phase.

Specify domain, problem and method: The specify-phase adds structures capturing the goal-directedness of problem solving. The goal is approached in two steps common to mathematics and software technology: The first

[8] The hints (1) and (2) can be given by use of the knowledge, the system needs for solving the *whole* example collection *once*, i.e. the formalization and the problem. However, the answer (3) would require special preparation by the author for this *single* example. Within the model-phase better support would be possible, if the tutoring-systems knowledge would contain the notions of 'circle', area, inscribed, etc. There is at least one system which tries to give this kind of support [ABY85]

step describes *what* the goal is, and the second step describes *how* to achieve the goal. The descriptions of the two steps, however, needs to declare where to take the necessary notions from. Thus this phase aims at stating three items, the domain, the problem and the method.

With respect to the example given, the student may be led by the observation of the function constants sin and cos, or the numerical constant 0.0 to the input [9]

```
...
student: Specify_Domain ℛ
tutor:   accept
student: ...
```

At this point the student can search the hierarchy of problems if necessary:

```
...
optimization_problems:
         maximum_with_additional_conditions
         minimum_with_additional_conditions
         linear_optimization
         ...
```

Let the user mistakenly select the third problem:

```
...
student: Specify_Problem 'linear_optimization'
tutor:   is_linear_in (A = 2ab − a²) [a, b] is not true
student: Specify_Problem 'maximum_with_additional_conditions'
tutor:   accept
...
```

The specified problem[10] is instantiated by the input formulae, in particular the precondition ('where') and the postcondition ('with') [11]:

problem 'maximum_with_additional_conditions'
given: $constant\_values : [R = ArbFix]$
where: $0.0 \leq R$
find: $maximum : A$
$other\_values : [a, b]$

[9] Input in this phase of problem solving generally is done by selecting items from the knowledge spanned along the three axes domains, problem-types and methods; see Fig.2.1.

[10] The (type of) problem is rather general. In fact is solves all seven examples of 'mathematics at a glance' [GHHK77], p. 426 - 429.

[11] The where-field contains the precondition, and the with-field contains the postcondition. The latter shows up with a formula, which is never in a high-school syllabus. But it is indispensable in formally describing the problem. Thus pre- and postconditions are generated automatically.

with: $A = 2ab - a^2 \ \wedge \ \frac{a}{2} = R\sin\alpha \ \wedge \ \frac{b}{2} = R\cos\alpha \ \wedge$
$\forall A'a'b'\alpha'.\ A = 2a'b' - (a')^2 \ \wedge \ \frac{a'}{2} = R\sin\alpha' \ \wedge \ \frac{b'}{2} = R\cos\alpha'$
$\Rightarrow \ A' \leq A$
relate: *relations* : $[A = ab + ab - a^2,\ \frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha]$

Finally the method is specified (eventually by another search in the knowledge base) and applied:

...
student: `Specify_Method 'maximum_by_calculus'`
tutor: `accept`
student: `Apply_Method 'maximum_by_calculus'`
...

Solve the example is the phase in problem solving which is generally most emphasized in maths courses at highschools. Let us proceed with the 'tutor' being the active partner [12]:

...
tutor: `we substitute` $a \mapsto 2R\sin\alpha$ `in` $A = 2ab - a^2$
student: `accept`
tutor: $A = 2(2R\sin\alpha)b - (2R\sin\alpha)^2$
student: `accept`
...

And now the DG starts to involve the student more and more.

...
tutor: `we substitute` $b \mapsto 2R\cos\alpha$ `in` $A = 2(2R\sin\alpha)b - (2R\sin\alpha)^2$
`Which gives ?`
student: $A = 8R^2\sin\alpha\cos\alpha - 4R^2(\sin\alpha)^2$
tutor: `accept`
`we solve subproblem` *differentiate*
`for` $\lambda\alpha.\ A = 8R^2\sin\alpha\cos\alpha - 4R^2(\sin\alpha)^2$:
$A' = \frac{d}{d\alpha}(8R^2\sin\alpha\cos\alpha - 4R^2(\sin\alpha)^2)$
`Which rule would you apply:`
#1 : (*diff_bdv*, $\frac{d}{dx}x = 1$)
#2 : (*diff_const*, $\frac{d}{dx}x = 0$)

[12] Another preview in order to explain the flexibility of the system: There are two modules, the mathematics-engine (ME) and the dialog-guide (DG). While the ME justifies each step by applying an appropriate rule and suggests a next step due to the methods script working behind the scenery, the DG decides whether this step should be done by the ME (in a demonstration mode) or whether the student should get involved. The DG uses many dialogue patterns which may involve the student in different ways.

In the solve-phase the reactive user-guidance is provided by a socalled (proof-)skript, see chapter 2 and the appendix

$\#3 : (\mathit{diff\_sum}, \frac{d}{dx}(u+v) = \cdots)$
$\ldots$

Notice that the DG skipped the whole specify-phase, i.e. the DG and the ME built the according parts of the prooftree behind the scenery due to the instructional design by the author.

$\ldots$
student: $\#3$
tutor: $A' = \frac{d}{dx}(8r^2 \sin\alpha \cos\alpha) - \frac{d}{dx}(4R^2(\sin\alpha)^2)$
`next formula ?`
student: $A = 8R^2 \cos\alpha \sin\alpha - 8R^2 \cos\alpha$
tutor: `cannot derive the formula !`
$\ldots$

Now, as several 'dialog-patterns' more and more involved the student, he has failed. And the general 'dialog-strategy' active in the dialog-guide will decide how to react to the students failure: by doing similar proofsteps itself, or by presenting an analoguous example as a special exercise, or still another strategy implemented in the DG by the author of the exercises:

tutor: $\ldots$
`which rule did you apply ?`
student: ⟨`dontKnow`⟩
tutor: `Rewrite` $\mathit{diff\_prod}, \frac{d}{dx}(u \cdot v) = (\frac{d}{dx}u) \cdot v + u \cdot (\frac{d}{dx}v)$
student: ⟨`accept`⟩
tutor: $A = (\frac{d}{dx}8R^2 \sin\alpha)\cos\alpha + \ \ldots\ldots\ - \frac{d}{dx}4R^2(\sin\alpha)^2$
`fill in` $\ldots$
student: $\ldots + \ 8R^2 \sin\alpha \ \ \frac{d}{dx}\cos\alpha \ - \ldots$
tutor: $\ldots$
$\ldots$

sectionWhat can we adopt from existing tools ? After having captured the user requirements for the tutor in the previous section, it is necessary to go sure not to reinvent the wheel when approaching an implementation of that piece of software. With this aim related software-tools are checked for useful or missing features w.r.t. the requirements specification. The twofold evaluation is the basis for selection of the most appropriate system for prototyping.

1.2.4 Related systems

The selection of systems considered skips features not immediately related to calculational proof like geometry, graphing or facilities for compiling tables.

Educational software can be dropped. The great bulk of what is advertised for math education in algebra, calculus etc. does not meet by far the requirements of flexible dialog and general problem solving ability. There is one very important exception, the system MathPert [Bee84b].

This commercial system has a thoroughly constructed mathematical basis ([Bee84a], [Bee95] and others), does calculations similar to pencil and paper work, engages the student to select operations to be applied at the current formula, shows the sub-term an operation is applied to, and inhibits incorrect steps in a calculation. There is little missing:

- Mathpert does not include the model- and specification-phase into solving a problem.
- Consequently examples cannot be divided into subproblems, and domain, problem and method cannot be specified explicitly.
- Mathpert does not allow the student to input formulae as intermediate steps, and has no explicit representation of the solving method.
- The individual teacher can extend neither example collections nor mathematics knowledge.

Computer algebra systems, Mathematica [Wol96], Maple [CGG$^+$92], Derive [Sof94] and others more science biased (e.g. [H$^+$93]), dominate the field; their advantages need no further comment. However, there are severe deficiencies w.r.t. the specified requirements:

- CAS deliver the result as a whole with one keystroke. Some of the systems have tracing facilities, but these do not allow to interactively modify the computation in any way.

- CAS do not have a type system; [13] thus it is the users responsibility which method to apply to a particular problem. Selecting the right domain and the right method usually is a task of setting switches.
- The methods employed are hard-coded, and the user cannot inspect them (regardless their complexity which would obviate comprehension by a high-school student in most cases).
- Being designed (at least initially) for use by professional engineers and scientists, user-guidance is not provided in CAS.
- The handling of subproblems in solving an example is left to the user as well.
- Cogent logical rigor was no issue in designing CAS; formally inclined scientists find numerous points for criticism, e.g. [Har97].

Computer theorem provers (CTP) are more mature and more numerous as is known among math educators. Just to list the ones known best: Analytica [CZ93], NQTHM [BM79], Nuprl [C$^+$86], Coq [HKPM94], HOL [GM93], Oyster-Clam [Hor88], [vHIN$^+$89], IMPS [FGT90], Redlog [DS96], PVS [ORR$^+$96].

They all provide for logically sound application of rules (at least one of them even at marked sub-terms [B$^+$92]), the CAS labeled 'interactive' allow for a step by step mode, and they all have a type system preventing them from incorrect inferences. Those CTP labeled 'generic' theorem provers come along with a particularly appealing feature: their knowledge can be extended in a logically concise way – thus having the knowledge on the object-level and enabling the user to inspect it easily.

The CTPs strength in formal reasoning is diminished for educational purposes by the fact, that the proofs are not really readable by humans. A great progress in that respect has been made by Theorema [BJ98] which is particularly dedicated to human readable proofs [Buc97].

Gradually CTP approach what is a matter of course for a CAS, interactive mathematics books: there is already one computer science textbook [Nip98] mechanized by Isabelle [Pau94], one of the recently most developing interactive and generic theorem provers.

In spite of some appealing advantages, the CTPs appropriateness to the requirements specified is limited:

- The knowledge in formal logic necessary to use a CTP is out of the scope of high-school math. And even just inspecting a proof is not

[13] There seem to be only one exception: AXIOM [Dav92], formely known as Scratchpad, was developed at IBM research laboratories over many years before it was offered as a supported product.

helpful, because the representation of proofs is too technical (eventually with exception of the proof checker Mizar [TB85])

- CTP are made for reasoning, i.e. for objects evaluating to 'true' or 'false', while even elementary computations go beyond these two values. In fact, the concept of inference need to be extended for the purpose of this thesis.
- The calculational power of CTP is rapidly increasing in many systems (e.g. by incorporating decision procedures [Sho79] or [AG93]), but still far beyond CAS: no factorization, no integration etc at the time of writing this thesis.
- User-guidance, of course, is no matter in this kind of tool.

Constraint solvers seem to be rather unknown in education, in spite of rather early implementations like TK!Solver [KJ84], prolog-based systems, or NUT [TMPE86] which is based on object technology. The interesting feature w.r.t. the specified requirements is emphasis on modeling. Unlike all other tools, the user gets support in this phase, e.g. by a hint for missing input values. The problem specification is modeled by relations, thus the user may decide dynamically, whether a variable of a given model should be input or output.

Besides these outstanding advances there are deficiencies already mentioned for other tools:

- No stepwise calculation.
- No explicit rule application.
- No extension of knowledge by the teacher.

1.2.5 Components useful for construction

Let us turn from the functional point of view to the constructive point of view: what parts of existing software or which parts of them maybe useful for constructing the tutor specified ?

A parser is one of the most basic components, converting the external representation of formulae to an internal one, appropriate for symbolic manipulation. Thus it must be possible to call the parser from arbitrary functions within the tutor, and to operate on the internal data structure representing a formula.

CAS parse each formula just after input. Mathematica makes the internal data structure accessible by the way of its `Fullform` function, Reduce

as open source product is even more convenient in that respect. CTP offer more or less convenient interfaces to their parsers.

A rewriter, transforming formulae by application of certain theorems, is the basic work-horse for the most typical parts of symbolic computation [14].

The predominant requirement specified for the tutor is a step by step access to the rewriter (i.e. the possibility to only apply one theorem per step). This requirement is naturally met by CTP. Some CAS provide convenient interfaces to the rewriter within their programming facilities (e.g. Mathematica, Maple, not Derive).

None of these tools employ a rewriter marking the sub term where a rule is being applied (with the exception of the Raise-tool which is commercial). Thus it will be necessary to modify the rewriter for the tutor, which excludes commercially closed systems and restricts the choice to open source products like Reduce and some of the CTP.

Implemented knowledge should be available and extensible. Contemporary CAS or CTP represent dozens or hundreds of man-years for implementation of mathematics knowledge, it would not make sense to start from scratch.

Both, CAS and CTP, incorporate knowledge which is extensible: The former can be extended by programming new functions and packages, the latter by implementing new 'theories' containing definitions and axioms very similar to mathematical textbooks, and by proving theorems. A particularly good hierarchical presentation of theories is given by Isabelle. The CTP way of extending knowledge reflects the constituent method of mathematics, whereas the CAS way does not enforce logical consistency in any way.

Thus, when teaching mathematics and explaining its constituent methodology, the knowledge representation as provided by CTP is the tool of choice.

A graphical user interface (GUI) is a feature of any contemporary software which seems not can be done without. CAS come along with a GUI (with exception of [H$^+$93] which has an emacs-interface). The majority of CTP (except [HKPM94] for instance) still do not have a GUI.

If an internet application is under consideration, the powerful java tool 'swing' is a good alternative.

Animation tools play an important role in contemporary educational software. There are crucial math matters to explain which are closely intertwined with basic data structures. For instance an explanation why the rule

$$c \neq 0 \Rightarrow \quad \frac{a \cdot c}{b \cdot c} \quad \rightarrow \quad \frac{a \cdot \not{c}}{b \cdot \not{c}} = \frac{a}{b}$$

[14] For the tutor unification (see Def.2.1.2) needs not to be implemented at at the first approach.

is applicable only to two of the following terms and one of them is wrong

$$\frac{2\cdot \not{3}}{4+5\cdot \not{3}} \quad \text{or} \quad \frac{2\cdot \not{3}}{4\cdot 5\cdot \not{3}} \quad \text{or} \quad \frac{2\cdot \not{3}}{(4+5)\cdot \not{3}}$$

cannot be given without access to the data structure of the formulae, and to the rewriter. This is a major restriction in the choice for animation tools.

Such tools are integrated in some of the CAS, but as it is necessary to access the rewriter, they become unusable together with their hosts. Fortunately there are generic tools like [HS93] or [Sta92] with reusable abstractions which have already been successfully exploited by [Aus00] for educational software in teaching computing.

Debuggers successfully realize the most crucial feature of the tutor specified: They allow to set break points at certain locations within a program, where the control is given to the user, and then the program can resume execution again [ASU86]. Just this is needed if, given a 'program' (their name will be 'script') describing an algorithm for solving a problem, at certain locations the control will be passed over to the student allowing him or her to decide on the next step.

However, there was no way to use a debugger, if the user is allowed to take another step as this one to be executed next in the program. The choice of several steps in parallel occurs in a very natural way during calculations, and a support of this feature is considered indispensable. Thus the idea of reuse a debugger for implementing this feature has been dropped.

The programming environment is the final point of consideration. It has become clear by the investigation on the related systems (p.17.ff) that a considerable amount of functionality must be added to either tool, and the check above reveals that a considerable portion of new code will be necessary.

Some CAS offer powerful new programming languages, e.g. Mathematica or Maple, which are appropriate for adding the new code required. Reduce is an open source system based on standard prolog [Hea69], an appropriate (and portable !) language for symbolic computation with many professional tools. CTP are based on different languages, one of which, ML [MTH90], has originally been designed for implementing a theorem prover.

1.2.6 The choice for prototyping

Having investigated various software-tools for appropriateness and usefulness, how and where to start implementation ?

The tutor as specified offers enough functionality to be a stand alone application, but a student may want to have the calculational power he is used to have from CAS in parallel, and all the graphing capabilities and features for manipulating tables as done in spreadsheets (and the logical

rigor this work wants to establish in order to guide the student accurately). Thus an integration with (an) other system(s) is highly desirable.

At the time being the only feasible decision is for a stand alone system. But which system, from the CAS-family or the CTP-family ? The investigation so far indicates that few choices are left; let us take the most appropriate representatives from both families and compare them in Tab.1.1.

Tab. 1.1: Comparison Reduce – Isabelle

feature	Reduce / Redlog	Isabelle
object language		
typed	\\	√
extensible	\\(programming)	√
stepwise calculation	√	√
sub term handling	\\	\\
calculatinal power	full fledged	poor
reals, complex numbers	√	\\
factoring, integration etc.	√	\\
term order adaptable	√	√
knowledge extensible	√	√
formally proven	partially	√
hierarchically displayed	\\	√
programming environment	Standard Lisp	SML
developer tools	√	\\
typed language	\\	√
graphical user interface	\\	\\

Reduce is distributed together with Redlog [DS96], a computer logic system implementing symbolic algorithms on first-order formulas w.r.t. temporarily fixed first-order languages and theories. Theories currently available are algebraically closed fields, real closed fields and discretely valued fields – i.e. the basic theories of high-school math are missing, and no effort is in sight to provide them.

The highly elaborated hierarchy of theories and the ongoing efforts to extend them is the compelling advantage of Isabelle [Pau97b]. The readability of Isabelle proofs and proof scripts is not satisfactory, as already mentioned. But recently Isabelle joined with an easy to use general proof environment ([BT98] and [BKSS97]) and there are ideas how to increase the readability of Isabelles proofs itself [Har96].

With this in mind, and in respect to the advantages of the strictly typed implementation language, SML, for rapid prototyping (and frequent changing) the decision is Isabelle.

Isabelle has, like Reduce, no graphic user interface. Thus there are plans

to construct a java-based GUI, and to make the tutors services available on the internet [Fin00b].

1.3 Scope and structure of the thesis

The thesis is in short:

> *A new kind of software capable of autonomously doing high-school math,* and *of guiding the students attempts as well, can be constructed with a reasonable effort.*

The novelty of such a software engine (called 'the tutor'), and the demand for it, has been under-pinned in the introductory sections. This has been done from the users point of view. The task of this thesis is to select and develop concepts and techniques necessary to construct such a tutor; it is a technical task. Thus a more technical formulation of the requirements has to be done, in order to limit the scope of the technical work, and to be able to check what actually has been achieved in the dissertation at the end. The following software specification [Maz91] is structured into four subsections, concerning the logic basis, autonomous problem solving, answering students questions, and reactive user-guidance.

1.3.1 Design of a logical framework for calculation

Mapping calculation to software must respect the essence of mathematics, which is formal deduction based on logic. That means, that each formula in a calculation must be justified by some (mechanically applicable) rule, which has been proven or given as an axiom somewhere. And the relation between the formulae of a calculation must be logically clear and evident; for instance, what is done by hand with placing some subsidiary calculations on an additional sheet of paper, or at the margin of the worksheet, needs to be modeled accordingly.

Check the correctness of a solution must be done mechanically as well. Thus the problem to be solved is to be formulated and presented in the system such that the user *and* the system work on the same representation, where each of the two partners have their own means to operate on that representation

The operations in a calculation w.r.t. such a representation should be balance controversial requirements: the student, in particular a beginner, should not be overwhelmed by technical details; the system, on the other hand, needs a lot of such details (as we will see in chapter 2) for working properly. This requires careful design of the semantics of the operations, the structure of the related commands, and the kind of their input.

1.3.2 Concepts for autonomous problem solving

Restriction to high-school math as is and to problems traditionally solved in education similarly all over the world, is an important limitation of this works scope. To be precise, 'problems in high-school math' will denote that part of mathematics in that kind of presentation, as given by the textbooks licensed for Austria, i.e. [S$^+$98a] to [S$^+$98b], [BDUS99a] to [BDUS99b], [N$^+$99a] to [N$^+$99b], [RMLH99a] to [RMLH99b], or [SUSD99a] to [SUSD99b] and the respective addenda for the solutions.

Autonomous problem solving should be done by a general engine doing calculations and inference. The mathematics knowledge required for autonomously doing math (i.e. without further input necessary from the student) must be separated from the math-engine, thus being extensible and reusable.

The mathematics knowledge base has to be prepared by mathematics-experts, course-designers and by teachers. This authoring-system must be separated from the tutoring-system for students. The students access to the knowledge must be guided and controlled.

All phases of problem solving should be supported by the system: the model-phase, the specify-phase and the solve-phase.

1.3.3 Reflection for answering students questions.

Reflection is a term coined by the developers of java, and denotes facilities of the software-system to inspect its own language-constructs.

Reflection instead of didactic intention should guide the design of answers requested by the student, this is the (novel ?) concept presented here. This concept means: on user request the system exhibits its knowledge and mechanisms in reasonable portions and steps towards more and more intimate internals. This requirement excludes didactic and psychologic considerations from the scope of the thesis; rather it coincides with the following:

As close to traditional math as possible the system should work. This requirements concerns the mechanisms the math-engine uses for inference (calculation of numerical constants is not considered interesting), as well as the representation of the knowledge.

1.3.4 Techniques for reactive user-guidance

A general mechanism for the dialog should provide for a seamless transition between several dialog modi, ranging from a demonstration by the system (where the student just watches the system in solving an example) to an exam-like mode (where the student has to provide the steps of solution, and the tutor restricts the students access to the knowledge).

The dialog modus can be preset by an author, a course-designer or by a teacher.

The student can change the dialog modus, and inspect the dialog state which is represented in 'human readable' way, too. The possibility to change the dialog modus can be restricted by the preset modus.

Resume guidance after user-input is the most important requirement. The student can select an arbitrary position in a calculation, and input either a rule to be applied at this position, or a formula deducible from the current one. The system must fit the input into the logical context, and then be able to continue with proposing the next step on demand of the student.

A general mechanism for the dialog should provide for an adequate rhythm in the flow of the interactions.

1.3.5 Survey on the chapters

Chapter 1 gives an introduction by starting from the authors background as a teacher, and surveys the situation with educational software for mathematics at high-schools as discussed in the literature of didactics of mathematics. An informal description of user requirements opens a bracket to be closed in chapter 4.

Chapter 2 formally circumscribes the scope of a mathematics engine capable of autonomously solving problems, and develops the underlying concepts: the hierarchy of problemtypes, the representation of calculations (calculational proofs), scripts describing methods for solving problems, and a script-interpreter providing for reactive user-guidance.

Chapter 3 describes the dialog-component which models the dialog between the user and the mathematics engine as partners on an equal base. The description is comparably informal, leaving space for the presentation

of a prototype system, featuring a subsequent estimation of effort necessary for future development.

Chapter 4 closes the bracket opened by stating the user requirements in chapter 1: After a statement on the key contributions of this thesis, a critical check of the user requirements is given, open questions are mentioned, and the effort for the next steps of development of the prototype is estimated.

Chapter 5 collects case studies of several kinds, two presenting material from early phases of this work, one surveys topics of high-school math, and one collects and completes examples from within the whole thesis.

1.4 Remarks on the notation

In addition to the conventional notation of mathematics we use the following notions.

- @ is the 'such' quantifier, $\exists!$ is unique existence
- $\equiv$ is associative equality on some meta-level, as opposed to $=$ as operator with result-type *bool* which is not associative in contrary to the former; used for the respective object-level
- $f\ x_1\ x_2 \cdots x_n$ is application of the function f to the arguments $x_1\ x_2 \cdots x_n$. Functions will be mapped over sets without explicitly mentioning it, i.e. for $x \in S$ a function may be applied to both, $f\ x$ and $f\ S$
- ϵ for 'empty', overloaded for arbitrary types
- _ for arbitrary, used in pattern matching

Sets and lists sometimes come close, in particular when approaching implementation.

- { } denotes sets, whereas [ ] denotes lists; sets often contain elements of different type, whereas lists contain (following SML traditions) elements of one and the same type
- the operator $\bullet$ is used as list-constructur, as well as operator for appending, as the difference is always clear from the context:

$$l = [2,3],\ \ 1 \bullet l = [1,2,3],\ \ l \bullet 4 = [2,3,4]$$

- for both, identifiers with initial capital letters are used, or with an uppercase calligraphic letter if capital letters are used for the respective elements already. Accordingly SML type-constants start with capital letters (which is not SML coding standard - see the stack example below)
- the function *map* is used for both, mapping over sets as well as over lists

SML syntax is used if necessary to go into implementation details; here is an example:

- ```
  type 'a Stack = 'a List;

  exception empty_stack;
  fun top ([]: 'a Stack) = raise empty_stack
    | top s = hd s;
  ```
  ```

```
fun pop ([]: 'a Stack) = raise empty_stack
  | pop s = (tl s): 'a Stack;
fun push e (s: 'a Stack) = (e::s: 'a Stack);
```

- in signatures $\times$ stands instead of $*$

Records are used instead of the product of sets, if precise notions for accessing elements are required.

- for instance instead of $\mathcal{N} \times \mathcal{R}$

$$Rec = \langle\ field_1\ of\ \mathcal{N},\ \ field_2\ of\ \mathcal{R}\ \rangle$$

  where $\langle\ \rangle$ are used because { } (SML notation) are reserved for sets

- individuals within records can easily be described element-wise (by using '.' as a selector) and as a whole:

$$r \in Rec,\ r = \langle 123, 4.5 \rangle,\ r.field_1 = 123,\ r.field_2 = 4.5$$

- as the naming of the fields is unique, one can describe a record by part of the fields:

$$r \in Rec,\ r = \langle field_2 = 4.5, \cdots \rangle$$

BNF, Backus normal form, follows these conventions:

- the symbols of the meta language are $::=\quad |\quad (\quad)\quad ^*\quad ^+\quad \epsilon$ as usual, whereas italic parenthesis *( )*, braces { } and brackets *[ ]* belong to the object language
- terminal symbols start with an upper-case letter (reminding of SML data-type constructors), non-terminals start with a lower-case letter
- the nonterminals *int* and *string* are as defined in SML (in particular: negative numbers are˜$1, \cdots$ and not $-1, \cdots$, strings are quoted by " "), *digit* has the obvious meaning
- inserted comments *(* ... *)* do not belong to the object-language.

Chapter 2

THE AUTONOMOUS MATHEMATICS-ENGINE

This main chapter develops the concepts of the mathematics-engine (ME). The ME is that component of the tutor which establishes a 'math expert' as an autonomous partner of the student in solving and discussing examples. 'Autonomous' means that, provided the related knowledge about domains, problem-types and methods, the ME is able to solve the example, and to check the students input for correctness w.r.t. the deductive system and purposefulness w.r.t. the method.

The MEs deductive ability is based on rewriting; this technique and related basic notions are presented in the first section.

The second section clarifies the boundaries of problems mechanized in the tutor, i.e. 'example construction problems', defines the notion of problem, and develops the concept of a hierarchy of problem-types; the mechanization of the latter is presented as original work.

The internal and external data representation of the deductive system, and the students operations on them while specifying and solving an example, are described in the second section.

The last section develops the source of the MEs autonomy in solving problems many of which are undecidable in general. A subset of Isabelle/HOL is used to describe algorithms, one of them solving all the examples of a problem-type. The interpreter of this language can be driven by two actors: by the methods' descriptions as well as by the student. A very general mechanisms allows to resume guidance by the method from the students' input. The latter is the main contribution of this thesis.

2.1 Basics: terms, parse-trees, rewriting and the math language

This section provides for the basic notions of symbolic computation. The notions presented are not only prerequisites to describe properties of formulae (called terms in this section) and to manipulate them, but they are also useful to describe proof-trees and parse-trees of (proof-) scripts.
Term rewriting is the work-horse of symbolic computation following highly developed techniques, which give clear advice which properties to observe. These properties are discussed w.r.t. the application of rewriting to tasks in high-school mathematics.
A brief statement on the choice of the object language describing mathematical objects within the tutor, concludes this section.

2.1.1 Basics, rewriting, and matching

The subsequent presentation of the material closely follows [NB98], restricted to the minimum necessary for the work to be done in the thesis. The most basic notions are just mentioned, for more specific notions the definition is given.

At the very beginning there is the notion of term, more precisely a $\Sigma - \textbf{term}\ t$, $t \in T(\Sigma, X)$ where Σ is a **signature** and X is a set of **variables**. $\mathcal{V}ar(t)$ is the set of variables in t, t is a **ground term** iff $\mathcal{V}ar(t) = \emptyset$.

In special contexts terms are called **directed acyclic graphs (DAG)**, **m-trees**, or **parse-trees** respectively. The structure of a DAG is useful for modeling different data, for instance Isabelles hierarchy of theories is a DAG.

The notion of **position** allows to address the parts of a term, and to describe the relations **parallel**, **above** and **below** on them. $\mathcal{P}os(t)$ is the set of position within term t. A position in a term can also be seen from a dynamic point of view as a **path** (from the root to the respective sub-term). The concept of position is useful not only for terms, but for the proof-tree and in particular for the parse-tree of the (proof-)script.

A **substitution** is a function $\sigma : X \to T(\Sigma, X)$ such that $\sigma(x) \neq x$ for only finitely many xs. Note that, in contrary to a **replacement**, a substitution σ *simultaneously* replaces *all* occurrences of variables by their σ-images. A substitution corresponds to β-conversion in λ-calculus. In a special context a substitution will be called **environment**. For those **function updating** is needed: for a function $f : A \to B$ we let $[a_0 \mapsto b_0]f$ be the function that acts like f except that it maps the specific value $a_0 \in A$ to $b_0 \in B$, i.e.:

$$\begin{aligned} ([a_0 \mapsto b_0]f)\ a_0 &= b_0 \\ ([a_0 \mapsto b_0]f)\ a &= f\ a \qquad \text{otherwise} \end{aligned}$$

Substitutions often are used in context with transformations of terms into 'equivalent' terms, where a set of rules justifies the equivalence; this involves the notions **identity**, **left-hand side (lhs)** and **right-hand side (rhs)**, **redex** is an instance of the *lhs*. **Contracting** the redex means replacing it with the corresponding instance of the *rhs*.

External and internal representation of terms is different. For our purpose of describing the mathematics-engine it suffices to identify the internal representation with the mathematical notion of terms as defined above. And it suffices to identify the external representation with strings.

Definition 2.1.1: Let *Str* be a set of strings, Σ_D the signature and V the set of variables ($\Sigma_D \cap V = \emptyset$) in some domain D. Then we define two functions
parse : $Str \times \Sigma_D \to T(\Sigma_D, V)$
pprint : $T(\Sigma_D, V) \to Str$
where $parse \circ pprint = pprint \circ parse = id$, the identity function.

The mathematical domain D may contain additional information about Σ like infix position and operator precedence. D may change during a session of the mathematics-engine.

Unification and matching
are both taken as black box from Isabelle. This even, while rewriting is re-implemented in order to be able to mark redexes. Thus a view 'from outside' is sufficient here. Simple unification and matching algorithms are exponential in space and time, both become (almost) linear by the use of sophisticated techniques [NB98].

Unification is the process of solving the satisfiability problem: given a set of equations E, terms s and t, find a substitution σ such that $\sigma s \approx_E \sigma t$. Unification is undecidable in general, but with $E = \emptyset$ it becomes decidable and then is called **syntactic unification**.

Definition 2.1.2: Let Σ a signature, V a countable infinite set of variables ($\Sigma \cap V = \emptyset$), $Bool = \{true, false\}$, and S the set of all substitutions on $T(\Sigma, V)$. Then we define the functions

- **Unify** : $T(\Sigma, V) \times T(\Sigma, V) \to S$
  where $Unify(s,t) = \sigma \iff \sigma s = \sigma t$.
- **Unifiable** : $T(\Sigma, V) \times T(\Sigma, V) \to Bool$
  where $Unifiable(s,t) \iff Unify(s,t) \neq \emptyset$.
- **Resolve** : $T(\Sigma, V) \times T(\Sigma, V) \to T(\Sigma, V)$
  where, given $s = ([\psi_1, \cdots, \psi_m] \Rightarrow \psi)$ and $t = ([\phi_1, \cdots, \phi_n] \Rightarrow \phi)$,
  $Resolve(s,t) = u \iff \exists \sigma \in S.\ \sigma\psi = \sigma\phi_j$ for some j, $1 \leq j \leq n$
  and $u = \sigma([\phi_1, \cdots, \phi_{j-1}, \psi_1, \cdots, \psi_m, \phi_{j+1}, \cdots, \phi_n]) \Rightarrow \phi)$

This definition subsumes high-order unification [Nip93], which is not decidable but has proven to be practically successful [Pau94].

Matching can be regarded as a special case of unification (given s, t, find σ with $\sigma s = \sigma t$), where all variables in t are taken as constants. Thus the definition looks rather similar, in spite the algorithms should differ for efficiency reasons.

Definition 2.1.3: Let Σ a the signature, V a countable infinite set of variables ($\Sigma \cap V = \emptyset$), E a set of rewrite rules, $Bool = \{true, false\}$, and S the set of all substitutions on $T(\Sigma, V)$. Then we define the functions

- **Match** : $T(\Sigma, V) \times T(\Sigma, V) \to S$
  where $Match(s,t) = \sigma \iff \sigma s = t$.

- **Matching** : $T(\Sigma, V) \times T(\Sigma, V) \to Bool$
  where $Matching(s,t) \iff Match(s,t) \neq \emptyset$.

- **Rewrite** : $T(\Sigma, V) \times T(\Sigma, V) \to T(\Sigma, V)$
  where, given $t = (l \to r)$,
  $Rewrite(s,t) = u \iff \exists \sigma \in S, p \in \mathcal{P}os(s).\ s|_p = \sigma(l)$ and $u = s[\sigma(r)]_p$
  where $s|_p$ is the sub-term s at position p, and $u = s[\sigma(r)]_p$ is replacing the sub-term at position p by the right-hand side of t.

Rewriting is, as compared to resolution, the simpler concept and it suffices for the tutor. Unification involves equation systems, whereas matching even can be demonstrated by animation.

2.1.2 Rewriting and its application

Rewriting, termination and confluence

These two questions motivate the definitions given in the sequel:

1. Transformation of theorems or terms by a given set of rules: Can we expect that applying the rules of a set stops, i.e. after a finite number of steps none of the rules is applicable any more ? This is the notion of *termination*.

2. Testing terms for equivalence: For instance given the terms $x + x - 1$ and $3 + 2x - 4$ and the rules for handling integers, can we apply these rules (in arbitrary order ?) to the terms until termination, and then expect them to be litterally equal ? This leads to the notion of *confluence*.

Termination is related to some notion of making terms simpler or smaller. Thus we have to establish orders on terms. This involves the notions **quasi-order**, **partial order**, **strict order**, and **total order**. The **lexicographic order** is particularly useful.

The orders used for terms are recursive path orders. Besides the **multiset path order**, the following order is used, and *the recursive path order with status* which combines these approaches [NB98].

Definition 2.1.4: Let Σ be a finite signature and $>$ be a strict order on Σ. The **lexicographic path order** $>_{lpo}$ on $T(\Sigma, V)$ induced by $>$ is defined as follows: $s >_{lpo} t$ iff
(LPO1) $t \in \mathcal{V}ar(s)$ and $s \neq t$, or
(LPO2) $s = f(s_1, \cdots, s_m), t = g(t_1, \cdots, t_n)$, and
(LPO2a) there exists $i, 1 \leq i \leq m$, with $s_i \geq_{lpo} t$, or
(LPO2b) $f > g$ and $s >_{lpo} t_j$ for all $j, 1 \leq j \leq n$, or
(LPO2c) $f = g, s >_{lpo} t_j$ for all $j, 1 \leq j \leq n$, and there exists $i, 1 \leq i \leq m$, such that $s_1 = t_1, \cdots, s_{i-1} = t_{i-1}$ and $s_i >_{lpo} t_i$.

This definition leaves the choice for the basic order $>$ on Σ open; the lexicographic would be appropriate. If an order should be useful for proving termination in rewriting, it must meet general requirements preserving properties of terms under rewriting: The weakest kind is the **reduction-order** compatible with Σ-operations, followed by the **rewrite-order** closed under substitutions, and then the **simplification order** satisfying the sub-term property.

Now the preparations are done to introduce the notion of rewriting (simplification) and the respective properties: An **abstract reduction system** is a pair $(A, \rightarrow)$, where the **reduction** $\rightarrow$ is a binary relation on the set A, i.e. $\rightarrow \subseteq A \times A$. Instead of $(a, b) \in \rightarrow$ we write $a \rightarrow b$.

A reduction $\rightarrow$ is called **confluent** iff $y_1 \ {}^*\!\leftarrow x \rightarrow^* y_2 \Rightarrow y_1 \downarrow y_2$ (where $y_1 \downarrow y_2$ iff there is a z such that $x \rightarrow^* z \ {}^*\!\leftarrow$, and $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$), **terminating** iff there is no infinite descending chain $a_0 \rightarrow a_1 \rightarrow \cdots$, **convergent** iff it is both confluent and terminating.

A **rewrite rule** is an identity $l \approx r$ directed from left to right $l \rightarrow r$, and such that l is not a variable and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$.
A **term rewriting system** is a set of rewrite rules, and thus often called a **rule-set**.

Finally, how can we prove a term rewriting system to be terminating ? There are three different methods for solving the problem, which is undecidable in general. The first method has been prepared by the previous definitions of order, it is most important because it can be fully automated.

1. **Use reduction orders** for checking $l > r$ for all rewrite rules $l \rightarrow_R r \in R$. If this check results in true we can imply termination of R

only, if $>$ is a well-founded rewrite order on $T(\Sigma, V)$. Lexicographic path orders are simplification orders, and these are reduction orders [NB98]. Kunth-Bendix-orders have the same property.

2. **The interpretation method** defines a reduction order on $T(\Sigma, V)$ by considering the terms' interpretation in a Σ-algebra that is equipped with a well-founded order, instead of considering the terms over Σ themselves, (as done in 1.).

3. **The inverse image construction** maps the $(A, \rightarrow)$ under consideration into another reduction system $(B, \rightarrow)$ which is know to terminate. The mapping is called the *measure function* and often takes the naturals $\mathcal{N}$ as its range.

Confluence concerns the other basic problem: Given a term rewriting system R and a term t, how can we be sure that arbitrary rules $r_i \in R$ applied in arbitrary chains $t \rightarrow_{r_{i_1}} t_{i_1} \rightarrow_{r_{i_2}} t_{i_2} \rightarrow \cdots t_1$ or $t \rightarrow_{r_{j_1}} t_{j_1} \rightarrow_{r_{j_2}} t_{j_2} \rightarrow \cdots t_2$ would yield the same result $t_1 = t_2$ upon termination ? This problem is undecidable in general, as the two forks $t \rightarrow \cdots$ of arbitrary length may indicate. However, it is decidable for terminating finite term rewriting systems. The reason is that for those systems confluence is equivalent with *local* confluence (defined as $y_1 \leftarrow x \rightarrow y_2 \Rightarrow y_1 \downarrow y_2$). The local study on confluence may detect multiple redexes in a term which interfere with one another, called **critical pairs**.
The critical pairs (p, q) identified within a term rewriting system R can be rewritten into a normal form $(\hat{p}, \hat{q})$. If $\hat{p} \neq \hat{q}$ we can create a new rewrite rule, either $\hat{p} \rightarrow \hat{q}$ or $\hat{q} \rightarrow \hat{p}$, such that the left-hand side is greater than the right-hand side w.r.t. an appropriate reduction order. This new rule and all further ones can be added to R until, hopefully, 'saturation' will be reached. This is a rough description of the so-called **completion procedure**. An algorithm has fist been given by [KB70]. The procedure has been generalized in several ways [NB98], and improved for efficient implementations.

An important example are the axioms of a group as (bidirectional) equations

(1) $1 \cdot x = x$ (2) $x^{-1} \cdot x = 1$ (3) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

which, interpreted as (unidirectional) rewrite rules, result in the following term rewriting system (rule set) after completion:

(1) $1 \cdot x \rightarrow x$ (2) $x^{-1} \cdot x \rightarrow 1$ (3) $(x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z)$
(4) $1^{-1} \rightarrow 1$ (5) $x^{-1} \cdot (x \cdot y) \rightarrow y$ (6) $x \cdot 1 \rightarrow x$
(7) $(x^{-1})^{-1} \rightarrow x$ (8) $x \cdot x^{-1} \rightarrow 1$ (9) $x \cdot (x^{-1} \cdot y) \rightarrow y$
(10) $(x \cdot y)^{-1} \rightarrow y^{-1} \cdot x^{-1}$

This is a modest explosion in the number of rules for the central part of elementary algebra at high-school, i.e. the groups w.r.t. addition in the ring $\mathcal{Z}$, and the groups w.r.t addition, multiplication and power in the fields $\mathcal{Q}$ and $\mathcal{R}$. Because of the great advantages of complete rule sets (see Def.2.1.5 below) all rule sets should be completed if possible. The only exceptions occur, if one tries to tackle examples like the following by rewriting (which is questionable), where the method of 'completing a square' is used to solve an equation:

$$\begin{aligned}
x+6x+8 &= 0 \\
x+6x &= -8 \\
x+2\cdot 3x &= -8 \\
x+2\cdot 3x+3\cdot 3 &= -8+3\cdot 3 \\
(x+3)^2 &= 1 \\
x+3 &= +1 \quad \vee \quad x+3 = -1 \\
x &= -2 \quad \vee \quad x = -4
\end{aligned}$$

This kind of calculation depends on a sequence of rewritings, which may *not* be rearranged.

Confluence and termination together constitute a very powerful tool which is worth to have its own name.

Definition 2.1.5: A terminating and confluent term rewriting system is called a **canonical simplifier**. A normal form of a canonical simplifier is called a **canonical form** (which is unique).

For most of the calculations in high-school mathematics a canonical simplifier can be found, including calculations in the ring $\mathcal{Z}$ of integers, the fields $\mathcal{Q}$ of rationals, $\mathcal{R}$ of reals and $\mathcal{C}$ of complex numbers, also for differentiation, manipulating logarithms etc. An enumeration of canonical simplifiers, exhaustive w.r.t. the high-school syllabus, will be given in the case study 5.3.

Canonical simplifiers also allow for solving the so-called word problem, which is undecidable in general.

Definition 2.1.6: The **word problem** for a set of identities E is the problem of deciding $s \approx_E t$ for arbitrary $s, t \in T(\Sigma, V)$. The **ground word problem** for E is the word problem restricted to ground terms s and t.

This notion captures the situation already mentioned as challenging in the construction of the tutor: given a calculation until an intermediate formula f the student inputs a formula f' meant as a 'correct' continuation to the calculation. If the calculation is done by a canonical simplifier, the question

is decidable by rewriting $f \to^* n$ and $f' \to^* n'$ and comparing $n = n'$. Very often, however, more than one rule set is involved, which complicates the situation. This will be demonstrated in an example on p.39 and thoroughly discussed in 2.4.6.

Rewriting as discussed so far still lacks some concepts in order to successfully calculate some of the high-school examples. Let be given the equation (this example will come several times)

$$9 + 4x = 2x + 2\sqrt{5x + x^2} + 5 \tag{1}$$

The students are taught to isolate the root by $(a = b + c\sqrt{d}) \to (a - b = c\sqrt{d})$ and then to square the equation by $a > 0 \wedge b > 0 \Rightarrow (a = b) \to (a^2 = b^2)$

$$9 + 4x = 2\sqrt{5x + x^2} + 5 + 2x \tag{2}$$
$$(a = b + c\sqrt{d}) \to (a - b = c\sqrt{d})$$
$$9 + 4x - (5 + 2x) = 2\sqrt{5x + x^2} \tag{3}$$
$$a > 0 \wedge b > 0 \Rightarrow (a = b) \to (a^2 = b^2)$$
$$(9 + 4x - (5 + 2x))^2 = (2\sqrt{5x + x^2})^2 \tag{4}$$

which finally will cause the root to disappear. These few rewritings show the necessity of two concepts not discussed so far, ordered rewriting and conditional rewriting.

Ordered rewriting is necessary for the step from (1) to (2). The basic rewriting framework is unable to deal with commutative and associative operators ! For instance the rule $b + a = a + b$ cannot be oriented to form a *terminating* rule set. One method to cope with rules like this is to shift the termination proof from 'compile time' to 'run time' [NB98]. Instead of proving termination of $\to_R$ once and for all by showing that R is contained in a reduction order $\succ$, termination is enforced at each rewrite step by admitting rewriting only when it decreases the term w.r.t. $\succ$. For instance $a + b \to b + a$ is being applied at $y + x \to x + y$, but not at $x + y$ provided $x \succ y$. This concept together with the rule set $\{y+x \to x+y,\ (x+y)+z \to x+(y+z),\ x+(y+z) \to y+(x+z)\}$, a so-called AC-operator, may order the sub-terms by a kind of bubble-sort. By this technique the $\sqrt{5x + x^2}$ can be shifted out from the terms middle in step (1) to (2) above yielding

$$9 + 4x = 2\sqrt{5x + x^2} + (5 + 2x)$$

It is the task of the simplification order $\succ$ to shift the $\sqrt{5x + x^2}$ to the left side, and not to the right side. At the right side the root would be buried within the parentheses and the rule $(a = b + c\sqrt{d}) \to (a - b = c\sqrt{d})$ would not apply.

Conditional rewriting concerns rules like the one rewriting from step (3) to (4) above. Also such important canonical forms as the polynomial form need conditional rewriting as soon numerical constants are involved. The rule $a(b+c) \to ab+ac$ is the most important to calculate the polynomial form. But for simplifying the term

$$6 + 5x + 4x + 3x^2$$

to polynomial form the distributive law needs to be directed vice versa. Thus termination would be corrupted if not reestablished by posting a guard

$$is\_numeral(a) \wedge is\_numeral(b) \Rightarrow a \cdot c + b \cdot c \to (a+b) \cdot c$$

and then ensuring, that the term $a+b$ is computed to *one* numeral, before the distributive law comes into play again.
Conditional rewriting, unfortunately, differs from its unconditional relative in many important aspects [NB98]. Thus implementations within this topic will be done rather by using practical experience gained with algebra systems than by further developing the the theory.

Combination of rule sets is frequently necessary, but usually cuts down the properties of termination and confluence. Given a property P over rule sets R_1, R_2 with the respective signatures Σ_1, Σ_2, the **combination problem** concerns the question for which conditions

$$P(R_1 \cup R_2) \Leftrightarrow P(R_1) \wedge P(R_2) \tag{2.1}$$

may hold. The most interesting properties P are termination and confluence. [NB98] shows that 2.1 holds for termination with $\Sigma_1 \cap \Sigma_2 = \emptyset$ (R_1 and R_2 are **disjoint**); for confluence there are larger subclasses (orthogonal rule sets).

However, combinations are *not* disjoint in high-school examples (in general *all* rule sets contain $+ \cdot$ together), thus the conditions for termination are lost in those examples, and consequently confluence does not practically help either. In order to tackle the main issue in constructing the tutor, i.e. solving the word-problem for a formula input by the student, also in the case of combination of rule sets, other techniques are necessary. Let us look to a typical example from a textbook: make m_2 explicit in the equality *equ* (a law from electro-engineering), where students are taught to follow the steps *factorize, multiply-denominators, expand, variable-to-left, factor-out, isolate-variable*, all of which can be modeled by canonical simplifiers:

$$E = \frac{m_1 m_2 v_1^2}{2(m_1+m_2)} + \frac{m_1 m_2 v_2^2}{2(m_1+m_2)}$$

ruleset *factorize*

$$E = \frac{m_1 m_2 (v_1^2 - v_2^2)}{2(m_1+m_2)} \qquad \neg Match(equ, a+b=c) \wedge \neg Match(equ, a=b+c)$$

ruleset *mulitply-denominators*

$$2E(m_1+m_2)=m_1m_2(v_1^2-v_2^2) \qquad \neg Match(equ,\tfrac{a}{b})$$
ruleset *expand*
$$2Em_1+2Em_2=m_1m_2v_1^2-m_1m_2v_2^2 \qquad IsPoly(Lhs\ equ)\wedge IsPoly(Rhs\ equ)$$
ruleset *variable-to-left*
$$2Em_1+2Em_2-m_1m_2v_1^2-m_1m_2v_2^2=0 \qquad m_2\notin(Rhs\ equ)$$
ruleset *factor-out*
$$2Em_1+m_2(2E-m_1v_1^2-m_1v_2^2)=0 \qquad Occurs\ m_2=1$$
ruleset *isolate-variable*
$$m_2=-\tfrac{2Em_1}{2E-m_1v_1^2-m_1v_2^2}$$

where *IsPoly, Occurs*, etc. are functions on terms as can be found for instance in Mathematica.
At the right margin above there are conditions P found true after termination of the rule set R_i (P is the postcondition of R_i) and before rewriting with R_{i+1} (P is the precondition of R_{i+1}). These pre- and postconditions of the rule sets provide for information helpful in solving the word problem introduced in 2.4.6.

Out of the scope of rewriting are some elementary calculations, which cannot be dropped in high-school mathematics. But a kind of 'reverse rewriting' allows to fetch these calculations back into the realm of rewriting. The term

$$2x^3+4x^5$$

cannot be factorized by rewriting ! The obstacle are not the powers (which easily can be made products by rewriting), but the numeral constants: a rewriter never can find out, that $4=2\cdot 2$. The 'reverse rewriting' needs the help of a *hidden* [1] CAS function capable of factorization, returning

$$2xxx(1+2xx)$$

A quick postprocessing reversely multiplies all factors but one, and presents the rule

$$2x^3+4x^5\rightarrow 2(x^3+2x^5)$$

In the same line the calculation of numeric constants is done by socalled 'proforma theorems' like in

$$2-3+4x^5-6 \ \xrightarrow[2-3\rightarrow -1]{}\ -1+4x^5-6$$

This technique is simliar to a proposal of [Har97] to combine the calculational power of CAS with the logical rigor of CTP in integration: call the CAS as an 'oracle' for providing the integral, and then employ the CTP to check the integrals correctness by 'reversely' differentiating.

[1] Hiding the high-brow techniques of factorization is completely according to the way the task is taught in schools: factors of polynomials (which of course are restricted to simple ones in the exercises) are found by 'intuition', and predecessors of factorization algorithms are only mentioned as an aside when solving equations.

2.1.3 The mathematical object language

What kind of language describing math objects is appropriate for high-schools, first-order predicate calculus or high-order, with or without extensionality, etc.? And above all, are the notions developed above really a good formal base for elementary mathematics ?

Many-sorted, typed terms extending the notion of a sigma-term $t \in T(\Sigma, X)$ (Def.2.1.1) introduced so far are indispensable. The comparison of the different algebraic laws for the different domains of natural numbers $\mathcal{N}$, integers $\mathcal{I}$, rationals $\mathcal{Q}$ etc. is an important means to foster the idea of algebraic structures in the students comprehension. The different domains of numbers in highschool-mathematics will be considered as *different* sorts, e.g. $\mathcal{N}$ is not a subsort of $\mathcal{I}$. Thus *all* the results and definitions of section 2.1 carry over ([NB98] pp.35), luckily.

It suffices to regard Σ beeing associated with **types** in addition to arities. As an example, exponentiation may have the arity 2 and the (typed) signature $\mathcal{R} \times \mathcal{N} \to \mathcal{R}$. In order to allow for simply describing manipulations on terms (like compose and decompose subterms) we do not distinguish between the type 'term' and the type of the terms 'value', i.e. $2.0^3 \in \mathcal{R}$ whether evaluated to 8.0 or not. Thus, given two terms t_1, t_2, the function $Typeq : T \times T \to bool$ tests for equality of the result-types only. [2]

A first-order or a high-order language for math at high-schools ? There are two reasons which indicate for a first-order language:

1. First order calculi are used in highschool mathematics presently.

2. First order unificiation is decidable, high order unification is not.

Both reasons, however, are questionable.

1. The mathematics stuff taught in highschools necessarily limps behind the front of the wave in science. Thus some look ahead is advisable.

   At the rise of modern mathematics a more or less informal version of high order logic was used. About hundred years ago several paradoxials and contradictions seemed to endanger the foundations of mathematics. In a two or three decades effort the formal base has been clarified, the trust in the foundations of mathematics was reestablished. Some of the difficulties encountered stem from self-referential constructs; high-order constructs may seem similiar. Whatever the

[2] The tutor will rely on Isabelles knowledge, and also on its basic datastructures. In the case of terms this is simply typed λ-calculus [NPS90], which serves as a model for implementing terms in the tutor.

reasons were, Bourbaki based their great foundamental work on first-order logic. And now, at the end of the century, Bourbakis-formalisms (which are based on first order logic) have found its way into high-schools. However, already in the thirties of the century, high order logic has got a powerful and elegant foundation by the invention of the λ-calculus. Very soon the λ-calculus became the theoretical basis of functional languages. And there are recent academic announcements considering high order logic to be more natural. Now, how long would it take, until the discussion carries over to highschool syllabi ?

2. There are computer based logic frameworks which successfully use λ-calculus as abstract syntax, e.g. [Pau94]. Even Mathematica employs a rewriting engine which may called high-order [Buc96b] because the related language is capable of currying.

So it might be wise to build the first order language of a tutoring system upon a high order abstract syntax in order to be open for further developments. The tutor employs, following Isabelle, high-order predicate logic with equations and extensionality, uses the theories derived in Isabelle so far (up to the reals at the time of writing this thesis), and trusts in further development of theories on calculus and other high-school stuff not yet covered by the present release of Isabelle.

A more technical question arises from a very fundamental syntactic ambiguity in traditional mathematics notation:

$$h(a+1)$$

is this a function application or a product of terms, i.e.

$$(\lambda x.\, t)(a+1) \text{ where } h = \lambda x.\, t \qquad \text{or} \qquad h \cdot (a+1)$$

CAS prefer to maintain the traditional notation for multiplication dropping the $*$, and to distinguish function application by another *non*-traditional notation (e.g. Mathematica by using the brackets $h[a+1]$ and thus missing the usual notation for lists, too). CTP in contrary, and in particular Isabelle, denote multiplication by mandatory $*$ and leave function application as is. The tutor will adhere to the latter choice.

2.2 Problem-types for mechanized problem solving

The first goal in constructing the tutor is to restrict mathematical problem solving in a way that allows to automatically generate a solution. This raises two issues: (1) formalize what a problem is, and (2) formalize mathematical knowledge to serve mechanical problem solving.
There are different types of problems, differently accessible by mechanical treatment. One of the types is exercised predominantly in high-school mathematics; this one is shown adequate for a mechanical approach.
The notion of 'problem' is formally defined, the naming of objects is clarified, and two kinds of relations between problems are discussed: the relation 'problem P_1 is a special case of P_2', and the relation 'P_1 uses P_2 as a subproblem in method M'.
The case study 5.2 discusses their application to equation solving in detail.

In this section Greek letters denote predicates, capital letters denote sets, and lower-case letters denote variables.

2.2.1 General classes of problems

The notion of 'problem' characterizes a dynamic view; problem solving aims at creating new knowledge. The realm of mathematics knowledge can be accessed by different kinds of questions. [Buc94] suggests five classes of problems in mathematics. These will be discussed and illustrated with examples from high-school.

Universal knowledge problems have the form

> *Prove* that
> for all x with $D\,x$ we have $\eta\,x$

where x denotes a vector of variables, $D\,x$ the 'domain' of the objects under consideration and $\eta\,x$ some formula of predicate logic with variable x. An example for a universal knowledge problem for equations is the existence of solutions, for instance:

$$\begin{array}{l}\text{for all polynomials } \sum_{i=0}^{n} a_i x^i \text{ with } a_n \neq 0, a_0 \neq 0 \\ \textit{prove that } \exists x \in I.\ \sum_{i=0}^{n} a_i x^i = 0\end{array} \tag{1}$$

where I is the domain of integers, i.e. this problem looks for solutions in the domain of integers I only. The equation $\sum_{i=0}^{n} a_i x^i = 0$ demonstrates the peculiarity of the restriction to I: whereas the equation is not trivial at all

in the domain of reals, it is simple in I. Isabelle's distribution [3] provides an abundance of knowledge necessary to give answers to the problem above:

```
dvd_def        m dvd n == EX k.  n = m*k
dvd_0_right    m dvd 0
dvd_0_left     0 dvd m ==> m = 0
dvd_1_left     1 dvd k
dvd_refl       m dvd m
dvd_trans      [| m dvd n; n dvd p |] ==> m dvd p
dvd_imp_le     [| k dvd n; 0 < n |] ==> k <= n
```

The definition and many theorems (we only show the head of the list) are contained in theory 'Divides.thy'. Now, looking at the hundred theorems, how could one find out the few theorems which are useful to give an answer to the knowledge problem above ? Isabelle provides excellent features to search in its knowledge base. Just as an example we show the hierarchy [4] of theories relevant for our topic:

```
Primes
  \__Divides
  |     \__Arith
  |          \__Nat
  |               \__NatDef
  |                    \__WF (* well-founded recursion *)
  |                         \__Trancl (* refl.trans. closure of relations *)
  |                              \__Lfp (* Knaster-Tarksi theorem *)
  |                              |    \__mono
  |                              |    |     \__equalities
  |                              |    |          \__subset
  |                              |    |               \__Fun
  |                              |    |                    \__Set
  |                              |    |                         \__Ord
  |                              |    |                              \__HOL
  |                              |    |                                   \__CPure
  |                              |    \__Prod      (* products of sets *)
  |                              |         \__Fun...
  |                              \__Relation
  |                                   \__Prod...
  \__WF_Rel
       \__Finite
            \__Divides...
```

It is obvious, that even excellent features like those of Isabelle support very little in mechanically finding answers to universal knowledge problems. A given theorem may be proved by Isabelle. But how to guide the search for the proof ? And which structures lead to new definitions and the formulation of new knowledge ? There are no serious answers relevant for constructing

[3] Isabelle in the release 1996.

[4] actually a DAG, a directed acyclic graph

the tutor. The difficulty of mechanically solving universal knowledge problems corresponds to their treatment at high-school: this kind of knowledge is to be learned by the students, and afterwards applied to (another class of) problems, rather than to be constructed and proven by students.

Decision problems are different to the previous problems in that they are geared towards computing solutions:

> *Find an algorithm* $\mathcal{A}$ such that
> for all x with $D\,x$
> if $\eta\,x$ then $\mathcal{A}\,x$ else $\neg\mathcal{A}\,x$

where $D\,x$ again is the domain, in this special case called the precondition (input specification) of A whereas $\eta\,x$ is the 'output specification'.
To this problem class the same observations wrt. mechanic solutions and treatment in high-school apply as for the first problem class: students prefer to get more 'practical results' than a 'true' or 'false'. For checking user input, however, such decision problems need to be solved.

Example (1) for the universal knowledge problem about the solvability of polynomial equations over I leads to a trivial decision procedure: systematically trying all the numbers for the variables ocurring in a predicate gives a semi-decision procedure, and if there are bounds, we already have a decision procedure. For instance, if we apply the theorem `dvd_imp_le` to the definition for 'divides', `dvd_def`, we have this kind of 'decision procedure' of enumeration in `divisors_alg`:

```
dvd_def       m dvd n == EX k.  n = m*k
dvd_imp_le    [| k dvd n; 0 < n |] ==> k <= n
divisors_def  divisors(n) == {k.  k dvd n}
divisors_alg  divisors(n) == {k.  k <= n} ∩ {k.  k dvd n}
```

By the algorithm `divisors_alg` the set *divisors(n)* can be computed. We will need `divisors(n)` in the feasibility studies.

Explicit computation problems have the form

> *Find an algorithm* $\mathcal{A}$ such that
> for all x with $D\,x$ we have $P\,x\,(\mathcal{A}\,x)$

where $D\,x$ is the 'input specification' of $\mathcal{A}$, $\mathcal{A}\,x = y$ denotes the result of applying $\mathcal{A}$ to x, and $P\,x\,y$ is the 'post condition', the relation between input and output. If we set $P\,x\,(\mathcal{A}\,x) = (\text{ if } P\,x \text{ then } \mathcal{A}\,x \text{ else } \neg\mathcal{A}\,x\ )$ then we see that a decision problem is a special case of an explicit computation problem.

Let us give an answer to example (1) for universal knowledge problems in the form of an explicit computation problem:

Find an algorithm $\mathcal{A}$ such that
for all polynomials $\sum_{i=0}^{n} a_i x^i \in \mathcal{I}[x]$ with $a_n \neq 0, a_0 \neq 0$
we have $@\, L_{\mathcal{A}} \subseteq \mathcal{I}.\ L_{\mathcal{A}} = \{x.\ \sum_{i=0}^{n} a_i x^i = 0\}$

There is an algorithmic solution to this problem:

$$\{x.\ \textstyle\sum_{i=0}^{n} a_i x^i = 0\} \ = \ \mathit{divisors}(a_0) \ \cap \ \{x. \textstyle\sum_{i=0}^{n} a_i x^i = 0\}$$

i.e. we can compute the solutions for this class of Diophantine equations. If we take into account, that the set $\mathit{divisors}(a_0)$ is finite, we only need to take all elements of $\mathit{divisors}(a_0)$ and try whether they belong to the intersection, i.e. whether they solve the equation and are elements of $\{x. \sum_{i=0}^{n} a_i x^i = 0\}$.

Example construction problems are the class of problems concrete enough for providing support to solve them by software systems:

Find y *such that*
$P\ y$

The predicates Dx specifying the domain are missing, because the vector of input variables is empty, i.e. only constants are given. [5] Such a problem is solved by determining values which fulfill the postcondition P.

The example construction problems make up the largest part in traditional elementary mathematics education. Students are taught a certain collection of algorithms defined by the syllabus; 'solving a problem' then is the students task to select the appropriate algorithm and to 'construct an example' by applying the algorithm to the constants given.

Example construction problems are most concrete in the sense that their solution are first order objects, and not second order objects like algorithms. Thus, the problem solving process can be guided by considerations 'one level lower' than by the other problem classes:

$$\begin{aligned}\{x.\ x^3 + x + 2 = 0\} &= \mathit{divisors}(2) \ \cap \ \{x.\ x^3 + x + 2 = 0\} \\ &= \{-1, 1, -2, 2\} \ \cap \ \{x.\ x^3 + x + 2 = 0\} \\ &= \{-1, 1, -2, 2\} \ \cap \ \{-1\} \\ &= \{-1\}\end{aligned}$$

Another example of the class 'example constructing problems' would be solving the equation

$$1 + 2x + 3(4 + 5x) = 6 - 7x$$

The *search* for algorithms solving this type of equation, however, leads to the last problem class.

[5] Buchberger [Buc94] calls this situation 'throw-away' algorithms, which are used only once for a particular set of given constants.

Implicit computation problems These problems have the form

Find algorithms $A_1, ...A_k$ such that P

where P is a formula involving terms $A_1t_1, ..., A_kt_k$ where the $t_i \in T$ are terms at arbitrary positions and $T = T(\Sigma, V)$ as defined in 2.1.1.

The difference to explicit computation problem is: the latter can be written like an 'explicit definition' (i.e. with the definiendum on the lefthand side, and the definiens on the righthand side of an equality) in a natural way, whereas with implicit computation problems this is not the case. For instance searching for algorithms to solve $1 + 2x + 3(4 + 5x) = 6 - 7x$ would lead to the implicit computation problem connected with confluence, termination, normal forms etc., which have been considered in the previous section:

Find an algorithm $\mathcal{S}$ (a 'canonical simplifier')
such that $\forall A, s, t$, where
A is a set of equational axioms and
s and t are expressions
the following two properties are satisfied:
$s = \mathcal{S}(s)$ can be derived from A and
if $s = t$ can be derived from A then $\mathcal{S}(s) = \mathcal{S}(t)$

The class of implicit computation problems may be considered as the most abstract one among the classes mentioned: It can be viewed as an example construction problem in the sense that the task consists in finding 'examples of algorithms' that satisfy P, i.e. this problem is located 'one level higher'.

In [Buc94] one can find a wealth of considerations about 'problem context analysis' and the interdependence of problem types.

2.2.2 Modeling example construction problems

The class of example construction problems, being predominant in traditional high-school mathematics, is the one supported by the tutor. Modeling is discussed at two levels here: (1) there will be definitions of a formal model, which gives the framework for implementing a tutor, which guides the student in (2) modeling and specifying mathematics examples.

An example will serve to clarify both levels: it will motivate the definitions, and it will illustrate the process of modeling done by the student. This example can be found in many textbooks on calculus together with a lot of similar ones, for instance in [GHHK77] or the introductory example at p.11:

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

This example will bei referred to by 'the maximum-example' in the sequel.

The goal is to have the notion of a 'problemtype', which the maximum-example is an instance of, where the particular items of the special example instantiate this problemtype producing a problem.
Let us begin with the description of the particular items of an example, which can be of various kind: variables with and without a value, equalities, and a variable denoting the bound variable to make up an equation together with an equality, sets, lists, etc (all of which we call *items*; these at least are terms, and thus the function *Vars* can be applied; similarly simplifying, we denote the values of items i by $Val\, i$):

Definition 2.2.1: Given a set I of **input-items**, a set $O \neq \emptyset$ of **output-variables**, and a set $R \subset P$ of **relations**, the triple $F = (I, O, R)$ is a **formalization** iff $(Vars\, I) \cap O = \emptyset$.

where P is a set of predicates. A formalization of the maximum-example as discussed so far is

$$\begin{array}{ll} F \equiv (I, O, R), \text{ where} & \\ I & \equiv \{\, \{r = 7\}\, \} \\ O & \equiv \{\, A,\ \{a, b\}\, \} \\ R & \equiv \{\, \{A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\}\, \} \end{array}$$

Such a formalization should provide the mathematics-engine with all *particular* information necessary to automatically solve a *particular* example (not to speak of the knowledge about a *whole type* of problem), *and* such a formalization should provide the information to guide the user in modeling and specifying the example. The list for the maximum-example above contains very little information for those two tasks. How to accomplish these tasks will be considered after the central notions have been introduced.

The definition of example construction problems first requires clarification of some fundamentals. Let us again take the maximum-example to illustrate them. The maximum-examples characteristics captured by a postcondition may look like

$$\begin{array}{l} \exists A.\ A = a \cdot b \wedge (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2 \wedge \\ \quad \forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\frac{a'}{2})^2 + (\frac{b'}{2})^2 = r^2 \Longrightarrow A' \leq A \end{array} \qquad (2)$$

where r is regarded as quantified by $\forall$. This postcondition states the existence of such an A without demand of calculating it. It states, that given some input I the solution of the problem has to provide an output O which meets some predicate $\rho(I, O)$. Now, with respect to description (1) one may ask, what the input and the output are in the example given by the description on p.47. Usually one is not interested in the value of the area A, but in

a and b (and eventually their ratio). Then a and b are requested as output, and may be regarded as bound by @ on the meta-level, giving

$$@a\ b\ A.\ A = a \cdot b \wedge (\tfrac{a}{2})^2 + (\tfrac{b}{2})^2 = r^2 \wedge \quad (1)$$
$$\forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\tfrac{a'}{2})^2 + (\tfrac{b'}{2})^2 = r^2 \Longrightarrow A' \leq A$$

The difference between (1) and (2) makes clear, that the description of a problem must explicitly mention input and output.

The descriptions (1) and (2) also show, that correct formulations of high-school problems are not trivial. Students cannot be expected to input them. But the relations $A = a \cdot b$ and $(\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2$ are essential for the example and distinguish it from others. Consequently their treatment is divided into two parts: (a) into a part to be input by the student, but without quantifiers and other complicated formalisms of predicate calculus, and (b) the exact formulation, however computed by the system. This gives:

Definition 2.2.2: Given the sets I of items and O of variables with $(\mathit{Vars}\ I) \cap O = \emptyset$, and R of predicates, the predicates $\eta(\mathit{Vars}\ I)$ and $\rho(\mathit{Vars}\ I, \mathit{Vars}\ O, \mathit{Vars} R)$, with ρ quantifying all free variables by @, 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**, η is the **pre-condition** and the predicate ρ is the **post-condition**, relating input and output.
R consists of subterms of ρ, it is redundant for pedagogical reasons.
A problem is **applicable** iff η evaluates to true, and a problem is **solved** iff there exists a set V of values for all output-variables, $\overline{O} = O \times V$ such that $\rho\ (\mathit{Val}\ I)\ V$ evaluates to true. The set V is called the **solution** of L.

The relation η guards the input-items in order to ward off unreasonable values [6]. The maximum-example written as a problem is

problem "maximum"
$I \quad \equiv \{\ \{\ r = 7\ \}\ \}$
$\eta(r) \quad \equiv (0 \leq r)$
$O \quad \equiv \{\ A,\ \{a, b\}\ \}$
$\rho(a, b, r) \equiv \exists A.\ A = a \cdot b \wedge (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2 \wedge$
$\forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\frac{a'}{2})^2 + (\frac{b'}{2})^2 = r^2 \Longrightarrow A' \leq A$
$R \quad \equiv \{\ \{A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\}\ \}$

[6] The solvability of problems may also concern predicates, say γ, which do not fit into the above distinction of the precondition η and the postcondition ρ: For instance the 'maximum'-problems solvability is commonly described as "for each additional variable you need an additional relation". This is a predicate $\gamma(I, O, R)$ and thus is syntactically related to ρ, but in its relevance in guiding the user completing the problem with appropriate input it may be closer related to η.

where the sets contained in the sets I, O, R indicate elements, the number of is not fixed in the 'type' (see the definition below) of problem; e.g. another maximum-example may compute the maximum rectangle inscribed into a triangle given by three values.

A problem is related to a particular example. We want to have a more abstract notion, which allows for grouping examples with the 'same kind' of input-items, output-variables and relations, and with the 'same kind' of postcondition.

Definition 2.2.3: Let $X_1, X_2 \subset X$ be sets of variables, and $P_1 \subset P$ a set of predicates.
Given the sets I' of variables and structured values, the sets O' and R' of variables, and given two functions η' and ρ', both of them creating predicates, $\eta'(I', X_1) \in P$ and $\rho'(I', O', R', X_1, X_2, P_1) \in P$, then $Y = (I', \eta', O', \rho', R')$ is a **problem-type**. I', O', R' are called the input-components and and are often collected and abbreviated as IOR'.

The functions η' and ρ' are templates for the pre-condition and the post-condition; they evaluate to predicates by term-composition, which requires a new notation: the composition of a term t from elements of some set S we describe by $t(S)$, whereas $(t\ x)$ is still the evaluation of a term by substituting x. '\$' is a term constructor, for instance, $+\,\$\,a\,\$\,b$ composes to $a + b$.
The problem-type capturing the maximum-example could look like

$$
\begin{array}{lll}
\text{problemtype "maximum"} & & \\
I' & \equiv & \{\, \mathit{fix\_} \,\} \\
\eta' & \equiv & \mathit{map}\ (\$\,0\,\$ \leq \$)\ \mathit{fix\_} \\
O' & \equiv & \{\, m\_,\ vs\_ \,\} \\
\rho' & \equiv & \mathit{let}\ x_1 = \mathit{Vars}\ R' - \mathit{Vars}\ (I' \cup O'); \\
 & & \quad\ x_2 = \mathit{Primed}\ (\mathit{Vars}\ R' - m\_'); \\
 & & \quad\ x_3 = \mathit{Vars}\ R'; \\
 & & \quad\ x_4 = \mathit{Primed}\ (\mathit{Vars}\ R'); \\
 & & \ \ \mathit{in}\ \exists m\_\ \$, x_1\ \$.\ rs\_ \wedge \\
 & & \quad\ \forall m\_'\ \$,\ x_2\ \$.\ (\lambda\ \$\ x_3\ \$.\ rs\_)\ x_4 \Longrightarrow m\_' \leq m\_ \\
R' & \equiv & \{\, rs\_ \,\}
\end{array}
$$

Note how the prerequisite $((\mathit{Vars}\ I') \cup (\mathit{Vars}\ O') \cup (\mathit{Vars}\ R')) \cap ((\mathit{Vars}\ I) \cup (\mathit{Vars}\ O) \cup (\mathit{Vars}\ R)) = \emptyset$, where $(I, O, R) = F$ is a formalization, is guaranteed by marking all identifiers in Y with an underscore at the end.

The notion of problem-type, as defined so far, is a weak concept. What we want to have is: given the formalizations $\mathcal{F}$ of some examples and some problem-types $\mathcal{Y}$, establish a relation between $\mathcal{F}$ and $\mathcal{Y}$ in order to get some

grouping, where an $F \in \mathcal{F}$ belongs to *one* $Y \in \mathcal{Y}$, and not to the others. This relation, called 'F *instantiates* Y, will be developed in the sequel, and requires one last preparatory step.

Definition 2.2.4: Let $IOR' \in \mathcal{I}$ be input-components of some problem-type $Y = (I', \eta', O', \rho', R')$, $F \in \mathcal{F}$ formalizations, X a set of variables and T a set of terms.

$$\begin{array}{rcll} \textbf{Ymatch} & : & \mathcal{I} \times \mathcal{F} & \longrightarrow \quad \mathcal{P}(X \times T) \\ & & \mathit{Ymatch}\ IOR'\ F & \equiv \quad \bigcup_{i \in IOR', f \in F} \mathit{match}\ i\ f \quad \equiv \quad \sigma_Y \\ \textbf{Ymatching} & : & \mathcal{I} \times \mathcal{F} & \longrightarrow \quad \mathit{Bool} \\ & & \mathit{Ymatching}\ IOR'\ F & \equiv \quad (\mathit{Vars}\ IOR') \cap (\mathit{Vars}\ F) = \emptyset \quad \wedge \\ & & & \quad\ \forall\, t' \in IOR'.\ \exists!\, t \in F.\ \mathit{matching}\ t'\ t \end{array}$$

Ymatching cares for completeness of the input w.r.t Y; *Ymatch* generates an environment σ_Y for instantiating a problem-type Y with formalization F. Now all notions are developed for describing the key-point, how to instantiate a problem-type with a formalization in order to get a specified problem:

Definition 2.2.5: Let $F_0, F \in \mathcal{F}$ 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, and P a set of predicates.
Then we say F **instantiates** Y **given** F_0 **yielding** L iff

(i) $F_0 = \emptyset \ \wedge$ *Ymatching* $IOR'\ F$ while generating $\sigma_Y = \mathit{Ymatch}\ IOR'\ F$
$\vee\ F_0 \neq \emptyset \ \wedge$ *Ymatching* $F_0\ F$ while generating $\sigma_Y = \mathit{Ymatch}\ F_0\ F$
(ii) $\sigma_Y\ (\eta'(\mathit{Vars}\ I')\ (\mathit{Val}\ I))$ i.e. the precondition is true
(iii) $\exists \rho \in P.\ \rho = \rho'(\mathit{Vars}\ I', O', \mathit{Vars}\ R', \mathit{Vars}\ I, O, \mathit{Vars}\ R)$
(iv) $L = (I, \eta, O, \rho, R)$

Condition (i) contains a case-distinction discussed in the paragraph below, condition (iii) states that the postcondition can be constructed by term-composition, \$. Evaluation of the postcondition will be adequate as soon as the variables in O have got their respective values.

Stepwise model and specify an example is the usual mode of using the tutor. Discussing this mode below will include the explanation of the case-distinction in the definition of *instantiate* and the introduction of an additional notion.
First let us explain the two cases in (i) of Def.2.2.5: F_0 is a formalization of a particular example, known to the system, i.e. it has been prepared by an author while compiling the respective example collection. F_0 may b e empty

(first case) according to the possibility, that the student inputs an example *not* known to the system. In this case *Ymatch* is weak; for instance by

$$matching \;\; rs\_ \;\; \{A = a + b\}$$

the system has no means to reject a wrong formula for the rectangles area, as long as it is of correct type. An appropriate rejection is possible, however, if $F_0 \neq \emptyset$:

$$matching \;\; \{A = a \cdot b, (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\} \;\; \{A = a + b\}$$

In this case (second case in (i) of Def.2.2.5) even reveals that the second item is missing; this knowledge can be used to guide the students input.

Introducing another feature disambiguates the input even more in both cases: a keyword can be provided for each element of the input-components, yielding the following formalization of the maximum-example:

$$\begin{array}{ll} F_{0_I} & \equiv (I, O, R), \text{ where} \\ I & \equiv \{\ \mathit{fixed\_values}\ \{r = 7\}\ \} \\ O & \equiv \{\ \mathit{maximum}\ A,\ \mathit{values\_for}\ \{a, b\}\ \} \\ R & \equiv \{\ \mathit{relations}\ \{A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\}\ \} \end{array}$$

These keywords are called **descriptions**; the problem-type is being decorated with these descriptions, too:

$$\begin{array}{ll} \text{problemtype "maximum"} & \\ I' & \equiv \{\ \mathit{fixed\_values}\ \mathit{fix\_}\ \} \\ \eta' & \equiv \mathit{map}\ (\$\,0\,\$ \leq \$)\ \mathit{fix\_} \\ O' & \equiv \{\ \mathit{maximum}\ m\_,\ \mathit{values\_for}\ vs\_\ \} \\ \rho' & \equiv \mathit{let}\ x_1 = \mathit{Vars}\ R' - \mathit{Vars}\ (I' \cup O'); \\ & \qquad x_2 = \mathit{Primed}\ (\mathit{Vars}\ R' - m\_'); \\ & \qquad x_3 = \mathit{Vars}\ R'; \\ & \qquad x_4 = \mathit{Primed}\ (\mathit{Vars}\ R'); \\ & \quad \mathit{in}\ \exists m\_\ \$, x_1\ \$.\ rs\_ \wedge \\ & \qquad \forall m\_'\ \$,\ x_2\ \$.\ (\lambda\ \$\ x_3\ \$.\ rs\_)\ x_4 \Longrightarrow m\_' \leq m\_ \\ R' & \equiv \{\ \mathit{relations}\ rs\_\ \} \end{array}$$

The problem resulting from instantiation of the problem-type with the formalization is the following:

$$\begin{array}{ll} \text{problem "maximum"} & \\ I & \equiv \{\ \mathit{fixed\_values}\{r = 7\}\ \} \\ \eta(r) & \equiv (0 \leq r) \\ O & \equiv \{\ \mathit{maximum}\ A,\ \mathit{values\_for}\ \{a, b\}\ \} \\ \rho(a, b, r) & \equiv \exists A.\ A = a \cdot b \wedge (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2 \wedge \end{array}$$

$$\forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\tfrac{a'}{2})^2 + (\tfrac{b'}{2})^2 = r^2 \Longrightarrow A' \leq A$$
$$R \equiv \{\ \textit{relations}\ \{A = a \cdot b,\ (\tfrac{a}{2})^2 + (\tfrac{b}{2})^2 = r^2\}\ \}$$

This problem can be initially presented with the descriptions only, providing the student with suggestive help for input. And on an input of $A = a + b$ (a faulty +) the system can not only reject this item; it even can remark: 'an item is missing in *relations*' !

2.2.3 The hierarchies of subproblems and refinements

There are two completely different kinds of hierarchies on problems: (1) given an example, into which subproblems can the problem broken down in order to find a solution, and (2) given a problem-type, which other problem-types refine the given one by providing more input-items or stronger pre-conditions ?

The hierarchy of subproblems is established by the question raised for each problem: which are the subproblems to be solved in order to solve the problem at hand. The decision for different sequences of subproblems is a characteristic difference between methods solving the same problem. Methods will be discussed in 2.4, and a method solving maximum-example can be found there, too.
Here only the subproblems of the maximum-example are presented. Having specified the example, we may ask, what has been gained by specifying it? Experience in program construction [Gri81] shows that formalizing the postcondition of a problem is the first step, which in certain cases can be followed by rather mechanical steps leading successfully to a solution. This is not the case with this example. Rather it needs a really creative idea, namely to introduce a function - a notion not indicated by anything in the problem as described so far.

Once this ideas has been conceived, one can 'divide and conquer' and break the problem down into subproblems: make a function, find the maximum (-argument) of the function in a reasonable interval, and calculate the values required:

problem "make-fun"
$I \equiv \{\ \textit{function\_of}\ A = a \cdot b,\ \textit{bound\_variable}\ a,$
$\quad \textit{equalities}\{(\tfrac{a}{2})^2 + (\tfrac{b}{2})^2 = r^2\}\ \}$
$\eta \equiv A\ \textit{is\_variable}$
$O \equiv \{\ \textit{function\_term}\ A_1\ \}$
$\rho \equiv \epsilon$
$R \equiv \{\ \}$

Solving the problem "make-fun" will yield the function $a \cdot \sqrt{4 \cdot r^2 - a^2}$ which is an input-item of the next subproblem:

problem "max-of-fun-on-interval"
$I \equiv \{\ \mathit{function\_term}\ a \cdot \sqrt{4 \cdot r^2 - a^2},\ \mathit{bound\_variable}\ a,$
$\quad \mathit{interval}\ \{x.0 \leq x \wedge x \leq 2 \cdot r\}\ \}$
$\eta \equiv \epsilon$
$O \equiv \{\ \mathit{max\_argument}\ a_1\ \}$
$\rho \equiv \forall x.\ 0 \leq x \wedge x \leq 2 \cdot r$
$\quad \Rightarrow \left(\lambda a.\ a \cdot \sqrt{4 \cdot r^2 - a^2}\right)\ x \leq \left(\lambda a.\ a \cdot \sqrt{4 \cdot r^2 - a^2}\right)\ a_1$
$R \equiv \{\ \}$

After this most important subproblem [7] has found a solution, say $(a_1, \sqrt{2} \cdot r)$, the final task is:

problem "calculate-values"
$I \equiv \{\ \mathit{max\_argument}(a_1, \sqrt{2} \cdot r),$
$\quad \mathit{function\_term}\left(A_1, a \cdot 2 \cdot \sqrt{r^2 - (\frac{a}{2})^2}\right),\ \mathit{bound\_variable}\ a\ \}$
$\eta \equiv \epsilon$
$O \equiv \{\ \mathit{values\_for}\{a, b\}\ \}$
$\rho \equiv \epsilon$
$R \equiv \{\ \}$

A careful scan over the input-components of the subproblems reveals, that the formalization F_0 provided by the author would need some additional elements:

$F_0 = \{\ \mathit{fixed\_values}\ \{r =\},$
$\quad \mathit{maximum}\ A,\ \ \mathit{values\_for}\ \{a, b\},$
$\quad \mathit{relations}\ \{A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\},$
$\quad \mathit{bound\_variable}\ b,\ \ \mathit{interval}\ \{x.\ 0 \leq x \wedge x \leq 2 \cdot r\}$
$\quad \mathit{error\_bound}\ (\epsilon = 0.0)\ \}$

Having identified these problems as subproblems (which may be further nested, e.g. "max-of-fun-on-interval" will again be broken down into "differentiate" and "equation" at least), each of them hopefully can be solved more easily then the original one. Using the subproblems solutions on the 'top-level' in solving the maximum-example, the only task on this level is to direct the flow of data between the subproblems.

Clearly, this kind of subproblems introduces the notion of modularization. A module provides for re-use in different problem-types and for parallel use in multiple instances in one and the same problem.
The notion of a 'module' stems from software technology. When mechanizing proofs we can not (and do not want to) foresee how problems are being combined. For instance, what shall we do if more than one polynomial

[7] Not even the postcondition of *this* subproblem gives a clear hint for the respective subproblems, differentiation of A_1, solving the equation $A_1' = 0$, etc.

equation of degree two, described by the structured value $x^2 + 2x + 3 = 0$, has to be solved within one problem ? Assuming, that the problem-type introduces the variables p, q from the structured value, we get several instances of p, q. In order to distinguish these we have to use the mechanisms established for such cases by software technology, i.e. we have to deal with scopes, environments and parameter passing. This will be done in the subsequent section 2.4, and a case study in 5.2 illustrates the usefulness of the subproblem-hierarchy for solving equations.

The problem refinement hierarchy is created by a particular relation between problem-types. The hierarchy allows for mechanically searching for the problem-type most appropriate for a particular formalization. The relation is defined by

Definition 2.2.6: 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 $\forall F \in \mathcal{F}.\ F$ *instantiates* $Y_2 \Rightarrow F$ *instantiates* Y_1.

This definition, based primarily on an implication, induces a quasiorder which in turn induces the hierarchy mentioned. Here are some properties within this hierarchy:
Corollary: Let P be a set of predicates. 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 formalization $F = I \cup O \cup R$ with input-items I and output-variables O, then
(1) $I'_1 \subseteq I'_2$
(2) $O'_1 \subseteq O'_2$
(3) $\sigma_{Y_1}\ (\eta'_1(I'_1)\ I) \Rightarrow \sigma_{Y_2}\ (\eta'_2(I'_2)\ I)$

The resulting hierarchies are modeled by acyclic graphs (basically terms); let us recall the respective notions *below* and *parallel* (Def.2.1.1) and define

Definition 2.2.7: Let $id_i \in ID$ be strings called 'labels', and $Y_i \in \mathcal{Y}$ some problem-types. Then we call the m-tree *problemtree* with constructor *Join* and nodes $(ID \times \mathcal{Y})$

$$\textit{datatype problemtree} = \textit{Join of } ((ID \times \mathcal{Y}) \times (\textit{problemtree list}))$$

a **problem-tree** iff
(i) $\textit{problemtree} \neq \epsilon$
(ii) for all parallel nodes (id_i, Y_i) the labels id_i are pairwise disjunct
(iii) Y_i *below* Y_j iff Y_i *refines* Y_j
(iv) Y_i *parallel* Y_j iff $\neg(Y_i \textit{ refines } Y_j) \wedge \neg(Y_j \textit{ refines } Y_i)$

Given a *problemtree*, we can *automatically refine a vague formulated problem to a stronger formulated one* !
Let us illustrate this with the maximum-example. There are (at least) two possible formalizations: let us call the one shown on p.52, F_{0_I}, then the second is

$$\begin{aligned} F_{0_{II}} \equiv \{ & \textit{fixed\_values } \{r\}, \\ & \textit{maximum } A, \ \ \textit{values\_for } \{a, b\}, \\ & \textit{relations } \{A = a \cdot b,\ \tfrac{a}{2} = r \cdot \sin\alpha,\ \tfrac{b}{2} = r \cdot \cos\alpha\ \} \end{aligned}$$

Depending on the result of modeling there are two different instantiations of the problem-type, the first shown on p.53 and the second

problem "make-fun"
$I \equiv \{ \textit{function\_of } A = a \cdot b,\ \textit{bound\_variable } a,$
$\quad \textit{equalities} \{\ \tfrac{a}{2} = r \cdot \sin\alpha,\ \tfrac{b}{2} = r \cdot \cos\alpha\ \}$
$\eta \equiv A \textit{ is\_variable}$
$O \equiv \{ \textit{function\_term } A_1 \}$
$\rho \equiv \epsilon$
$R \equiv \{\ \}$

Many mathematics textbooks (and all for Austrian high-schools) teach students two different methods to solve 'make-fun' in this context,
(a) eliminate one variable, which yields $A_1 = a \cdot \sqrt{4 \cdot r^2 - a^2}$
(b) introduce a new variable α, which yields $A_1 = 2r \sin\alpha \cdot 2r \cos\alpha$. The applicability of the two methods can be captured by the two respective problem-types

problem "make-fun-by-elimination", $Y_{11} \equiv (I'_{11}, \eta'_{11}, O'_{11}, \rho'_{11}, R'_{11})$
$I'_{11} \equiv \{ \textit{function\_of } f\_ = t\_,\ \textit{bound\_variable } x\_,$
$\quad \textit{equalities } eqs\_$
$\eta'_{11} \equiv f\_ \textit{ is\_variable } \wedge x\_ \in \mathit{Vars}(f\_ = t\_)$
$O'_{11} \equiv \{ \textit{function\_term } f\_1\_ \}$
$\rho'_{11} \equiv \epsilon$
$R'_{11} \equiv \{\ \}$

problem "make-fun-by-new-variable", $Y_{12} \equiv (I'_{12}, \eta'_{12}, O'_{12}, \rho'_{12}, R'_{12})$
$I'_{12} \equiv \cdots$
$\eta'_{12} \equiv f\_ \textit{ is\_variable } \wedge x\_ \notin \mathit{Vars}(f\_ = t\_)$
$O'_{12} \equiv \cdots$
$\rho'_{12} \equiv \cdots$
$R'_{12} \equiv \cdots$

differing (in this case) in the precondition η' only. The preconditions η'_{11}, η'_{12} make the respective problem-types Y_{11}, Y_{12} refine the original Y_2 of 'make-fun':

Y_2 instantiated by $F_{0_I}, F_{0_{II}}$ both yield: $\sigma_{Y_2}(\eta_2(f\_)\ F_{0_I}) \equiv A\ is\_var \equiv true$
Y_{11} instantiated by F_{0_I} yields:

$$\sigma_{Y_{11}}(\eta'_{11}(f\_, x\_, t\_)\ F_{0_I}) \equiv A\ is\_var \wedge\ a \in \{A, a, b\} \equiv true$$

Y_{11} instantiated by $F_{0_{II}}$ yields:

$$\sigma_{Y_{11}}(\eta'_{11}(f\_, x\_, t\_)\ F_{0_{II}}) \equiv A\ is\_var \wedge \alpha \in \{A, a, b\} \equiv false$$

Y_{12} instantiated by F_{0_I} yields:

$$\sigma_{Y_{12}}(\eta'_{12}(f\_, x\_, t\_)\ F_{0_I}) \equiv A\ is\_var \wedge\ a \in \{A, a, b\} \equiv true$$

Y_{12} instantiated by $F_{0_{II}}$ yields:

$$\sigma_{Y_{12}}(\eta'_{12}(f\_, x\_, t\_)\ F_{0_{II}}) \equiv A\ is\_var \wedge \alpha \in \{A, a, b\} \equiv false$$

Thus we have $\sigma_{Y_{11}} \Rightarrow \sigma_{Y_2}$ and $\sigma_{Y_{12}} \Rightarrow \sigma_{Y_2}$, i.e. Y_{11}, Y_{12} refine Y_2, and Y_{11}, Y_{12} are to be located below Y_2 in the hierarchy (when the root is top). And $\sigma_{Y_{11}} \not\Rightarrow \sigma_{Y_{12}}$ as well as $\sigma_{Y_{11}} \not\Leftarrow \sigma_{Y_{12}}$, consequently Y_{11} and Y_{12} can be parallel in the hierarchy.

For addressing a node in *problemtree* all labels on the path to that node are collected:

Definition 2.2.8: Let $p = \{p_1 \cdots p_n\}$ be a position in the problem-tree *problemtree*, and id_i the identifiers in the nodes (id_i, Yi) along the path of p, then we call the list $id = \{id_1, \cdots, id_n\}$ the **problemID** of p.

A *problemID* may become a rather long list. We decided for this choice after trials with large collections of problem types ending up with names like 'poly-equ-univar-deg2', 'poly-equ-univar-degn', 'linear-equ-bivar', etc., which already come close to the length we have now, *but did not show the underlying structure* ! Now the *problemID*s look like

{ *"Real","equation","univariate","linear"* }
{ *"Real","equation","univariate","polynomial","degree 2"* }
{ *"Real","equation","univariate","polynomial","degree n"* }
{ *"Real","equation","univariate","square-root"* }
...
{ *"Real","equation","bivariate","linear"* }
...

The notion of a specification concludes the series of definitions concerning the modeling and specification phase.

Definition 2.2.9: Given a domain D containing typed signatures (and eventually axioms and theorems), a problem-type Y and a method M, then the triple $\mathcal{S} \equiv (D, Y, M)$ is a **specification**.

The notion of a method M will be introduced in section 2.4. The above example in refining problem-types, however, shows already that it is useful, to guard a method with the same structure as given for a problem-type, called Y_M below. This is sufficient for presenting the final definition

Definition 2.2.10: Let F be a formalization, and $\mathcal{S} \equiv (D, Y, M)$ a specification with the domains signature Σ_D, and with method $M = (Y_M, \_)$.
$\mathcal{S}$ is **complete** w.r.t F iff
(i) *parse* f Σ_D is defined for all $f \in F$
(ii) F instantiates Y
(iii) F instantiates Y_M.

Note that (i) is a prerequisite for (ii), and also for (iii), which never has been mentioned before for simplicity reasons.

2.2.4 Summary and related work

This section restricted the variety of mathematics problems to be handled by the tutor to example construction problems, formally defined the respective notions, and illustrated the students activity in modeling and specifying by an example.

A problem is input in the form of a formalization, i.e. by formulae eventually extracted by the student from a textual description of an example. The example can be unknown to the system, or it may be known by a hidden formalization prepared by the author of the respective example collection. Hidden formalizations allow for more rigorous checks and better user-guidance.

Problem-types provide for grouping examples with input-items of equal type and with pre- and post-conditions generated from the same template. A formalization instantiates the problem-type yielding a problem which consists of input-items, a pre-condition, output-variables, a post-condition, and relations. The elements of the relations are to be input by the student, whereas the pre- and post-condition is generated by the system.

The rôle of the post-condition deserves some remarks. The post-condition is the most characteristic part of a problem(-type). A very ambitions project could try to use the post-condition for generating, interactively and more or less guided, a method which solves the problem. Work in this direction has started long time ago, and achieved some remarkable results [MW92]. Generally speaking, an existence proof has to be done by explicitly constructing an example. This thesis is much less ambitious, it confines itself to (large) problem-types which can be solved by *one* method provided by an author.

Another task of the post-condition could be, to check the solution of a particular example. This can easily be done, for instance, with a solution $\alpha \in \mathcal{R}$ of an equation $eq\ x = 0$ with error-bound ϵ,

$$\rho(\alpha, \epsilon) \quad \equiv \quad |eq\ \alpha| \leq \epsilon$$

But this check can *not* be done, for instance, with the post-condition of the maximum-example:

$$\rho(a,b,r) \equiv \exists A.\ A = a \cdot b \wedge (\tfrac{a}{2})^2 + (\tfrac{b}{2})^2 = r^2 \wedge \\ \forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\tfrac{a'}{2})^2 + (\tfrac{b'}{2})^2 = r^2 \Longrightarrow A' \leq A$$

The $\forall a'\ b'\ A'$ obviates the check of a particular example. In this case it even seems to be more straight-forward, to prove the correctness of the method itself w.r.t. the pre- and post-condition. Section 2.4 will develop the description of methods using a specific object-language of the theorem-prover Isabelle. Thus proving the correctness of such a method is not out of the scope of todays research, but it is considered out of scope of this thesis.

The concepts of problem-types and their respective hierarchies allow for substantial acchievments in constructing the tutor; they are a prerequisite for implementing a part of the tutor which

- autonomously models an example, i.e. shows the assignment of the related formulae one by one, if the examples formalization F_0, Def.2.2.1, has been prepared by an author.
- advises the student trying to input a formalization F on his or her own by:
  - giving a description, introduced on p.52, of the formulae to be input
  - notifying on unknown formulae f by use of F *instantiates* Y (Def.2.2.5), Y some problem-type
  - notifying on missing formulae by use of F *instantiates* Y
  - notifying on (pre-) conditions, Def.2.2.2, not met by the input.
- giving the same advice as above (with exception of notifying on missing formulae, which can be done partially with *Ymatch*) even if the example is 'new to the tutor'. This advice depends on the problemtype Y to be specified by the student.
- reports on request the goal of a whole calculation or of a subproblem, i.e. the variables (and their descriptions) of output O and the respective postcondition, Def.2.2.2.
- autonomously does the specification, Def.2.2.9, i.e. assignes the mathematical domain and the problem-type the actual example belongs to, and assignes the method the example shall be solved with, too. This is done by the hidden specification F_0 prepared by an author.

- autonomously refines (Def.2.2.6) a problem w.r.t. the actual example by searching a problem-tree, Def.2.2.7, and thus is able to select the most appropriate method among several possible ones.

The implementation of these features relies on well established techniques of softwaretechnology, acyclic graphs, matching etc. More original work concerns the user-guidance in stepwise input and the liberal model of interaction, introduced later in 3.1.

Related work concerns related software products, and concerns related concepts, discussed below in this order.

Mathpert [Bee84b] is the most closely related product. In comparison to the notions developed in this section, Mathpert does not include problems and problem-types. Thus the user cannot explicitly specify or select a problem-type, and consequently examples cannot be divided into subproblems. The long list of the tutors features from above is missing in Mathpert.

Algebra systems, for instance [Wol96], [CGG+92] or [Sof94], can distinguish between different problems very well. A good example is the *solve* function: it recognices a surprising variety of types of equations, and selects the appropriate methods to solve them; Mathematica and Maple allow the user to select the methods in special cases. In general, however, these systems recognise problems *tacitly* and avoid to challenge the user with decisions on problem-types (and methods in consequence).

Theorem provers, for instance Theorema [BJ98], HOL [GM93], or Isabelle [Pau94], are typically concerned with proving 'universal knowledge problems', i.e. proving theorems in order to extend mathematics knowledge. Solving (example construction) problems means: proving existence by constructing an example with the required properties — and this is a special case, CTPs do not offer special features for, comparable with problem-types and respective hierarchies as proposed in this thesis.

Related concepts are to be found by the question: which domain of knowledge developed an abstract notion of problem, a kind of 'meta description' which provides for formalisms to group examples, to build hierarchies of problem(-types), and to mechanically search the hierachy. Promising fields to get positive answers are some of the subdomains of software technology: formal methods, software architecture and program symthesis, and (meta-) mathematics itself, of course.

Formal specification of an application-problem and proving properties w.r.t. a specification is the concern of 'formal methods', a flourishing subdomain of software technology. The related techniques are applied to several knowledge domains, most successfully to safety critical systems [Bow93] and to hardware verification [Aag92]. For those application domains supportive software systems are available, several of them in commercial and professional versions.

All these successful technologies lay a trustworhty background for the kind of modeling and specification presented in this thesis. Howewever, an immediate usage of these technologies for the objective of this thesis is obstructed for the following reasons. This thesis concerns operating on mathematics knowledge, which is structured in deep hierarchies, containing well modularized, commonly acknowledged, highly re-usable units (as an example one may think of the problem-type equation, and the acknowledged methods to solve them at high-schools). It is an issue of pedagogy to use those units in various contexts and different problems again and again in order to give students a firm grasp of the related notions.

The techniques and tools in formal methods, however, do not provide for mechanical refinement as introduced in Def.2.2.6. Typical specifications contain much more variables than a typical specification in high-school mathematics, and on the other hand they do not have the kind of structure we are interested in: There are elaborated theories of formalized knowledge in some application domains, e.g. for railways [BGH$^+$97], [Han94], [Geo95], [BGP95], [BGH$^+$97], or [DM94], but still there are not the kind of *generally ackowledged and reusable* units which could build up hierarchies of knowledge comparable to mathematics – which is no suprise with respect to the juvenile age of formal methods as compared with the history of mathematics. Thus there was not yet a need for mechanically searching the knowledge already formalized.

Software architecture concerns gross organization and global control structure, protocols for communication, synchronization and data access, assignment of functionality to design elements, physical distribution, scaling and performance, selection among design alternatives, and composition of design elements [SG96]. The latter issue is related to the thesis' notion of problem-type, here called 'pattern'. Within an object oriented architecture (citing 'Formal And Precise Software Patterns Representation Languages' [8]), for instance, such a pattern should allow for an automated support of

- Application: The implementation of a pattern in a given context. For instance, given classes S and $O1, O2$, we may expect a tool to apply the OBSERVER pattern [GHJM94] to S and $O1, O2$, such that $O1$ and $O2$ become observers of S.

[8] http://www.cs.concordia.ca/∼faculty/eden/precise_and_formal

- Validation: Given a source code clip p in an OOP language and the (precise !) specification of a pattern p', we are interested in the answer to the question whether p is an instance of p' (also: p gives rise to p'; p manifests p'; p implements p').
- Recognition: Given a source code clip p in an OOP language and the (precise !) specification of a pattern p', we are interested in the answer to the question whether p is an instance of p', and if so, which element(s) of p' conform to p.
- Discovery: Given a source code clip p in an OOP language and a pattern specification language L, we are interested in the answer to the question whether "some elements" of p form valid patterns (formulae in L), and if so, which, and what patterns do they form.

All together, these points form a notion very close to the thesis' problem-types and the operations on the respective hierarchies.
[vEB97] claims that desing patterns just form another formal language which can be shown to be at least recursively enumerable. This fosters quite an enthusiasm in this community at the time being, but mechanical component search is still a dream of the future [G+96] or more recently [GS00], although there has been partial success already for some time. For instance the Amphion system [VBRLP98] puts together practical-level software composed from subroutine libraries, to meet graphical specifications formulated by astronomers.

Program synthesis aims at the same task as structural composition in software architecture, in principle, however not inclined to programming in the large. The most successful longterm effort seems to be [MW92] constructing functional programs. Gries [Gri81] elaborated Dijkstra [Dij76] for educational purposes, and suggests rather mechanical methods for constructing imperative programs. Looking at the examples given, some of them impressive indeed, shows: program synthesis goes the other way round – from an abstract specification to a (executable) realization, whereas the refinement defined by Def.2.2.6 matches a concrete example with a general problem-type.

Thus it is not worth, again, to go into the details with related work. An example with the KIDS system [Smi91] shows: Under the control of design tactics (plans) that are custom-built to the algorithm and problem classes inference can be tailored to the problem of interest. Problems are formally specified by giving information that includes an input domain and a notion of what constitutes a problem solution. This 'problem theory' is fitted to an 'algorithm class'. (Actually an algorithm theory extends the problem theory in the sense of adding appropriate logical structure including axioms about

the algorithm class. This corresponds to constructing an interpretation between theories.) For example, there is the problem reduction algorithm theory that includes divide-and-conquer and dynamic programming algorithm theories in hierarchical fashion. The divide-and-conquer algorithm theory, when coupled with a problem specification for sorting integers, provides the structure to yield a quicksort algorithm.

Mechanized theories of mathematics, for instance those developed within Isabelle [Pau97b], pose almost contrary obstacles for mechanical problem solving: Mathematics knowledge is highly structured in deep hierarchies of generally acknowledged units (this is true for high-school mathematics, and is less true if one looks to the different thesauri for mathematics at libraries of universities), well. *But this knowledge is not structered w.r.t. application !* Rather, mathematics knowledge is presented in the same way as it has been developed in the axiomatic deductive way. This is particularly true for mathematics theories deduced and checked by computers, for instance by Isabelle [Pau94] or Mizar [Rud92] [9] .
Some years ago a remarkable effort has been announced, the QED manifesto [Ano94], to make mathematics knowledge available for science, engineering and education. Let us cite verbally from the manifesto the part on education: *'The development of mathematical ability is notoriously dependent upon 'doing' rather than upon 'being told' or 'remembering'. The QED system will provide, via such techniques as interactive proof checking algorithms and an endless variety of mathematical results at all levels, an opportunity for the one-on-one presenting, checking, and debugging of mathematical technique, which it is so expensive to provide by the method of one trained mathematician in dialogue with one student. QED can provide an engaging and non-threatening framework for the carrying out of proofs by students [...]. Students will be able to get a deeper understanding of mathematics by seeing better the role that lemmas play in proofs and by seeing which kinds of manipulations are valid in which kinds of structures.'*
There is nothing to add from the point of view of this thesis. QED, however, did not really succeed; the research communities involved have their efforts bound to development of theories within the borders (and limitations) of their respective systems (Mizar, Isabelle, IMPS etc.); and these are considered not ready for education [Lov96].
Time seems not (yet) to have come for a general agreement on a logical and organizational framework for building up mathematics knowledge appropriate for mechanical access for engineering and education. Rather, the present

[9] Modularity and re-usability in the Mizar-project are adressed by an interesting kind of award: the Lesniewski Prize is granted yearly to author(s) of an article with the greatest number of references in MML, the mizar maths library. It may be obtained once only for one article. A winner is proposed yearly based on statistical reports created at the end of the preceding year.

activities are concerned more with technical details [Kir00].

A really application oriented structure of mathematics would approximately look like Fig.2.1, where this figure is due to verbal communication with the creator of the Theorema project [BJ98]. Remarkable groundwork on structuring mathematics by methods has been done in [Buc84], but with no intention to automate anything.

Fig. 2.1: The three-dimensional universe of mathematics

Fig.2.1 is just a sketch, and a drastic simplification, neither axis is 'linear': the highly nested structures necessary for the domain-axis alone may be viewed in Isabelles hierarchy of theories. [10]

This thesis strives for implementable parts of concepts for mechanical problem solving. Thus research of this thesis is restricted to

1. the domain- and problem-axes of the 3D-universe as indicated in Fig.2.1

2. problems of high-school mathematics

Restrictions (1) maintains two axes, domains and problem-types, which still raise a lot of issues. The case study 5.2 on a subclass of equations shows that these issues can be mastered by the concepts developed so far. This study also makes clear, that hard work on a bulk of details has to be expected:

[10] see for instance `http://isabelle.in.tum.de/library/`

how to structure the problem-tree for larger problem classes, which problem-types are in parallel on branches, which predicates are appropriate for the preconditions etc.

Restriction (2) maintains example construction problems, which can be grouped into problem-types solvable by (at least) one method. The method allows to automatically solve a problem, which makes the tutor a partner of the student on an equal base. These features would not apply to other classes of problems like universal knowledge problems.
Furthermore this restriction features field studies on the use of limited portions of knowledge, which nevertheless are considered useful enough to be frequently used. The expected group of users, namely high-school students, are a rather homogeneous group, and it seems comparably easy to plan feed back on their experience.

2.3 Representation and manipulation of calculational proofs

This section proposes the internal and external representation of the specification part and the deductive system appropriate for solving example construction problems (i.e. for doing 'calculational proofs'), and provides the means for the student to operate on the representation.

The internal representation extends the CTPs notion of proof-tree in order to capture not only predicates, but also formulae of arbitrary types, and subproblems.
The external representation suggested follows traditions in mathematics and closely reflects the internal proof-tree.
The manipulation of these representations constitutes the students input language (additional dialog related language elements will be introduced in 3.1). As the tutor fills a gap between the calculational power of CAS and the logical rigor of CTP, the respective input languages of both sides are considered.

2.3.1 Enhanced proof-trees

A proof-tree represents a (partially) completed proof. The basic structure for such a proof-tree is an m-tree. Recalling the notation for data-types and records the definitions is as follows.

Definition 2.3.1: The set $\mathcal{T}$ of **proof-tree**s is inductively defined on nodes $N = (O, Bs)$ where O is called a **proof-object** and Bs is a list called the **branch**es of O:

- $(O, [\,]) \in \mathcal{T}$
- $(O, Bs) \in \mathcal{T}$ where $Bs = [N_1, \cdots, N_n]$ with $N_i \in \mathcal{T}$

There are two types of proof-objects:

- **problem-object**s are records containing all data concerning the specification of an example
- **solve-object**s 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**.

In the proof-tree there are two different types of nodes, one for proof steps in solving, solve-object, and another for (sub) problems, problem-object. Most theorem provers have only one type of node, and provide for other means to combine sub-proofs; for instance, Isabelle [Pau97a] employs a stack of proof-trees, and PVS [ORR+96] has the notion of proof-obligation for that purpose.
The various types of branches, defined below, are in contrast to CTPs, which only have *And*-branches in general. The Theorema prover [Buc96a] employs *And*-branches and *Or*-branches, aiming at a natural deduction style as close to traditional proving as possible. Below there are several types more — thus the headline 'enhanced proof-trees'; the reason is, that CTP is concerned with predicates, whereas the tutor is concerned with terms, in addition to predicates; this introduces the construction of sets, guiding data-flow from one subproblem to the other etc.

First we give the definition of the branch-types, and subsequently motivate them by examples and discuss their design.

Definition 2.3.2: Let P be a set of predicates, T a set of terms, S a set of sets generated by predicates, C a set of (numeral) constants, CC a set of sets of (numeral) constants, and $Bool \equiv \{true, false\}$. Given a parent-node with branches, $(O, [O_1, \cdots, O_n])$, in a proof-tree, with $O, O_i,\ 1 \leq i \leq n$ records of type $\langle expr : expr, tactic : tactic, result : expr, branch : btype \rangle$ where $O.btype$ is enumerated as follows:

$O.branch$ is an **Empty-branch** iff $n = 0$ and
(i) O is a solve-object
(ii) $O.tactic \in R$
(iii) $O.expr \xrightarrow[O.tactic]{} O.result$

$O.branch$ is an **And-branch** iff $n \geq 2$, and
there exist $p_1, \cdots, p_n \in P$ and $b_1, \cdots, b_n \in Bool$ such that
(i) $O.expr \equiv {}'p_1 \wedge \cdots \wedge p_n{}'$
(ii) $O_1.expr \equiv p_1,\ \cdots,\ O_n.expr \equiv p_n$
(iii) $O_1.result \equiv b_1,\ \cdots,\ O_n.result \equiv b_n$
(iv) $O.result \equiv b_1 \wedge \cdots \wedge b_n$

$O.branch$ is an **Or-branch** analogously to *And-branch*

$O.branch$ is a **Transitive-branch** iff $n \geq 2$ and
(i) $O.expr \equiv O_1.expr$
(ii) $O_i.result \equiv O_{i+1}.expr$ for $1 \leq i \leq n-1$
(iii) $O.result \equiv O_n.result$

$O.branch$ is a **Collect-branch** iff $n \geq 2$ and
there exist $s \in S$ and $c_1, \cdots, c_n \in C$ such that
(i) $O.expr \equiv {}'\{c_1, \cdots, c_n\} \cap s{}'$
(ii) $O_i.expr \equiv c_i \in s$ and $O.result \in Bool$ for $1 \leq i \leq n$

(iii) $O.result \equiv \{c_i,\ 1 \leq i \leq n.\ c_i \in s\}$
$O.branch$ is a **Intersect-branch** iff $n \geq 2$ and
there exist $s_1 \cdots, s_n \in S$ and $cc_1, \cdots, cc_n \in CC$ such that
(i) $O.expr \equiv {}'s_1 \cap s_2 \cap \cdots \cap s_n{}'$
(ii) $O_1.expr \equiv s_1$ and $O.result \equiv cc_1$
(iii) $O_i.expr \equiv cc_{i-1} \cap s_i$ and $O.result \equiv cc_i$ and
$O_i.branch$ are *Collect-branches* for $2 \leq i \leq n$
(iv) $O.result \equiv O_n.result \equiv cc_n$

where $'\cdots op \cdots'$ denotes a formula containing operator *op*. The field *tactic* is redundant in all objects O except in O with Empty-branch; in the latter case the field *branch* is superfluous.

Empty-branch is the base-case of the proof-trees recursive structure.

And-branch occurs, for instance, in induction proofs, connecting the base-case and the induction-step, both required to evaluate to 'true',

$$P(n) = P(1) \wedge (P(n) \Rightarrow P(n+1))$$

This proof can be represented by the following proof-tree,

$$\begin{array}{ll} (\ O\ , & [\ (\ O_1\ ,\ [\ ]\ ) \\ & ,\\ & (\ O_2\ ,\ [\ ]\ )\ ]\ \ ) \end{array}$$

where the proof-objects consist of several fields, in particular $O = \langle O.expr = P(n), O.branch = And, O.result = true, \cdots\rangle$, $O_1 = \langle O_1.expr = P(1), O_1.result = true, \cdots\rangle$, and $O_2 = \langle O_2.expr = P(n) \Rightarrow P(n+1),\ O_2.result = true\rangle$. For readability reasons let us take a less formal notation as follows:

$$\begin{array}{ll} O.expr & = P(n) \\ O.branch & = \text{And} \\ & [\ O_1.expr\ = P(1) \\ & \ \ O_1.result = true \\ & \ , \\ & \ \ O_2.expr\ = P(n) \Rightarrow P(n+1) \\ & \ \ O_2.result = true\ ] \\ O.result & = true \end{array}$$

where the *result*-field of the root-problem is displayed below the respective *branch*-field containing the *And*-branches. More elaborated examples on induction can be found in 5.4.

Transitive-branch is the first special branch-type for example construction proofs, modelling chains of tactic applications, where a given formula f is transformed by applying a tactic r yielding f' etc., written as $f \overset{r}{\rightarrow} f'$, for instance

$$\frac{d}{dx}(x^3+x^2+x+1) \xrightarrow[\frac{d}{dx}\,u+v \mapsto \frac{d}{dx}\,u+\frac{d}{dx}\,v]{} \cdots$$

$$\cdots \xrightarrow[\frac{d}{dx}\,x \mapsto 1]{} 3x^3+2x+1$$

This example can be represented by the following proof-tree:

$$\begin{array}{ll} O.expr & = x^3+x^2+x+1 \\ O.branch & = \text{Transitive} \\ & [\, O_1.expr = x^3+x^2+x+1 \\ & \;O_1.tactic = \frac{d}{dx}\,u+v \mapsto \frac{d}{dx}\,u+\frac{d}{dx}\,v \\ & \;O_1.result = \frac{d}{dx}(x^3+x^2+x)+\frac{d}{dx}1 \\ & \;, \cdots , \\ & \;O_n.expr = 3x^3+2x+\frac{d}{dx}x \\ & \;O_n.tactic = \frac{d}{dx}\,x \mapsto 1 \\ & \;O_n.result = 3x^3+2x+1\,] \\ O.result & = 3x^3+2x+1 \end{array}$$

Again, the the parent-objects *result*-field is displayed after *branch* for readability reasons.
A little detour shows, that CTP handle such chains not the way done on blackboards and maths textbooks at high-school. The most natural way for a CTP is, to be given the result and to prove its correctness, i.e. to prove

$$\frac{d}{dx}(x^3+x^2+x+1) = 3x^2+2x+1$$

This would miss the clue of exercises at high-school. In Isabelle one could come closer to traditional notation by introducing a scheme variable $?z$

$$\frac{d}{dx}(x^3+x^2+x+1) = ?z$$

$$\frac{d}{dx}3x^2+\frac{d}{dx}(x^2+x+1) = ?z$$

$$\cdots\cdots$$

$$3x^2+2x+x = ?z$$

and finally apply the law of transitivity to obtain the desired result. Note that Isabelles metalogic masters the situation by creating deeper levels in

the prooftree and thus remembering the first formula. The tutor should present this calculation as

$$\begin{aligned} & \frac{d}{dx}(x^3+x^2+x+1) = \\ = \quad & \frac{d}{dx}3x^2 + \frac{d}{dx}(x^2+x+1) = \\ = \quad & \cdots \\ = \quad & 3x^2+2x+x \end{aligned}$$

and exactly this structure is being reflected in the proof-tree by use of *Transitive-branches*.

Collect-branch decomposes a one-step operation into several calculational steps, which are close to what a student does with paper and pencil. An example for such a one-step operation is

$$\{1,2,3,4,5,6,7\} \cap \{m \in \mathcal{N}.\ m//7\} = \{1,7\}$$

where // denotes 'divides' in the natural numbers. For students of certain grades the calculation of the set of divisors is not a one-step calculation, in particular if the number is higher than 7. The following part of a proof-tree provides the structure for a nice presentation to the student:

$$\begin{aligned} & O.expr \quad = \{1,2,3,4,5,6,7\} \cap \{m \in \mathcal{N}.\ m//7\} \\ & O.branch = \text{Collect} \\ & \qquad [\, O_1.expr \ = 1 \in \{m \in \mathcal{N}.\ m//7\} \\ & \qquad\ O_1.result = true \\ & \qquad\ , \\ & \qquad\ O_2.expr \ = 2 \in \{m \in \mathcal{N}.\ m//7\} \\ & \qquad\ O_2.result = false \\ & \qquad\ , \cdots , \\ & \qquad\ O_n.expr \ = 7 \in \{m \in \mathcal{N}.\ m//7\} \\ & \qquad\ O_n.result = true\ ] \\ & O.result \ = \{1,7\} \end{aligned}$$

Now the calculation is spread over several solve-objects each of which may contain other branches, until the calculational step is considered easy enough for the student.

In the context of this branch usually another issue in presenting calculations to students becomes vivid: calculational steps should be justified by simple theorems, apart from rewrite rules; in the example given a respective knowledge-base could contain these theorems:

(1) *diviors* $n = \{m \in \mathcal{N}.\ m//n\}$
(2) $m//n\ =\ m \leq n\ \wedge\ m//n$
(3) $\{x.\ P\,x \wedge Q\,x\} = \{x.\ P\,x\} \cap \{x.\ Q\,x\}$
(6) $a \in \{n.\ P\,n\} = P\,a$

Then the calculation could be represented in the following way, the structure being connected with, and justified by the theorems,

$$
\begin{array}{ll}
\textit{divisors}\ 7 = & \\
= \{m \in \mathcal{N}.\ m//7\} = & (1)\\
= \{m \in \mathcal{N}.\ m \leq 7\ \wedge\ m//7\} = & (2)\\
= \{m \in \mathcal{N}.\ m \leq 7\}\ \cap\ \{m \in \mathcal{N}.\ m//7\} = & (3)\\
= \{1,2,3,4,5,6,7\}\ \cap\ \{m \in \mathcal{N}.\ m//7\} = & (4)\\
\quad 1 \in \{m \in \mathcal{N}.\ m//7\} = & (5)\\
\quad\quad = 1//7 & (6)\\
\quad\quad = \textit{true} & (7)\\
\quad 2 \in \{m \in \mathcal{N}.\ m//7\} = & (8)\\
\quad\quad = \textit{false} & \\
\quad\quad \ldots & \\
\quad 7 \in \{m \in \mathcal{N}.\ m//7\} = & (9)\\
\quad\quad = \textit{true} & \\
= \{1,7\} & (10)
\end{array}
$$

where the labels on the right margin relate to those in the domains knowledge-base, (4) is a purely calculational step (not justified by a rewrite rule), (5), (8), (9) and (10) are just the definition of *Collect-branch*, and (7) is not decomposed any further here.

Intersect-branch is the second type constructing sets. This branch-type provides a structure for the following example, which calculates all local maxima on a given surface:

$$
\begin{array}{ll}
\{(x,y).\ (x,y)\ \textit{is-local-maximum-of}\ (x^3 - y^3 - 3x + 12y + 10)\} = & \\
= \{(x,y).\ f_x(x,y) = 0 \wedge f_y(x,y) = 0\} & (ii)\\
\quad \cap \{(x,y).\ f_{xx}(x,y) f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0\} & \\
\quad \cap \{(x,y).\ f_{xx}(x,y) < 0\} & \\
\quad\quad s_1 = \{(x,y).\ (f_x(x,y) = 0 \wedge f_y(x,y) = 0)\} & (iii)\\
\quad\quad\quad = \cdots & \\
\quad\quad\quad = \{(-1,-2),(-1,2),(1,-2),(1,2)\} & \\
\quad\quad s_2 = \{(-1,-2),(-1,2),(1,-2),(1,2)\} & (iv)\\
\quad\quad\quad\quad \cap \{(x,y).\ f_{xx}(x,y) f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0\} & \\
\quad\quad\quad = \cdots & \\
\quad\quad\quad = \{(-1,-2),(1,2)\} & \\
\quad\quad s_3 = \{(-1,-2),(1,2)\}\ \cap\ \{(x,y).\ f_{xx}(x,y) < 0\} & (iv)\\
\quad\quad\quad = \cdots &
\end{array}
$$

$$\begin{aligned} &= \{(-1,-2)\} \\ &= \{(-1,-2)\} \end{aligned} \tag{v}$$

The labels on the left margin relate to those of of the definition of *Intersect*-branch in Def.2.3.2 and thus give a justification for the respective steps. The $\cdots$ hold the place for rather expensive calculations of different kinds. The respective part of a proof-tree would be:

$$\begin{aligned} O.expr \quad &= \{(x,y).\ (f_x(x,y)=0 \wedge f_y(x,y)=0)\} \\ &\quad \cap \{(x,y).\ f_{xx}(x,y)f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0\} \\ &\quad \cap \{(x,y).\ f_{xx}(x,y) < 0\} \\ O.branch &= \text{Intersect} \\ &\quad [\, O_1.expr \ = \{(x,y).\ (f_x(x,y)=0 \wedge f_y(x,y)=0)\} \\ &\quad\ O_1.result = \{(-1,-2),(-1,2),(1,-2),(1,2)\} \\ &\quad\ , \\ &\quad\ O_2.expr \ = \{(-1,-2),(-1,2),(1,-2),(1,2)\} \\ &\quad\qquad \cap \{(x,y).\ f_{xx}(x,y)f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0\} \\ &\quad\ O_2.result = \{(-1,-2),(1,2)\} \\ &\quad\ , \\ &\quad\ O_3.expr \ = \{(-1,-2),(1,2)\} \ \cap\ \{(x,y).\ f_{xx}(x,y) < 0\} \\ &\quad\ O_3.result = \{(-1,-2)\}\ ] \\ O.result \ &= \{(-1,-2)\} \end{aligned}$$

After solve-object O_1 has produced a set of constants, this set is used for further calculations in O_2 and O_3, both of which will employ *Collect*-branches for their respective representation.

2.3.2 External representation of calculation

The representation of calculations at the front end has to meet two requirements: (1) a clear mapping between the internal proof-tree and the representation on the screen, and (2) the external representation as close as possible to what is written on a blackboard or what one can find in textbooks for high-school mathematics. A quick look at different textbooks makes evident, that their formal representations of calculations differ considerably. Thus there is welcome freedom on the one hand, but high responsibility on the other hand, when designing the details left open in how to present a calculation.
Approaching issue (2) not only simplifies implementation, but also provides a structure which is of pedagogical value; one can assume that the structure of a formal text has a deep impact on the structure of thinking.

First let us enumerate the objects mapped from the proof-tree to the front-end, before discussing the structure they are presented in: On the screen there will be

1. formulas as discussed in 2.1.3
2. specifications as defined in Def.2.2.9, containing
   (a) a function call similar to those in CAS, e.g. *solve* $(x^2 + 3x - 4 = 0,\ x)$; this kind of function call will be called 'CAS-format' in the sequel.
   (b) the identifier of a specified domain.
   (c) the identifier of a specified problem-type (Def.2.2.8).
   (d) the identifier of a specified method.
   (e) the components of the problem (Def.2.2.2) instantiated from a problem-type.
   (f) the components of the instantiated methods guard (Def.2.2.10).
3. tactics, which will be introduced in the subsequent section 2.3.3. For the description of the passive part of the representation here, these examples may suffice: *Rewrite* $(''add\_commute'', b + a = a + b)$, *Rewrite_Set* $''factorize''$, *Subproblem* $(R, [''equation'', ''univar''], \epsilon)$ and *Apply_Method* $(\mathcal{R}, ''solve\_linear'')$.

The objects belonging to one calculation are displayed on a so-called **worksheet**. The worksheets structure allows to identify the objects' type, described below.

Formulas and tactics are distinguished by their alignment on the left and right margin respectively, which coincides with a traditional format of mathematics texts; this is an example:

$L =$ *solve_root_equ* $(\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x})$ $(bdv = x)$ $(\epsilon = 0)$
1. $\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x}$
   Rewrite (square_equation_left, $a \geq 0 \wedge b \geq 0 \Rightarrow (a = b) = (a^2 = b^2)$ *)*
1.1. $(\sqrt{9+4x})^2 = (\sqrt{x} + \sqrt{5+x})^2$ *Rewrite_Set simplify*
1.2. $9 + 4x = 5 + 2x + 2\sqrt{5x + x^2}$ *Rewrite_Set isolate_root*
1.3. $\sqrt{5x + x^2} = \frac{(9+4x)-(5+2x)}{2}$
   Rewrite (square_equation_left, $a \geq 0 \wedge b \geq 0 \Rightarrow (a = b) = (a^2 = b^2)$ *)*
1.4. $(\sqrt{5x + x^2})^2 = (\frac{(9+4x)-(5+2x)}{2})^2$ *Rewrite_Set simplify*
1.5. $5x + x^2 = 4 + \frac{1}{4}x + x^2$ *Rewrite_Set_Inst [(bdv,x)] normalize_equation*
2. $x - 4 = 0$ *Subproblem (*$\mathcal{R}$*,[equation, univar],* ϵ*)*
3. $L_1 =$ *solve_univar* $(x - 4 = 0)$ $(bdv = x)$ *Apply_Method (*$\mathcal{R}$*,solve_linear)*
3'. $L_1 = \{4\}$
Check_elementwise $0 \leq \sqrt{x} + \sqrt{5+x} \wedge 0 \leq 9 + 4x \wedge 0 \leq x^2 + 5x \wedge 0 \leq 2 + x$
$L = \{4\}$

The representation shows a rather isomorphic mapping to the proof-tree, which would consist of the root-problem, a (sub)problem-object for (3.), both of them with *Transitive*-branches. In (1.) there is a solve-object

with *Transitive*-branches containing a second level of solve-objects (1.1.) to (1.5.) with *Empty*-branches.
The labels and indentations on the left margin reflect logic dependencies. Note for instance the bracketed structure of whole calculations belonging to one subproblem, i.e. the first and last line (without labels) of the root-problem, as well as line (3.) and (3'.) with the subproblem. As this lay-out reflects logical structure, it is maintained by the tutor, and cannot be changed by the student.

These design decisions relate to existing user interfaces for CAS, and user interfaces under construction for CTPs. CAS show up with a highly developed graphical representation of the symbols and formulas; they do not, in contrast to the suggestions w.r.t. the tutor, predefine or restrict the layout, because they do not maintain any logical context.
CTP on the other hand, have the input and output closely bound to the structure of the proof. CPT are still designed for use by experts, and thus their interfaces are not at the present state of the art in general interface design technology. But 'user interfaces for theorem provers' [11] is a vivid research area. Particularly interesting is 'Proof General' [Asp00], a generic user interface which also has been instantiated to Isabelle. This interface, however, differs considerably from the layout suggested for the tutor. The communication with Proof General takes place via three buffers (Emacs text widgets). The script buffer holds input, the commands to construct a proof. The goals buffer displays the current list of subgoals to be solved. The response buffer displays other output from the proof assistant. The tutor, instead, suggests aligning formulas and tactics on the left and right margin respectively, aiming at a printout of a calculation similar to one done by hand.
The representation by structured levels of indentation as shown on p.73 allows to fold calculations hiding deeper levels; this 'nested cells representation' technique has been introduced to proof representation by [Buc97]. Analogously to the subproblem labeled by (3.) above, the 'repeat' level could be folded:

$L =$ *solve_root_equ* $(\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x})$ $(bdv = x)$
1. $\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x}$
2. $x - 4 = 0$
Subproblem $(R, [''equation'', ''univar''], \epsilon)$
3. $L_1 =$ *solve_univar* $(x - 4 = 0)$ $(bdv = x)$
Apply_Method $(\mathcal{R}, ''solve\_linear'')$
3'. $L_1 = \{4\}$
$L = \{4\}$

The usefulness of the nested cells representation, and the possibility to 'zoom in and out' for survey or detail, increases with the length of proofs.

[11] Under the headline 'user interfaces for theorem provers' a series of workshops is being held, see http://cs-fm.lboro.ac.uk/facs/events/uitp.htm

Specifications of the root-problem and subproblems concern quite a lot of information, if all components are displayed. The maximum-examples full specification demonstrates this:

$[a,b] = \mathit{find\_maximum}\ (\mathit{fixed\_values}[r=7])\ (\mathit{maximum}\ A)\ \cdots$
$\quad \mathit{domain}\ \mathcal{R}$
$\quad \mathit{problem}\ [''\mathit{optimization}'', ''\mathit{maximum}'']$
$\qquad I \equiv [\ \mathit{fixed\_values}[r=7]\ ]$
$\qquad \eta(r) \equiv (0 \le r)$
$\qquad O \equiv [\ \mathit{maximum}\ A,\ \mathit{values\_for}\ [a,b]\ ]$
$\qquad \rho(a,b,r) \equiv \exists A.\ A = a \cdot b\ \wedge\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\ \wedge$
$\qquad\qquad \forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\frac{a'}{2})^2 + (\frac{b'}{2})^2 = r^2 \Longrightarrow A' \le A$
$\qquad R \equiv [\ \mathit{relations}\ [A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2]\ ]$
$\quad \mathit{method}\ ''\mathit{maximum\_by\_calculus}''$
$\qquad I \equiv [\ \mathit{fixed\_values}[r=7],\ \mathit{error\_bound}\ (\epsilon = 0.0)$
$\qquad\qquad \mathit{bound\_variable}\ b,\ \ \mathit{interval}\ \{x.\ 0 \le x \wedge x \le 2 \cdot r\}\ ]$
$\qquad \eta(r) \equiv (0 \le r)$
$\qquad O \equiv [\ \mathit{maximum}\ A,\ \mathit{values\_for}\ [a,b]\ ]$
$\qquad \rho(a,b,r) \equiv \exists A.\ A = a \cdot b\ \wedge\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2\ \wedge$
$\qquad\qquad \forall a'\ b'\ A'.\ A' = a' \cdot b' \wedge (\frac{a'}{2})^2 + (\frac{b'}{2})^2 = r^2 \Longrightarrow A' \le A$
$\qquad R \equiv [\ \mathit{relations}\ [A = a \cdot b,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2]\ ]$
$[\ a = r\sqrt{2},\ b = r\sqrt{2}\ ]$

The first line shows the 'CAS-format' of the problem, which is not considered very adequate for this example, and thus the end of the line is cut as indicated by $\cdots$.
Frequently, a specification is not interesting at all, for instance with equations as shown in the previous example of a root equation (see p.73). In these cases the possibility to hide some parts of a calculation is indispensable. With equations this is because the emphasis of exercising equations either is on rewriting towards the solution, or at most the emphasis is on selecting the appropriate method; the specification of the input-items, however, is not considered a challenging task.
This is the difference to the maximum-example: the interpretation of the textual description of an example, and the extraction of the input-items, the output-variables, and the relations, is a task to be exercised explicitly by students.

An example of 'zooming' into a special detail of the maximum-example may conclude the discussion of nested cells:

$[a,b] = \mathit{find\_maximum}\ (\mathit{fixed\_values}[r=7])\ (\mathit{maximum}\ A)\ \cdots$
1. $A_1 = \mathit{make\_fun\_by\_explicit}\ (\mathit{function\_of}\ A = ab)\ \cdots$
    1. $L = \mathit{solve\_rational}\ (\mathit{equality}(\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2)\ (\mathit{solve\_for}\ b)$
        1. $(\frac{a}{2})^2 + (\frac{b}{2})^2 = r^2$
        2. $b^2 = \sqrt{4r^2 - a^2}$

3. $b = +\sqrt{4r^2 - a^2} \vee b = -\sqrt{4r^2 - a^2}$
1'. $L = \{b_1 = +\sqrt{4r^2 - a^2}, b_1 = -\sqrt{4r^2 - a^2}\}$
2. $A_1 = Substitute(b, b_1)$
1'. $A_1 = a\sqrt{4r^2 - a^2}$
2. $a_1 = max\_by\_diff\ (function\_term)\ (a\sqrt{4r^2 - a^2})\ \cdots$
2'. $a_1 = r\sqrt{2}$
3. $[a, b] = calculate\_values\ (max\_argumenta_1 = r\sqrt{2})\ \cdots$
3'. $[\, a = r\sqrt{2},\ b = r\sqrt{2}\,]$
$[\, a = r\sqrt{2},\ b = r\sqrt{2}\,]$

This representation is not identical to what a teacher would write to the blackboard. In the authors opinion, students will overcome the differences to traditional representations on blackboards and in textbooks, as soon as they become aware of the avantages:

Given the feature of easily changing from a survey to a view into the details of a voluminous calculation, the student might prefer not to go through on a linear way, extending the calculation only at the current end.
The student might also wish to let a subproblem open for a moment, turning to another subproblem, and later finish the first one. In this case the nesting on the left margin is expected to clearly indicate where this is possible or not.
Another occasion where a students would like to leave a linear walk through is when he or she gets stuck in the calculation, having made wrong decisions before. The natural way, done with paper and pencil, is to continue at some previous point of the calculation considered still ok, and to simply cut the parts of the calculation below depending on the branch-types.
A problem common to all interactive editing is raised by the users who wish to pursue several variants of a calculation: which variants should be kept, how should they be represented in order not to confuse them (and how to maintain logical consistency !), when should they be dropped, and in which extent ? This case may be covered by a undo-facility, as usual.
Proof general [Asp00] uses colours in a proof script to show the state in the proof assistant. Parts of a proof script that have been processed are displayed in blue and are 'locked' – they cannot be edited. Parts of the script currently being processed by the proof assistant are shown in red. Proof General has commands for processing new parts of the buffer, or undoing already processed parts. These features are appropriate for the tutor, too.

2.3.3 Tactics for stepwise manipulation

The construction of a (calculational) proof requires a large and complicated set of tactics in an interactive CTP. The tutors users cannot be expected to be experts, however; thus the design of an appropriate set of commands is really an issue.

The tutors features bridge a gap between CAS and CTP, thus the respective input languages are considered here. CAS accept function calls returning formulas of any type (actually of no type, because the widely available CAS do not have a type system); assumptions about the domain of the formulas concerned, or the restriction to a special method, are often set by switches. The functions deliver their result in one go, whereas the tutor should guide the student through a calculation step by step, similiar to a calculation done by hand.
CTP require the assumptions in an explicit, logical formulation, and they are built for stepwise construction of a proof (not to speak of additional, powerful tools for automated deduction). This is the kind of operation, the tutor employs, too. In addition to CTP, the tutor has to provide for constructing formulas of arbitrary type – how can this be made simple ?

One idea is to simplify matters by a very general notion of tactic covering any kind of step altering the proofstate, called the **1 step - 1 tactic - 1 formula** metaphor. This view includes all phases of a calulation: adding an item during modeling is done by a tactic (*Add_Given* etc. below), specifying a domain, a problem-type or a method is done by a tactic, rewriting a formula with a theorem is done by a tactic. And always, such a step applying a tactic to a formula yields another formula — in this sense the whole record of *Given* ... *Relation* in a problem is considered one 'formula' consisting of individually accessible items. The methaphor can even be maintained within the complicated branch-types *Intersect* and *Collect* as we will see.

The syntax of the input language is described by Backus-Naur-Form.

tactic ::=

Init_Proof (spec , dexpr_list)	*Subproblem (spec , expr_list)*
patt = CAScmd expr_list	
Specify_Domain domID	*Specify_Problem pblID*
Specify_Method metID	*Refine*
Add_Given dexpr	*Del_Given dexpr*
Add_Find dexpr	*Del_Find dexpr*
Add_Relation dexpr	*Del_Relation dexpr*
Apply_Method metID	*Check_Postcond*
Rewrite_Inst (subs , thm)	*Rewrite thm*
Rewrite_Set_Inst (subs , rls)	*Rewrite_Set rls*
Calculate op	*End_Ruleset*

| *Substitute subs* | | *Apply_Assumption expr_list*
| *Take expr* | | *Group (con , int_list)*

| *Split_And str_list* | *And* | *Conclude_And*
| *Split_Or str_list* | *Or* | *Conclude_Or*
| *Begin_Trans* | | *End_Trans*
| *Intersect* | | *End_Intersect*
| *Check_elementwise* | | *Collect_Results*

patt ::= *id* :: (, *id*)* *id*
| (*id* (, *id*)*)
domID ::= *id*
pblID ::= *id_list*
metID ::= (*domID* , *id*)
spec ::= (*domID* , *pblID* , *metID*)

thmID ::= *id*
thm ::= (*thmID* , *expr*)
rls ::= *id*
con ::= $\wedge$ | $\vee$
op ::= + | − | ∗ | / | ˆ

subs ::= [ϵ | ((*expr* , *expr*) (, (*expr* , *expr*))*)]
int_list ::= [(*int* (, *int*)*)]
str_list ::= [(*str* (, *str*)*)]
id_list ::= [(*id* (, *id*)*)]
dexpr_list ::= [ϵ | (*dexpr* (, *dexpr*)*)]
expr_list ::= [ϵ | (*expr* (, *expr*)*)]
dexpr ::= *Description Form*
expr ::= *Form*
id ::= *ID*

where *Form* is a formula of the object language (see p.41), *dexpr* is such a formula preceded by a description Def.2.2.2, and *ID* is an identifier.

The semantics of the input language is given by the application of a tactic r to a formula f at a certain position in proof-tree P. (P, f) is called the **proof-state**. f, called the **given formula**, is present on the worksheet, r yields the **resulting formula** f' presented on the worksheet again, and r causes a side-effect to the proof-tree yielding (P', f'). The side-effect to P is invisible to the user except an eventual alteration in the indentation on the left margin of the worksheet.
The description of the side-effects to P requires the notion of transition.

Each tactic r will be associated with a transition, which will be described by its transition relation [MP92]. If r is applied in a state where the relation evaluates to true, r is called **applicable** in that (proof-)state. Otherwise, if the transition relation evaluates to false, the tactic 'fails' with **not applicable**, without any change to the proof-tree and submits an error message to the student. The elements of the relations range and image are denoted by unprimed identifiers and primed identifiers respectively.

The tactics cover all operations to construct a complete calculation. According to their purpose tactics can be grouped into tactics for subproblems, for specification, for rewriting, for proof-tree construction, and a group of miscellaneous tactics. The typical tactics of each group will given the semantics by a formal definition, which sometimes omits details described separately in an informal way below. Some of the tactics execution depends on the state of the dialog-guide (DG). Some of the tactics even can get additional arguments from the proof-script which will be introduced later in 2.4.

Tactics for subproblems create a problem-object O in a proof-tree. O contains all data concerned with the formalization and specification in the respective fields $O.domain, O.pbltype, O.pbl$, etc. The initialization of the proof, i.e. the creation of the root-object, is the same as the creation of descendant problem-objects; the only difference is that the initial formalization includes descriptions for the formulas.

Definition 2.3.3: Let P be a proof-tree with nodes $N = (O, [(O_1, B_1), \cdots, (O_n, B_n)])$, with the branches B_i of the next deeper level and the O, O_i containing the respective fields $O.hspec, O.hfmz$, $O.pbl, O.result$. Let f be the given formula, and f' the resulting formula.

With **Init_Proof (spec, fmz)** we associate a transition whose
transition relation is given by
$P = \epsilon \ \wedge \ f = \epsilon$
$\wedge$ $P' = (O', [])$ where O' is a problem-object
with $O'.hspec = spec, O'.hfmz = fmz, O'.pbl = \epsilon$
$\wedge$ $f' = O'.pbl$

With **Subproblem ((domID, pblID, metID), fmz)** we associate
a transition whose transition relation is given by
$N = (O, [(O_1, B_1), \cdots, (O_n, B_n)]) \ \wedge \ f = O_n.result$
$\wedge$ $N' = (O, [(O_1, B_1), \cdots, (O_n, B_n), (O'_{n+1}, [])])$
where O'_{n+1} is a problem-object
with $O'_{n+1}.spec = (domID, pblID, metID)$,
$O'_{n+1}.pbl = Yinst_{(Deref\ domID)}\ (Deref\ pblID)\ \epsilon\ fmz$
$\wedge$ $f' = O'_{n+1}.pbl$

Init_Proof (spec, fmz) initializes a proof-tree with 'hidden information' in the respective h-fields. Both arguments may be empty; in this case the model- and specify-phase starts asking the student for the domain and the problem-type he or she wants to work on. If both arguments are given, the DG may decide to skip (parts of) the specification-phase; in that case f' will be accordingly – the DG has this kind of choice for all tactics of this group.

Subproblem (spec, fmz) is denoted to be applicable at any (non-empty) proofstate in the definition above. There are several natural occasions to limitate the application, for instance within a Transitive-branch generated by a rule set. There are, however, no formalized concepts for such limitations yet. The arguments may both be empty; in this case the proof-script may provide for hidden information equivalent the one given by *Init_Proof*.

CAScmd expr_list is a relative of *Subproblem* which immediately calls for a method; similar to a function call in CAS, the arguments must be given completely and in order.

Tactics for specification concern the operations during the model- and specify-phase. They work on problem-objects, fill the respective fields with data, while these data are checked for consistency w.r.t. hidden data (contained in fields with initial letter h) and data already input. The policy accepting input is rather liberal, and the student is expected to draw the right conclusions from the plentyful feedback on the model-state of all formulas input so far.

Definition 2.3.4: Let P be a proof-tree with a node $N = (O, [])$ with the respective fields $O.problem, O.method$ and without branches, and f' a resulting formula.

With **Specify_Domain domID** we associate a transition whose
transition relation is given by
$N = (O, [])$ where O is a problem-object
with the fields $O.problem, O.method$
$\wedge$ f is some formula within $O.problem$
$\wedge$ $N' = (O, [])$
$\wedge$ $f' = O.problem$ while displaying $Ycheck\ O.problem$
$\vee$ f is some formula within $O.method$
$\wedge$ $N' = (O, [])$
$\wedge$ $f' = O.method$ while displaying $Ycheck\ O.method$

With **Add_Given dexpr** we associate a transition whose
transition relation is given by
$N = (O, [])$ where O is a problem-object
with the fields $O.problem, O.method$
$\wedge$ f is some formula within $O.problem$
$\wedge$ $N' = (O', [])$ with $O'.problem = yadd_S\ dexpr\ O.problem$

$$\begin{array}{ll} & \wedge\ f' = O'.problem \text{ while displaying } Ycheck\ O'.problem \\ \vee & \quad f \text{ is some formula within } O.method \\ & \wedge\ N' = (O', []) \text{ with } O'.method = yadd_S\ dexpr\ O.method \\ & \wedge\ f' = O'.method \text{ while displaying } Ycheck\ O'.method \end{array}$$

Specify_Domain domID is the tactic to start a proof with, because the knowledge in the domain addressed by *domID* is necessary to parse the formulas. The DG may decide to apply this tactic tacitly (using the hidden specification provided by *Init_Proof*) for the beginner who may consider other input as more urgent.

Specify_Problem pblID instantiates the problem-type addressed by *pblID* with formulas already input, i.e. checks the formulas w.r.t. this problem-type and labels them accordingly ('correct','superfluous','syntax-error' etc.). The DG may decide to apply this tactic tacitly (using the hidden specification provided by *Init_Proof*) for the beginner who may consider other input as more urgent. The problem-type provides for descriptions (Def.2.2.2) guiding the input of formulas.

Specify_Method metID works analogously to *Specify_Problem*, but may even be skipped or overridden by *Apply_Method*.

Refine tries to find a problem-type in the problem-tree (Def.2.2.7) more appropriate than the currently specified (sub)problem for the formulas already input in this problem. If a refining problem-type has been found, it is instantiated and the resulting problem is displayed.

Add_Given dexpr, Add_Find dexpr, Add_Relation dexpr input formulas to the respective component of a problem $L = (I, \eta, O, \rho, R)$ or to the methods guard of the same structure, *Add_Given* to I, *Add_Find* to O, and *Add_Relation* to R.

Del_Given dexpr, Del_Find dexpr, Del_Relation dexpr concerns the respective components of a problem or a methods guard; no check required.

Apply_Method metID can be applied if the formulas input so far instantiate the guard of the method addressed by *metID*. If so, this method can override the one chosen by *Specify_Method*. Successful application of this tactic finishes the model- and specify-phase of the (sub)problem.

Check_Postcond concerns the current (sub)problem and finishes the solve-phase.

Tactics for rewriting constitute the most typical kind of tactics. Given a formula f and a rewrite rule r, $f \rightarrow_r$ denotes the formula which results from rewriting with r, i.e. if f contains a redex for r then $f \rightarrow_r \neq r$ else $f \rightarrow_r = r$. Given a rule set *rls*, $f \rightarrow_{rls}$ is used analogously. In general tactics of this group are applied within Transitive-branches. Thus all tactics of this

group can be applied to both, $O.expr$ or to $O.result$ respectively (which is shown only in the definition of *Rewrite*). Rewriting always generates a basic solve-object without descendants, i.e. with an Empty-branch.

Definition 2.3.5: Let P be a proof-tree with nodes $N = (O, [(O_1, B_1), \cdots, (O_n, B_n)])$, the B_i the branches of the next deeper level and the O, O_i containing the respective fields $O.branch, O.expr, O.tactic, O.result$. Let f be the given formula, f' the resulting formula, and let r, r_i be rewrite rules. Let furthermore *auto* and *stepwise* be two kinds of the dialog-modus d.

With **Rewrite thm** we associate a transition whose transition relation is given by

$N = (O, [(O_1, B_1), \cdots, (O_n, B_n)])$ where $O.branch = Transitive$
and the $O_i, 1 \leq i \leq n$ are solve-objects
$\wedge\ f \rightarrow_{thm} \neq f$
$\wedge\quad f = O_n.expr$
$\wedge\ N' = (O, [(O_1, B_1), \cdots, (O'_n, [])])$
with $O'_n.expr = f,\ O'_n.tactic = Rewrite\ thm$
$O'_n.result = f \rightarrow_{thm},\ O'_n.branch = Empty$
$\wedge\ f' = O'_n.result$
$\vee\quad f = O_n.result$
$\wedge\ N' = (O, [(O_1, B_1), \cdots, (O_n, B_n), (O'_{n+1}, [])])$
with $O'_{n+1}.expr = f,\ O'_{n+1}.tactic = Rewrite\ thm$
$O'_{n+1}.result = f \rightarrow_{thm},\ O'_{n+1}.branch = Empty$
$\wedge\ f' = O'_{n+1}.result$

With **Rewrite_Set rls** we associate a transition whose transition relation is given by

$N = (O, [(O_1, B_1), \cdots, (O_n, B_n)])$ where $O.branch = Transitive$
and the $O_i, 1 \leq i \leq n$ are solve-objects
$\wedge\ f = O_n.result$
$\wedge\ f \rightarrow_{rls} \neq f$
$\wedge\quad d = auto$
$\wedge\ N' = (O, [(O_1, B_1), \cdots, (O_n, B_n), (O'_{n+1}, [])])$
with $O'_{n+1}.expr = f,\ O'_{n+1}.tactic = Rewrite\_Set\ rls,$
$O'_{n+1}.result = f \rightarrow_{rls},\ O'_{n+1}.branch = Empty$
$\wedge\ f' = O'_{n+1}.result$
$\vee\quad d = stepwise$
$\wedge\ N' = (O, [(O_1, B_1), \cdots, (O_n, B_n), (O'_{n+1}, [(O'_{n1}, [])])])$
with $O'_{n+1}.expr = f,\ O'_{n+1}.tactic = Rewrite\_Set\ rls,$
$O'_{n1}.expr = f,\ O'_{n+1}.branch = Transitive$
$\wedge\ f' = O'_{n1}.expr$

Rewrite thm applies rule *thm* to the current formula. *Rewrite* is, like all the tactics of this group, applicable either to the formula or the result in a

solve-object. In the first case the object is completed, in the second case a new object is appended.

Rewrite_Set rls rewrites with a (terminating) set *rls* of rules. There are two modes of execution which are set by the DG: the *auto* mode uses the rule set as a black box, applying the rules until termination, and returning the last resulting formula. The *stepwise* mode executes an implicit *Begin_Trans* and thus opens a node for applying the rules one by one. On this level no more *Rewrite_Set* is possible (i.e. *Rewrite_Set* may not be nested); rather a new object eventually generated by *Rewrite_Set* will be appended on the level above.

Rewrite_Inst subs thm instantiates *thm* by *subs* before application. The use of this tactic can be shown by the exmple on p.39 factoring out a special variable only, in this case the variable bound in the equation: given the equation in m_2, $2 \cdot E \cdot m_1 + m_2 \cdot 2 \cdot E - m_2 \cdot m_1 \cdot v_1^2 - m_2 \cdot m_1 \cdot v_2^2 = 0$ the theorem $?bdv \cdot ?b - ?bdv \cdot ?c = ?bdv \cdot (?b - ?c)$ is instantiated by (bdv, m_2) to $m_2 \cdot ?b - m_2 \cdot ?c = m_2 \cdot (?b - ?c)$ before application in order to yield the desired result.

Rewrite_Set_Inst subs rls instantiates the whole rule set *rls* by *subs* before applying it to the current formula. Otherwise it works like *Rewrite_Set rls*.

Calculate op searches for two numeral constants n_1, n_2 in the current formula adjacent to op, i.e. $n_1\ op\ n_2$ (for an infix operator like +), generates a proforma theorem (see p.40) reducing $n_1\ op\ n_2$ to *one* numeral constant, and applies that theorem like *Rewrite*.

End_Ruleset is equivalent to *End_Trans*; see below.

Tactics for prooftree construction open new branches on the proof-tree or finish the current one going back to the next higher level. There is a pair of tactics for each branch-type, one opening and one finishing that branch. Even this group maintains the '1 step – 1 tactic – 1 formula' methaphor, i.e. each tactic is applied to a formula at a particular position in the proof-tree, and returns *one* formula (with a particular indentation) to the worksheet. The definition for the *Collect*-branch may stand for all types.

Definition 2.3.6: Let P be a proof-tree with nodes $N = (O, [(O_1, B_1), \cdots, (O_n, B_n)])$, the B_i the branches and the O, O_i containing the respective fields $O.branch, O.expr, O.tactic, O.result$. Let f be the given formula, f' the resulting formula. Let furthermore c_i be (numerical) constants and s a set.

With **Check_elementwise** we associate a transition whose transition relation is given by

$$N = (O, [(O_1, B_1), \cdots, (O_n, [])])$$

where $O.branch \neq Collect$
and $O_n.expr = \{c_1, \cdots, c_l\} \cap s$ with $2 \leq l$
$\wedge$ $f = O_n.expr$
$\wedge$ $N' = (O, [(O_1, B_1), \cdots, (O_n, [(O'_{n1}, [], \cdots, (O'_{nl}, [])])])$
where $O'_{ni}.expr = c_i \ \$ \in \$ \ s \quad$ for $1 \leq i \leq l$
and $O'_{ni}.branch = Transitive$
$\wedge$ $f' = O'_{n1}.expr$
$\vee$ $N = (O, [(O_1, B_1), \cdots, (O_l, B_l)])$
where $2 \leq l, \quad O.branch = Collect,$
and $O.expr = \{c_1, \cdots, c_l\} \cap s$ with $2 \leq l$
$\wedge$ f is a formula within N
$\wedge$ there is such an i that $O_i.result \notin \{true, false\}, \ \ 1 \leq i \leq l$
and for this i : $f' = O_i.expr$

With **Collect_Results** we associate a transition whose transition relation is given by

$N = (O, [(O_1, B_1), \cdots, (O_l, B_l)])$
where $2 \leq l, \quad O.branch = Collect,$
and $O.expr = \{c_1, \cdots, c_l\} \cap s, \ \ O_i.result \in \{true, false\}$
$\wedge$ f is a formula within N
$\wedge$ $N' = (O, [(O_1, B_1), \cdots, (O_l, B_l)])$
where $O'.result = \{c_i.\ 1 \leq i \leq l \Rightarrow O_i.result = true\}$
$\wedge$ $f' = O'.result$

Miscellaneous tactics work similar to the rewriting group, although they serve various purposes.

Substitute subs applies the substitution *subs* to the current formula. An example for the usage of this tactic is the 'maximum-example' p.75.

Apply_Assumption expr_list applies the assumptions *expr_list* kept in the environment of the proof-trees parent-node of type problem-object. Such assumptions are created, for instance, by the rule 'square_equation' $a \geq 0 \wedge b \geq 0 \Rightarrow (a = b)^2 = (a^2 = b^2))$ in the root-equation on p.73. Application of this rule to $\sqrt{9 + 4x}\} = \sqrt{x} + \sqrt{5 + x}$ creates the assumptions $[0 \leq \sqrt{9 + 4x}\}, \ \ 0 \leq \sqrt{x} + \sqrt{5 + x}]$ which have to be checked for all elements x of the solution set.

Take expr is a tactic useful at the beginning of the solve-phase, or when beginning with calculation in one of the branch-types.

Group (con, int_list) prepares a formula for an *Intersect*-branch. In the example on p.71 the formula

$$\{(x,y).\ f_x(x,y) = 0 \ \wedge \ f_y(x,y) = 0 \\ \wedge \ f_{xx}(x,y) f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0 \\ \wedge \ f_{xx}(x,y) < 0\}$$

must be transformed into

$$\begin{array}{ll} & \{(x,y).\ f_x(x,y) = 0 \wedge f_y(x,y) = 0\} \\ \cap & \{(x,y).\ f_{xx}(x,y) f_{yy}(x,y) - (f_{xy}(x,y))^2 > 0\} \\ \cap & \{(x,y).\ f_{xx}(x,y) < 0\} \end{array}$$

and thus the associative $\wedge$ need to be grouped by $Group(\wedge,\ [2,1,1])$ first.

The application of the tutor to further examples will show, whether additional tactics will be necessary to construct the respective calculations in all details.

In order to prepare for subsequent sections, two definitions are derived from the transition relations above.

Definition 2.3.7: Let r be a tactic, (P, f) a proof-state consisting of a proof-tree P and a given formula f. Then we say r **is applicable in** (P, f) iff all predicates in the transition relation of r are true, (all predicates) which do not contain a primed variable.

In the context of function definitions the predicate is written as *applicable* r (P, f) according to SML-syntax.

Definition 2.3.8: Let $r \in \mathcal{R}$ be a tactic, $(P, f) \in \mathcal{P} \times F$ the proof-state, and $(P', f') \in \mathcal{P} \times F$ the primed versions in the associated transition relation ρ. For $\rho \equiv true$ we have a function

$$\begin{array}{rcccc} \mathcal{E}_X & : & \mathcal{R} \times F \times \mathcal{P} & \longrightarrow & F \times \mathcal{P} \\ & & \mathcal{E}_X\ (r, f, P) & = & (f', P') \end{array}$$

Associated to $\mathcal{E}_X$ we define $\mathcal{E}_{Pf}$, the **execution-function** of r as

$$\begin{array}{rcccc} \mathcal{E}_{Pf} & : & \mathcal{R} & \longrightarrow & F \\ & & \mathcal{E}_{Pf}\ r & = & f' \end{array}$$

Note that f is assumed to be visible on the work-sheet, and that the side-effect on P is invisible for $\mathcal{E}_{Pf}$.

2.4 Scripts for reactive user-guidance

The previous section showed how tactics, applied to a proof-state by the user, step by produce the formulae in a calculation.
This section introduces scripts describing algorithms how to solve a collection of examples belonging to a problem type. Scripts suggest appropriate tactics, which in turn generate the formulae from the respective proof-state.
In spite of the imperative *proof-state, scripts use a* functional *language, emphasizing simplicity (evaluation by rewriting) and power (modeling of parallel calculational steps). The language-interpreter does not execute scripts evaluating to formulae of the math object-language, its task is quite unusual:*
The interpreter tries to 'locate' tactics, input by the user, within the script, and only then proposes the next tactic. This is necessary in order cope with the challenge, to resume interpretation after user-input.

2.4.1 The syntax of scripts

The syntax is given by BNF, with the `keywords` written in `typewriter font`. A **(proof-) script** is a term defined by

script ::= `Script` *id arg** = *body*
arg ::= *id* | ((*id* :: *type*))
body ::= *expr*
expr ::= `%` *id* . *expr*
| `let` *id* = *expr* (; *id* = *expr*)* `in` *expr*
| `if` *prop* `then` *expr* `else` *expr*
| `while` *prop expr id*
| `repeat` *expr id*
| `try` *expr id*
| (*expr* `or` *expr*) *id*

| *tac* (*id* | *listexpr*)*

| *listexpr*
| *id*

type ::= *id*
tac ::= *id*

where *id* is an identifier with the usual syntax, *prop* is a proposition constructed by Isabelles logical operators (see appendix B.1), *listexpr* (called **list-expression**) is constructed by Isabelles list functions like `hd, tl, nth`

described in appendix B.2, and *type* are (virtually) all types declared in Isabelles version 99.

Expressions containing some of the keywords `let`, `if` etc. are called **script-expressions**.

Tactics *tac* are (curried) functions. For clarity and simplicity reasons, *listexpr* must not contain a *tac*, and *tac*s must not be nested, i.e.

$$tac \;\; (tac_1\ \overline{v_1}) \cdots (tac_n\ \overline{v_n})$$

a tactic with n arguments again involving tactics (with arguments $\overline{v_i} = v_{i1} \cdots v_{im}$) must be written as

$$\begin{array}{l} \texttt{let}\ v_1 = (tac_1\ \overline{v_1}); \\ \quad \cdots \\ \quad v_n = (tac_n\ \overline{v_n}) \\ \texttt{in}\ tac\ v_1 \cdots v_n \end{array}$$

which is equivalent w.r.t. the usual definition of `let`. Note that the arguments can be evaluated in parallel in this case (which we will refer to as **parallel let**). The other case, the case of a sequence of tactics,

$$\begin{array}{l} tac_n\ (tac_{n-1}\ \cdots (tac_1\ v) \cdots) = \\ = (tac_n\ \circ\ tac_{n-1}\ \circ \cdots \circ\ tac_1)\ v \end{array}$$

which is a composition of functions, accordingly must be written as

$$\begin{array}{l} \texttt{let}\ v = tac_1\ v; \\ \quad \cdots \\ \quad v = tac_{n-1}\ v; \\ \quad v = tac_n\ v \\ \texttt{in}\ v \end{array}$$

where, for simplicity, the *tac*s have only one argument v in this example.

A special concern is the syntactic representation of parallel processes: An important requirement for any description of a calculation is to provide means to model steps, where several of them be taken in parallel. The syntax defined above offers two constructs for modelling parallel steps:

1. `or` models steps *some* of which must be done (and at least one of them); an example is the script for the application of *Rewrite thm bool v* in a canonical rules-set, where the sequence of applying rewrite-rules *thm* is arbitrary.

2. `let` models steps *all* of which must to be done – both, in arbitrary order or in a prescribed *sequence*. The heavy usage of `let` stems from the restriction upon the language made above, that tactics must not be nested and mixed with list-expressions.

A script containing two tactics to be done in arbitrary order is one belonging to the maximum-example (more scripts for the maximum-example can be found in 5.4)

```
Script make_fun_by_new_variable (f_::real) (v_::real) (eqs_::bool list) =
  (let
    h_ = (hd o (filter (Testvar v_))) eqs_;
    es_ = dropWhile (ident h_) eqs_;
    vs_ = dropWhile (ident f_) (Var h_);
    v1_ = Nth #1 vs_;
    v2_ = Nth #2 vs_;
    e1_ = (hd o (filter (Testvar v1_))) es_;
    e2_ = (hd o (filter (Testvar v2_))) es_;
    s_1 = (solve_univar (Reals, [univar,equation], no_met) e1_ v1_);
    s_2 = (solve_univar (Reals, [univar,equation], no_met) e2_ v2_)
  in Substitute [(v_1, (Rhs o hd) s_1),(v_2, (Rhs o hd) s_2)] h_)
```

where the tactic *solve_univar* is called two times with disjunct arguments; the inital list-expressions calculate these argments from the scripts arguments. A script containing a sequence of tacticts to be done in a prescribed order is the one solving root-equations on p.73:

```
Script square_equation (eq_::bool) (v_::real) (err_::real) =
  (let e_ =
    (while (not o is_root_free) do
      %e_. (let
        e_ = try (Rewrite_Set simplify False) eq_;
        e_ = try (repeat (Rewrite assoc_plus_inv False)) e_;
        e_ = try (repeat (Rewrite assoc_mult_inv False)) e_;
        e_ = try (Rewrite_Set isolate_root False) e_;
        in ((Rewrite square_equation_left True) or
            (Rewrite square_equation_right True)) e_)
      eq_);
    e_ = try (Rewrite_Set_Inst [(bdv,v_)] norm_equation False) e_;
    L_ = solve_univar (Reals, [equation,univariate], no_met) e_ v_ err_
  in Check_elementwise L_ Assumptions)
```

As the same rewriting machinery will be used for both, for the meta-language, the language of scripts, and the object-language, the language of math, a clear distinction is necessary: Elements $T_o(\Sigma_o, V_o)$ of the object-language are called 'formulae', and felements $T_m(\Sigma_m, V_m)$ of the meta-language are called 'expressions'. Formal distinctions are: the function constants are different, $\Sigma_o \cap \Sigma_m = \epsilon$, which is acchieved by careful naming. And the variables are different, $V_o \cap V_m = \epsilon$. The latter will be acchieved by appending an underscore to all $v \in V_m$, which is unusual for variables in

mathematics, i.e. we write $v\_$.

2.4.2 The semantics of scripts as a program language

Whereas the syntax of scripts, as presented above, looks like a functional language, the semantics of scripts, however, should be imperative.
This is because of the scripts task, to guide the application of tactics in order to construct the proof-tree for a calculation. Proof-trees are considerably complex, and proof-states controlling the logical correctness of tactic-applications are complex as well, as has been shown in section 2.3; this complexity should not 'infect' the scripts, and the applicability (Def.2.3.7) of tactics should be handled outside of scripts. This can only be done, if the proof-state is invisible to the scripts, and kept as an imperative data-structure. [12]
Hiding the technical details has the other advantage, that the user cannot access the proof-tree directly, eventually violating logical succinctness.

In spite fo the imperative requirements we pursue the intention to come as close to the semantics of a functional language as possible, because this semantics is comparatively simple *and* powerful: (1) the evaluation of disjoint sub-terms can be done in parallel (which is crucial in modeling steps which can be done in arbitrary order in a calculation), and (2) evaluation can be done by rewriting (where a well elaborated rewriting-machinery is already available, developed for deduction over the math object-language; the intention is, to use this machinery for the meta-language as well).

For simplicity reasons let us first present the semantics of the script-expressions, as if they would be executed like in a conventional program language. We use the syntax of SML [MTH90] (omitting the underscores in this case), where *!* references the imperative proof-state P as an argument of the function *applicable* defined by Def.2.3.7:

```
exception err;

fun while p f x      = if p x
                       then while p f (f x)
                       else x;

fun repeat' true f x = if applicable f (!P)
                       then repeat' false f (f x)
                       else raise err
```

[12] Tactics and tacticals of Isabelle [Pau97a], for instance, work just the same way on a proof-state, which is presented in a readable format, while the details of checking the input before extending the proof-state are hidden from the user.

```
  | repeat' false f x = if applicable f (!P)
                      then repeat' false f (f x)
                      else x;
fun repeat f x      = repeat' true f x;

fun try f x         = if applicable f (!P)
                      then f x
                      else x:
infix or;
fun (f1 or f2) x    = if applicable f1 (!P)
                      then f1 x
                      else if applicable f2 (!P)
                           then f2 x
                           else raise err;
```

`while` resembles *until* as found in [BW88]; the script-expressions `if...then...else` and `let...in` have the meaning as usual in functional languages.

Environments and valuation functions are notions useful for further formalizing semantics. We want to assign a meaning to scripts, meanings are drawn from semantic domains; the values occurring in a semantic domain are called **denotable values**; the set *DenVal* of such values in our case is equal to the set *Math* of formulae of the math object-language:

$$DenVal = Math$$

The set *ExprVal* contains all the values that identifiers in a script may represent:

$$ExprVal = Math + Listexpr + Error$$

where + denotes disjoint union, and *Listexpr* is the set of list-expressions, and *Error* is necessary for technical reasons.

The meaning of a variable in a script depends on the context, described by an 'environment':

Definition 2.4.1: Given $i \in Id$ identifiers and $d \in DenVal$. Then an **environment** $\sigma \in Env$ is a mapping

$$\sigma : Id \longrightarrow DenVal$$

together with the two functions

$$\begin{array}{l} \mathit{access}\colon \mathit{Id} \longrightarrow \mathit{Env} \longrightarrow \mathit{DenVal} \\ \quad \mathit{access} = \lambda i.\ \lambda\sigma.\ \sigma\ (i) \\ \mathit{update}\colon \mathit{Id} \longrightarrow \mathit{DenVal} \longrightarrow \mathit{Env} \longrightarrow \mathit{Env} \\ \quad \mathit{update} = \lambda i.\ \lambda d.\ \lambda\sigma.\ [i \mapsto d]\sigma \end{array}$$

where $[i \mapsto d]\sigma$ is updating the function σ by $i \mapsto d$.

By use of the environment, syntax expressions denoted by

$$\mathcal{E} : \mathit{Expr} \longrightarrow \mathit{Env} \longrightarrow \mathit{ExprVal}$$

A method, called denotational semantics [Sch88], systematically assigns a value to *each* expression of the abstract syntax. For instance

$$\begin{array}{l} \mathcal{E}[\![i]\!] = \mathit{accessenv}[\![i]\!] \\ \mathcal{E}[\![\texttt{let}\ i = e_1\ \texttt{in}\ e_2]\!] = \lambda\sigma.\ \mathcal{E}[\![e_2]\!](\mathit{update}[\![i]\!](\mathcal{E}[\![e_1]\!]\sigma)\sigma) \end{array}$$

$$\begin{array}{ll} \mathcal{E}[\![\%\, i.e]\!] & = \lambda\sigma.\ \lambda d.\ \mathcal{E}[\![e]\!](\mathit{update}[\![i]\!]d\sigma) \\ & = \mathit{err} \qquad \text{if } \lambda d.\ \mathcal{E}[\![e]\!](\mathit{update}[\![i]\!]d\sigma) \text{ is no function} \end{array}$$

$$\mathcal{E}[\![\mathit{listexpr}]\!] = \ <\text{as defined in appendix B.2}>$$

where the second argument of $\mathcal{E}$, the environment σ is omitted on *both* sides of the equalities.

These are all expressions we define $\mathcal{E}$ for: In scripts, as a functional (applicative) language, all identifiers are constants and can be given attributes but once, at their point of definition: they come in as arguments of the script, or they are locally defined by the function definition `%`, or `let...in`, that's all. The reason for this incompleteness w.r.t. the development of a denotational semantics is given subsequently.

2.4.3 The scripts interpretation for reactive user-guidance

The task of the scripts interpreter is a *novel* one. It needs not compute elements of *Math* as indicated by the semantics on p.89, which has math formulae (the x) as denotable values. The scripts, in contrary, should serve interaction with the user, and deliver tactics as a proposal, the user might follow, or not, and apply another tactic. Here we introduce the following distinction: tactics proposed to and input by the user are called **user-tactics**, collected in the set *UTac*, whereas tactics contained in the script are called **script-tactics**, collected in the set *STac*. Thus the set of denotable values acually should be

$$\mathit{DenVal} = \mathit{UTac},$$

where *UTac* has been defined with their syntax in Def.2.3.3 and their respective semantics in Def.2.3.8. Thus the argument for the valuation function, we are really interested in, is tactics:

$$\begin{aligned}\mathcal{E}[\![tac\ i_1 \cdots i_n]\!]\sigma \equiv\ & \mathcal{E}_{Pf}(tac\ (\mathcal{E}[\![i_1]\!]\sigma) \ldots (\mathcal{E}[\![i_n]\!]\sigma)) \\ & \text{if } \forall i = 1..n.\ (\mathcal{E}[\![i_i]\!]\sigma) \in Mat \\ & \text{and } [\![tac]\!]\ (\mathcal{E}[\![i_1]\!]\sigma) \ldots (\mathcal{E}[\![i_n]\!]\sigma) \text{ is applicable in } P \\ \\ \equiv\ & napp \\ & \text{if } \exists i = 1..n.\ (\mathcal{E}[\![i_i]\!]\sigma) \notin Mat \\ & \text{or} \\ & \text{if } \forall i = 1..n.\ (\mathcal{E}[\![i_i]\!]\sigma) \in Mat \\ & \text{and } [\![tac]\!]\ (\mathcal{E}[\![i_1]\!]\sigma) \ldots (\mathcal{E}[\![i_n]\!]\sigma) \text{ not applicable in } P\end{aligned}$$

where *napp* may be read as "the tactic is not applicable in the current proof-state", the predicate *applicable* has been defined in Def.2.3.7, and $\mathcal{E}_{Pf}$ is the execution function for tactics, which has been defined in Def.2.3.8, and which references the imperative proof-state. [13]

Now we are ready to describe the relation between user-tactics and script-tactics.

Definition 2.4.2: Let $r = tac\ f_1 \cdots f_m$ be a user-tactic, applicable to the proof-state (P, f). Let $t = tac\ a_1 \cdots a_n$ a be a script-tactic in script s with environment σ.
r **is associated with** t (and symmetrically: t *associated with* r) iff
$(f_1, \cdots, f_m,\ f) \approx_{tac} (\mathcal{E}[\![a_1]\!]\sigma, \cdots, \mathcal{E}[\![a_n]\!]\sigma)$, where $\approx_{tac}$ is defined individually for all pairs in $UTac \times STac$.
The relation **weakly associated**, $\sim_{tac}$, is a superset of $\approx_{tac}$, also defined individually for $UTac \times STac$.

$\approx_{tac}$ and $\sim_{tac}$ are equivalence relations. Sometimes *associated* is called *strongly associated* in order to clearly distinguish from *weakly associated*.

The most characteristical ones of the 'individual details' will be presented and discussed in 2.4.5. Below there are some examples motivating the necessity of the equivalence relation instead of equality; the examples refer to the root-equation with the user-tactics shown flushed right on the worksheet on p.73 and the script-tactics in the script on p.88:

*Rewrite square_equation_left True e_* in the script proposes the user-tactic *Rewrite (square_equation_left*, $a \geq 0 \wedge b \geq 0 \Rightarrow (a = b) = (a^2 = b^2)$), i.e. the theorem is displayed, assumed to be applied to the current

[13] where actually the associated *user*-tactic is being applied to the proof-state, see Def.2.4.2

formula on the worksheet, and thus the formula is not displayed. Also the argument *True* is omitted: this switches to a special handling of conditional rewrite-rules, where the condition is *not* checked *before* rewriting, but stored to the assumptions to be checked later.

*Rewrite_Set simplify False e_* proposes *Rewrite_Set simplify*, a tactic which employs a whole set of theorems, called *simplify* (a canonical simplifier), for rewriting. The detailed rewrites theorem by theorem are kept as branches of the node generated by *Rewrite_Set*, i.e. one level deeper on the worksheet, which may be displayed on request. The switch is set to *False*.

*solve_univar (Reals, [equation,univariate], no_met) e_ v_ err_* proposes *Subproblem (*$\mathcal{R}$*,[equation, univar],* ϵ*)*, i.e. to generate a subproblem-node in the proof-tree, on which a whole specification process can be done. This at least comprises the specification of a method, which is done by the user-tactic *Apply_Method (*$\mathcal{R}$*,solve_linear)*.

Check_elementwise L_ Assumptions proposes the user-tactic *Check_elementwise Assumptions*, which also may be instantiated to *Check_elementwise* $0 \leq \sqrt{x}+\sqrt{5+x} \wedge 0 \leq 9+4x \wedge 0 \leq x^2+5x \wedge \leq 2+x$, i.e. the conditions of the theorem *square_equation_left* instantiated at the respective rewritings, which evaluate to *True* in this example. Again, the details are one level deeper and can be shown on request.

Interprete a script halting at tactics can be verbally described as follows:

1. beginning with the last tactic done (or the root of the scripts body) find the next tactic to do
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 students input (assumed to be a tactic here for simplicity reasons, and not a 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, and find the next tactic to do ...

Interpretation is interrupted at each tactic, and must be ready to resume at some other tactic in the script which is distinguished to be taken in parallel to the one the script would take. This is an important design decision: handling control to the user *and* providing side-effects on the proof-state, both are done at the same time.

The state of script-interpretation has to be described in a way which meets two different requirements:

1. resume interpretation after having passed control to the user
2. cut steps of interpretation done after a certain proof-state.

The latter requirement is a consequence of another design decision: the free choice for the user to apply a tactic to any formula on the worksheet, i.e. the choice to return to a previous proof-state, and thus eventually cut some branches of the proof-tree below the position of the selected formula and cut steps of interpretation accordingly.

Both requirements can be met by use of the **interpreter-state** represented by the following list:

$$type\ IState = (UTac \times DenVal \times Loc \times EnvStack)\ List$$

where *UTac* is a set of user-tactics, *DenVal* is a set of denotable values, *Loc* is a set of locations in a script, and *EnvStack* is a stack of environments.

The list is empty at the start of interpretation of script s, we denote it by I_0. The search for the first applicable tactic *st* ((1) on p.93) uses the environment σ_0 initialized by the (formal and actual) arguments of s. Then the user inputs a tactic *ut*, the math-engine checks it for applicability at the proof-state (assumed successfully), and the interpreter tries to locate ((4) on p.93) *ut* in s — starting by use of I_0 and σ_0, and updating to σ_1.
After locating has been successful, i.e. *outside* the locate-function the first element is being added to I_0; this element comprises the components of the 4-tuple:

- *Tac*: the script-tactic $t = tac\ i_1..i_n$ but with the arguments evaluated, i.e. as $tac\ (\mathcal{E}[\![i_1]\!]\sigma_1)\ldots(\mathcal{E}[\![i_n]\!]\sigma_1))$
- *DenVal*: the value obtained by execution of the tactic, i.e. as $\mathcal{E}_{Pf}(tac\ (\mathcal{E}[\![i_1]\!]\sigma_1)\ldots(\mathcal{E}[\![i_n]\!]\sigma_1))$
- *Loc*: the location of t in s
- *EnvStack*: the stack of environments with σ_1 on top.

In the subsequent steps of interpretation only *one* environment-stack is changed: the one paired with the location of the tactic which generated the current proof-state P. We will refer to this environment-stack as the **environment-stack related to the proof-state** P. For a single step of interpretation *all* tactics in an interpreter-state are necessary; we refer to them as the **list of executed tactics** $^{\#}E \in Etacs = (UTac \times DenVal \times Loc)List$. Instead of $t \in (map\ Utac\ ^{\#}E)$ we write $t \in {}^{\#}E$. The elements of this list are

distinct.

The fuctions on a stack *es* of environments, $es \in EnvStack$ are:

$accessenv : Id \longrightarrow EnvStack \longrightarrow DenVal$
$\quad accessenv\ i\ es = push\ i\ (top\ es)$
$updateenv : Id \longrightarrow DenVal \longrightarrow EnvStack \longrightarrow EnvStack$
$\quad updateenv\ i\ d\ es = push\ (update\ i\ d\ (top\ es))\ (pop\ es)$
$pushenv : Id \longrightarrow DenVal \longrightarrow EnvStack \longrightarrow EnvStack$
$\quad pushenv\ i\ d\ es = \lambda\ es.\ (\ update\ i\ d\ (top\ es))$
$popenv : EnvStack \longrightarrow EnvStack$
$\quad popenv = \lambda\ es.\ pop$

where *push*, *pop* and *top* are the usual functions on stacks (see p.28) and *update* and *access* are defined on p.90 for ordinary environments.

Finally let us introduce some notation: Tactics are the points in a script, where control is passend to the user. If the user again passes the control to the system, this point should be found again. Thus we need the notion of position, which carries over from (Def.2.1.1) as a script can be regarded as a term. A position in a script is called a **location** and denoted by $\#$. $^{\#}t$ denotes a tactic t at position $\#$. Analogously, we will write $^{\#}E$ for the list of executed tactics in a interpreter-state.

2.4.4 Find the next tactic to be done

Find the next tactic which is applicable at the current proof-state, in order to present it to the student (if the dialog-guide does not decide for another interaction, for instance skipping this step, or for presenting only a part for the tactic etc.) – this task is described as

1. given:
   (a) a script s
   (b) the tactic t at location $\#$ in s, short $^{\#}t$, executed last ($^{\#}t = {}^{\#}body$ initially)
   (c) the proof-state P established by execution of t
   (d) the environment-stack σ related to P.
2. find:
   (a) the next tactic $^{\#'}t'$ on a traversal $\mathcal{X}$ of s, or on the same $\mathcal{X}$ find the root of s ("execution of script finished successfully") such that
      i. $^{\#'}t'$ may have been executed already (in a parallel let); in this case $t' \in E$; continue with $\mathcal{X}$

ii. if $t' \notin E$ then finish:
   A. if t' is applicable in P finish returning $^{\#'}t'$ and the modified environment-stack σ'.
   B. if t' is not applicable, fail with "the tutor is helpless"

3. remarks:

   (a) the actual traversal $\mathcal{X}$ of s is a depth first traversal modified by the script-expressions `if` p `then` e_1 `else` e_2, and `while` p `do` e in a simple way; more complicated modifications are caused by the following

   (b) in addition to $\mathcal{X}$ there is another kind of traversal $\mathcal{A}$ checking for "applicability of sub-terms", which recursively descends into the sub-terms and implements the semantics as defined on p.89 in a straight forward manner.

Let us proceed bottom up and start with the formal definition of the search from (3b). This will be a function with the signature

$$\begin{aligned} P \to {}^{\#}E \to {}^{\#}S \to EnvStack \to & \\ & \{app\} \times UTac \\ & + \{skip\} \times DenVal \times EnvStack \\ & + \{napp\} \times EnvStack \\ & + \{err\} \end{aligned}$$

where *app* may be read as "applicable tactic found", *napp* as "not applicable", *skip* as "go on with the value found", and *err* may be read as "script not appropriate for this example".
The components of the four cases of the functions value will be decomposed by the selector-functions *Val* and *Utac*:

$$\begin{aligned} & Val : \cdots \times DenVal \times \cdots \to DenVal \\ & Val\ (\cdots, d, \cdots) \equiv d \\ & Utac : \cdots \times UTac \times \cdots \to UTac \\ & Utac\ (\cdots, t, \cdots) \equiv t \end{aligned}$$

and 'is *xxx*' denotes the whole tuple $(xxx, \cdots) \in (\{xxx\} \times \cdots)$.

Definition 2.4.3: Let $P \in \mathcal{P}$ be a proof-state, and s a script with environment-stack $\sigma \in EnvStack$ related to P and the list $^{\#}E \in Etacs$ of executed tactics. Let $^{\#}S$ be the set of sub-terms of s with their respective locations #, and let $^{\#}UTac$ be a set of tactics, and $DenVal$ a set of denotable values. For brevity reasons we put the (fixed !) arguments P and $^{\#}E$ as subscripts into the function-name as $\mathcal{A}_{PE}$. The **applicability-function** $\mathcal{A}_{PE}$ is defined on all kinds of expression of s as follows:

$$
\begin{array}{lll}
\mathcal{A} & : \mathcal{P} \to \mathit{Etacs} \to {}^{\#}S \to \mathit{EnvStack} \to \cdots & \\
\mathcal{A}_{PE} : & {}^{\#}S \to \mathit{EnvStack} \to & \\
& \quad \{\mathit{app}\} \times \mathit{UTac} & \\
& + \{\mathit{skip}\} \times \mathit{DenVal} \times \mathit{EnvStack} & \\
& + \{\mathit{napp}\} \times \mathit{EnvStack} & \\
& + \{\mathit{err}\} &
\end{array}
$$

$$
\mathcal{A}_{PE}[\![\%\, i.\ e]\!]\sigma \equiv \lambda d.\ \mathcal{A}_{PE}[\![e]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\texttt{let}\ i = e_1\ \texttt{in}\ e_2]\!]\sigma & & \\
\quad \equiv \mathcal{A}_{PE}[\![e_2]\!](\mathit{pushenv}[\![i]\!](\mathit{Val}\ \mathcal{A}_{PE}[\![e_1]\!]\sigma)\sigma) & & \\
\qquad\qquad & \text{if } \mathcal{A}_{PE}[\![e_1]\!]\sigma \text{ is } \mathit{skip} & \\
\quad \equiv \mathit{err} & \text{if } \mathcal{A}_{PE}[\![e_1]\!]\sigma \text{ is } \mathit{napp} & \\
\quad \equiv \mathcal{A}_{PE}[\![e_1]\!]\sigma & \text{otherwise} &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\sigma & & \\
\quad \equiv \mathcal{A}_{PE}[\![e_1]\!]\sigma & \text{if } \mathcal{E}[\![p]\!]\sigma & \\
\quad \equiv \mathcal{A}_{PE}[\![e_2]\!]\sigma & \text{otherwise} &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\texttt{while}\ p\ e\ i]\!]\sigma & \equiv \mathcal{A}_{PE}[\![e\ i]\!]\sigma & \text{if } \mathcal{E}[\![p]\!]\sigma \\
& \equiv (\mathit{skip}, i, \sigma) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\texttt{repeat}\ e\ i]\!]\sigma & \equiv \mathit{err} & \text{if } \mathcal{A}_{PE}[\![e\ i]\!]\sigma \text{ is } \mathit{napp} \\
& \equiv \mathcal{A}_{PE}[\![e\ i]\!]\sigma & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\texttt{try}\ e\ i]\!]\sigma & \equiv (\mathit{skip}, i, \sigma) & \text{if } \mathcal{A}_{PE}[\![e\ i]\!]\sigma \text{ is } \mathit{napp} \\
& \equiv \mathcal{A}_{PE}[\![e\ i]\!]\sigma & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![(e_1\ \texttt{or}\ e_2)\ i]\!]\sigma & \text{for a tactic within } e_i,\ i = 1,2 \text{ found in } {}^{\#}E & \\
\quad \equiv \mathcal{A}_{PE}[\![e_i\ i]\!]\sigma & & \\
\quad \text{for no tactic within } e_i,\ i = 1,2 \text{ found in } {}^{\#}E & & \\
\quad \equiv \mathcal{A}_{PE}[\![e_2\ i]\!]\sigma & \text{if } \mathcal{A}_{PE}[\![e_1\ i]\!]\sigma \text{ is } \mathit{napp} \text{ or } \mathit{skip} & \\
\quad \equiv \mathcal{A}_{PE}[\![e_1\ i]\!]\sigma & \text{otherwise} &
\end{array}
$$

$$
\mathcal{A}_{PE}[\![\mathit{listexpr}]\!] \equiv (\mathit{skip}, \mathcal{E}[\![\mathit{listexpr}]\!]\sigma, \sigma)
$$

$$
\begin{array}{lll}
\mathcal{A}_{PE}[\![\mathit{tac}\ \overline{e}]\!]\sigma & \equiv (\mathit{app}, \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma) & \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma \text{ is } \mathit{applicable\ in}\ P \\
& \equiv (\mathit{skip}, \mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma, \sigma) & \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma\ \in {}^{\#}E \\
& \equiv (\mathit{napp}, \sigma) & \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma \text{ is } \mathit{napp}
\end{array}
$$

The base case of the recursion are the two rules in the last four lines, where $\mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma$ denotes the tactic with the arguments instantiated, and $\mathcal{E}[\![\mathit{tac}\,\overline{e}]\!]\sigma$ denotes the formula resulting from the execution on P.

The script-expression `or` needs special treatment by $\mathcal{A}_{PE}$ (`or` offers, besides `let`, the second kind of parallel steps in computations !): The check for executed tactics in a subterm is possible, because $^{\#}E$ contains tactics together with their location in the script. And the check is necessary, because the traversal by $\mathcal{A}_{PE}$ (and of $\mathcal{X}_{PE}$) should follow the sequence of tactics as chosen by the user; the traversal should *not* find new *intermediate* tactics, just because the script would like to do so.

The above definition of $\mathcal{A}_{PE}$ exploits the scoping within recursive function calls going top down in a tree: the environment within a function is droppped on return of the functions value, i.e. on completion of a subterm; thus *popenv* is not necessary. For the same reason, instead of *pushenv* the normal *update* for environments would be sufficient.

However, the traversal $\mathcal{X}$ searching for the next applicable tactic t_{i+1} does not go top down, it goes as depictured in Fig.2.2. The search starts

Fig. 2.2: Traverse the script for the next tactic

with the location of t_1, and has to 'go up' in the script several times, before again going top down. In order to be precise the notion of *path* is needed: a *path* is a list *ss* of selector-functions *s*. A selector-function is specialized for each script-expression, and for each part of such an expression,

$$s\_f\ [\![f e_1 \cdots e_i \ldots e_n]\!] = [\![f]\!]$$
$$s\_e_i\ [\![f e_1 \cdots e_i \ldots e_n]\!] = [\![e_i]\!]$$

For an arbitrary expression *e* the application of the selector-function is written as *s e* (i.e. the double brackets are omitted). Then a traversal *X* 'going up' the parse-tree of a script (see Fig.2.3) would be done by the following kind of function:

```
fun X scr path =
  case last path of
    s_f   => xxx_0; X scr (drop_last path);
  | s_e_1 => xxx_1; X scr (drop_last path);
  | ...   => ...    X scr (drop_last path);
  | s_e_i => xxx_i; X scr (drop_last path);
  | ...   => ...    X scr (drop_last path);
  | s_e_n => xxx_n; X scr (drop_last path);
```

Fig. 2.3: Path and selector-function for 'go up' in a script

Selector-functions and paths, however, would overload the subsequent presentation of the next-tactic function and the locate-function. This is avoided by introducing a new notation: Given an arbitrary but fixed script *scr*, and a *path* = *[. . . , s_e_i]*, the call of the above function *fun X* is written as

$$\mathcal{X}\ [\![f\ e_1 \cdots (e_i{}^{\#\wedge}) \ldots e_n]\!] = xxx_i$$

and thus staying with the pattern notation, simply extending it by $^{\#\wedge}$ in order to show, that $\mathcal{X}$ arrived at the term $[\![f\, e_1 \cdots e_i \ldots e_n]\!]$ on its way 'going up' from subterm $[\![e_i]\!]$. And

$$\mathcal{X}\ [\![(f\ e_1 \cdots e_i \ldots e_n)^{\#\wedge}]\!]$$

denotes the call *X scr (drop_last path)*. Now the definition of the traversal can be written as follows: all script-expressions appear on the left-hand side, and $^{\#\wedge}$ takes each meaningful position within an expression (except the last one, i.e. $\mathcal{X}\ [\![(f\, e_1 \cdots e_i \ldots e_n)^{\#\wedge}]\!]$); list-expressions and tactics do not appear on the left-hand side, because they are atomic and thus cannot be reached by 'going up'.

Definition 2.4.4: Let $P \in \mathcal{P}$ be a proof-state, and s a script with body *body*, with environment-stack $\sigma \in$ *EnvStack* related to P and the list $^{\#}E \in$ *Etacs* of executed tactics. Let $^{\#}S$ be the set of sub-terms of s with their respective locations #, and let *UTac* be a set of tactics, *DenVal* a set of denotable values, and *EnvStack* a stack of environments. For brevity reasons we put the (fixed !) arguments P and $^{\#}E$ as subscripts into the function-name as $\mathcal{X}_{PE}$. The **next-tactic-function** $\mathcal{X}_{PE}$ is defined on all kinds of expression of s as follows:

$$\begin{array}{ll} \mathcal{X} & : \mathcal{P} \to \mathit{Etacs} \to \mathit{DenVal} \to {}^{\#}S \to \mathit{EnvStack} \longrightarrow \cdots \\ \mathcal{X}_{PE} : & \mathit{DenVal} \to {}^{\#}S \to \mathit{EnvStack} \longrightarrow \\ & \qquad\qquad \mathit{UTac} + \{\mathit{finished}, \mathit{helpless}\} \end{array}$$

$$\mathcal{X}_{PE} d [\![\%\, i.\ (e\,{}^{\#\wedge})]\!] \sigma \ \equiv\ \mathcal{X}_{PE} d [\![(\%\, i.\ e)^{\#\wedge}]\!] (\mathit{popenv}\ \sigma)$$

$$
\begin{aligned}
&\mathcal{X}_{PE}\,d[\![\texttt{let}\ i = (e_1\ ^{\#\wedge})\ \texttt{in}\ e_2]\!]\sigma \\
&\qquad\qquad \equiv \mathit{Utac}(\mathcal{A}_{PE}[\![e_2]\!](\mathit{pushenv}[\![i]\!]d\sigma)) \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e_2]\!](\mathit{pushenv}[\![i]\!]d\sigma) \text{ is } \mathit{app} \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{let}\ i = e_1\ \texttt{in}\ e_2)^{\#\wedge}]\!](\mathit{popenv}\sigma) \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e_2]\!](\mathit{pushenv}[\![i]\!]d\sigma) \text{ is } \mathit{skip} \\
&\qquad\qquad \equiv \mathit{helpless} \quad \text{if } \mathcal{A}_{PE}[\![e_2]\!](\mathit{pushenv}[\![i]\!]d\sigma) \text{ is } \mathit{napp} \text{ or } \mathit{err} \\
&\mathcal{X}_{PE}\,d[\![\texttt{let}\ i = e_1\ \texttt{in}(e_2\ ^{\#\wedge})]\!]\sigma \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{let}\ i = e_1\ \texttt{in}\ e_2)^{\#\wedge}]\!](\mathit{popenv}\sigma)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{X}_{PE}\,d[\![\texttt{if}\ p\ \texttt{then}(e_1\ ^{\#\wedge})\ \texttt{else}\ e_2]\!]\sigma \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^{\#\wedge}]\!]\sigma \\
&\mathcal{X}_{PE}\,d[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}(e_2\ ^{\#\wedge})]\!]\sigma \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^{\#\wedge}]\!]\sigma
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{X}_{PE}\,d[\![\texttt{while}\ p\ (e\ ^{\#\wedge})\ i]\!]\sigma \\
&\qquad\qquad \equiv \mathit{Utac}(\mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma)) \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{app} \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{while}\ p\ e\ i)^{\#\wedge}]\!]\sigma \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{skip} \\
&\qquad\qquad \equiv \mathit{helpless} \quad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{napp} \text{ or } \mathit{err}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{X}_{PE}\,d[\![\texttt{repeat}(e\ ^{\#\wedge})\ i]\!]\sigma \\
&\qquad\qquad \equiv \mathit{Utac}(\mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma)) \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{app} \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![(\texttt{repeat}\ e\ i)^{\#\wedge}]\!]\sigma \\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{skip} \text{ or } \mathit{napp} \\
&\qquad\qquad \equiv \mathit{helpless} \quad \text{if } \mathcal{A}_{PE}[\![e]\!](\mathit{updateenv}[\![i]\!]d\sigma) \text{ is } \mathit{err}
\end{aligned}
$$

$$
\mathcal{X}_{PE}\,d[\![\texttt{try}(e\ ^{\#\wedge})\ i]\!]\sigma \equiv \mathcal{X}_{PE}\,d[\![(\texttt{try}\ e\ i)^{\#\wedge}]\!]\sigma
$$

$$
\begin{aligned}
&\mathcal{X}_{PE}\,d[\![((e_1\ ^{\#\wedge})\ \texttt{or}\ e_2)\ i]\!]\sigma \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![((e_1\ \texttt{or}\ e_2)\ i)^{\#\wedge}]\!]\sigma \\
&\mathcal{X}_{PE}\,d[\![(e_1\ \texttt{or}\ (e_2\ ^{\#\wedge}))\ i]\!]\sigma \\
&\qquad\qquad \equiv \mathcal{X}_{PE}\,d[\![((e_1\ \texttt{or}\ e_2)\ i)^{\#\wedge}]\!]\sigma
\end{aligned}
$$

$$
\mathcal{X}_{PE}\,d[\![\mathit{body}]\!]\sigma \qquad \equiv \mathit{finished}
$$

where 'is *app*', 'is *napp*', 'is *skip*' and 'is *err*' are used as introduced on p.96.

The next-tactic function $\mathcal{X}_{PE}$ only returns, if any, a (user)-tactic: no proof-state is updated yet — this is delayed after the acknowledge of the user.

$\mathcal{X}_{PE}$ in principle does nothing else as stepping up one level in the expression in order to call the applicability-function $\mathcal{A}_{PE}$. For `try` and `or` the rules

are trivial, because the only situation they apply to is 'going up' from a tactic which has been applied in the preceding step. The search is finished, when the root of the script (i.e. the whole body) is reached (after *skip*).

The search of the first tactic can be done by calling $\mathcal{A}_{PE}$ with the environment-stack containing only the environment made of the formal and the actual arguments of the script. The search for further tactics (i.e. after a tactic t has been found *and executed* propagating the proof-state to, say P, and has been entered to ${}^{\#}E$) is done by $\mathcal{X}_{PE}$ taking as arguments P, the list of executed tactics ${}^{\#}E$, the current formula $d \in DenVal$, the script-expression of tactic $[\![t]\!]$ at its location in the script, and the environment-stack σ related to P.

Calls within a parallel `let` require to start the search with the first subexpression within this `let`, because the user may have chosen a tactic t_{i+1} skipping a tactic t_i in the `let`. Whether a tactic has been skipped or not, this is handled in the basecase $\mathcal{A}_{PE}[\![tac\,\overline{e}]\!]\sigma$ by the test $\in {}^{\#}E$.

Iteration by `while` or `repeat` do not require *pushenv* and *popenv*, because reaching a tactic is recorded by an entry in the interpreter-state. No entry, however, is done for a list-expression in a `let`. The depth of the environment-stack represents the nesting of the script, and *not* the number of iterations.

The rules for `try` and `or` are surprisingly simple, because the only situation these rules are used is 'going up' from the tactic applied in the previous call of $\mathcal{X}_{PE}$, i.e. the initial argument $[\![t]\!]$ of the current call.

2.4.5 Locate a tactic in a script

Locate a user-tactic in the script is the other task to be done in interpreting a script; this is task (4) in the survey on page p.93: after a tactic has been suggested to the student, the student may respond with this tactic or another one. This is the requirement 'resume interpretation of the script after assignment of an arbitrary tactic (applicable in the current proof-state)'. And this requirement is best met by adapting to it the whole mechanism of interpretation. The task in more detail is:

1. given:
   (a) a script s
   (b) a tactic t_0 at location # in s, shortly ${}^{\#}t_0$
   (c) the proof-state P, established by the execution of the user-tactic ut_0 associated with t_0
   (d) the environment-stack σ related to P.
   (e) a just input user-tactic ut, already checked for being applicable in P
2. find:

   (a) the list of tactics $[t_1, \cdots, t_n]$ on a traversal $\mathcal{C}$ of s with t_n is associated with ut (the user is allowed to skip steps !)

   (b) a judgement wether a 'safe' continuation of the guidance by s can be expected, i.e. whether appropriate atctics may be proposed by $\mathcal{X}_{PE}$ in the sequel leading to the result. The judgement is based on how t_n is associated with ut:

      i. continue 'safely' if they are strongly associated, $t_n \approx_{tac} ut$

      ii. continue 'unsafely' if they are weakly associated, $t_n \sim_{tac} ut$; see Fig.2.4.

3. remarks:

   (a) if the list of tactics found contains only one element, $[t_1]$, this tactic can be executed immediately

   (b) if there are several elements in the list of tactis found, $[t_1, \cdots, t_n]$, several decisions have to be made

      i. the dialog-guide (see Chapter 3) decides whether to accept the 'big' step done by the user with ut, or to discuss the intermediate t_i (and to delay updating P)

      ii. special cases are handled by the mathematics-engine, e.g. if all tactics are concerned with rewriting (see below)

   (c) the traversal $\mathcal{C}$ can employ an auxiliary traversal $\mathcal{B}$ top-down on s, analogously as $\mathcal{X}_{PE}$ employs $\mathcal{A}_{PE}$.

Here, before discussing the traversals $\mathcal{C}$ and $\mathcal{B}$ of the script, is the right place to motivate and present the defintion Def.2.4.2 of $\approx_{tac}$ and $\sim_{tac}$ individualized for some tactics.

The user-tactic Substitute is associated with the respective script-tactic in a way which fits well to point (2a) above. Executing all the tactics until an associated one is reached is appropriate: Given the script *make_fun_by_new_variable* on p.88 called with the arguments

$f\_$	A
$v\_$	α
$eqs\_$	$[A = 2ab - a^2,\ \frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha]$

the user may immediately apply the tactic *Substitute* $[(a, 2R\sin\alpha), (b, 2R\cos\alpha)]$ to $A = 2ab - a^2$. This 'big' step even skips solving two subproblems (i.e. solving the two equations $\frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha$ for a and b respectively, which is trivial in this case) in the script. After this step the script (which called *make_fun_by_new_variable*) can be expected to continue 'safely'. The definition Def.2.4.2 specialized for *Substitute* is as follows.

Fig. 2.4: Locate a tactic in a script

Definition 2.4.5: Given the user-tactic *Substitute* σ_{ut} (σ_{ut} is a substitution) applicable in some proof-state (P, f). Given the script-tactic *Substitute* $\sigma_{st} f\_$ (σ_{st} is a substitution, $f\_$ is a variable) contained in a script with environment σ. Given a canonical simplifier D. Then

$$(\mathit{Substitute}\ \sigma_{ut}, f) \quad \approx_{tac} \quad \mathit{Substitute}\ \sigma_{st}\ f\_$$

iff (i) $\sigma_{ut} \subseteq \mathcal{E}[\![\sigma_{st}]\!]\sigma$
(ii) $f \approx_D \mathcal{E}[\![f\_]\!]\sigma$

where $\approx_D$ denotes equivalence modulo D. For *Substitute*, $\sim_{tac}$ is defined equally, $\approx_{tac} \equiv \sim_{tac}$.

(i) allows for stepwise substitution (reflected in the *applicable*-predicate, too), and (ii) allows to associate substitutions equivalent modulo a canonical simplifier appropriate for the respective domain, for the example given

$$\mathit{Substitute}\ [(a, 2R \cdot \sin\alpha), (b, \cos\alpha \cdot 2R)]$$
or $$\mathit{Substitute}\ [(b, 2 \cdot \cos\alpha \cdot R), (a, R \cdot \sin\alpha \cdot 2)]$$

The user-tactic Rewrite is associated with the respective script-tactic in a very different way. Given the script square_equation on p.88 the user might apply the rule-set *norm_equation* even to the initial equation, say

$$\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x}$$

Rewrite_Set_Inst [(bdv,x)] normalize_equation

$$\sqrt{9+4x} - \sqrt{x} - \sqrt{5+x} = 0$$

The search $\mathcal{C}$ as described in (2a) on p.102 is too general in this situation: expecting arbitrary tactics in the script, the search propagates the proof-state at each tactic found applicable before *Rewrite_Set_Inst [(bdv,x)] normalize_equation e_* is reached. While propagating the proof-state the equation is transformed already to (see the work-sheet on p.73)

$$5x + x^2 = 4 + \tfrac{1}{4}x + x^2$$

before *Rewrite_Set_Inst [(bdv,x)] normalize_equation* is being applied. The user, however, applied the tactic to $\sqrt{9+4x} = \sqrt{x} + \sqrt{5+x}$! Thus, in order to continue interpreation of the script in this case (where *all* tactics are concerned with rewriting *only*) the intermediate proof-states need to be dropped.
And the method given by the script *square_equation* continues 'unsafely', i.e. eventually no next tactic can be proposed by $\mathcal{X}_{PE}$; but the user still can apply an appropriate tactic within the scripts *while (not o is_root_free) do*, and subsequently $\mathcal{X}_{PE}$ may succeed, again.
With respect to this situation the following definition is quite different to Def.2.4.5

Definition 2.4.6: Given the user-tactic *Rewrite_Set_Inst* rls_{ut} (rls_{ut} is a rule-set) applicable in some proof-state (P, f). Given the script-tactic *Rewrite_Set_Inst* rls_{st} *bool e_* (rls_{st} is a rule-set, *bool* is a boolean value, *e_* is a variable) contained in a script with environment σ. Then

$$(\textit{Rewrite\_Set\_Inst } rls_{ut}, f) \;\approx_{tac}\; \textit{Rewrite\_Set\_Inst } rls_{st} \textit{ bool } e\_$$

iff (i) $rls_{ut} = rls_{st}$
(ii) $f = \mathcal{E}[\![e\_]\!]\sigma$ equal as terms

and $(\textit{Rewrite\_Set\_Inst } rls_{ut}, f) \;\sim_{tac}\; \textit{Rewrite\_Set\_Inst } rls_{st} \textit{ bool } e\_$

iff (i) $rls_{ut} = rls_{st}$

For being strongly associated, the equality $f = \mathcal{E}[\![e\_]\!]\sigma$ on the term-level is essential, if *Rewrite_Set_Inst, Rewrite_Set* and in particular *Rewrite* should stand for exact rewriting, i.e. for application of one particular theorem. The switch *bool* concerns conditional rewriting and is set by the system.

Traversing the script on the search for an associated tactic is exactly the same as the traversal on the search for a next tactic. In order to stress this welcome fact, we will stay with the notation and the level of abstraction used for $\mathcal{X}_{PE}$ and $\mathcal{A}_{PE}$. Analogously to $\mathcal{X}_{PE}$ the function $\mathcal{O}$ calls a function $\mathcal{B}$, recursively descending analogously to $\mathcal{A}_{PE}$. Both functions, $\mathcal{O}$ and $\mathcal{B}$, take more arguments than their relatives already defined, and deliver a longer tuple as value. The description of passing the arguments and values through the function calls requires SML notation, i.e. *let*, *val*, *in*, *end* distinguished from the script-expressions `let`, `in`. In order to shorten the presentation, only the base-cases and one example are shown in the definition below.

For this definition the data are collected which make up a step: the set *Steps* pairs a set of elements of the interpreter-state, ($UTac \times DenVal \times Loc \times EnvStack$), with a set of proof-states $\mathcal{P}$

$$Steps = (UTac \times DenVal \times Loc \times EnvStack) \quad \times \quad \mathcal{P}$$

Definition 2.4.7: Let $P \in \mathcal{P}$ be a proof-state, and s a script with body *body*, with environment-stack $\sigma \in EnvStack$ related to P and the list ${}^{\#}E \in Etacs$ of executed tactics. Let ${}^{\#}S$ be the set of sub-terms of s with their respective locations #, and $UTac$ be a set of tactics, $DenVal$ a set of denotable values, $EnvStack$ a stack of environments, and *Steps* a set of interpreter- and proof-states as described above.
The **associate-function** $\mathcal{X}_{PE}$ is defined on all kinds of expressions of s:

$$\begin{aligned}
\mathcal{B} \;:\; & \mathcal{P} \to Etacs \to {}^{\#}S \to EnvStack \to UTac \to Steps \longrightarrow \\
& \quad \{ass\} \times Steps \\
& + \{skip\} \times DenVal \times EnvStack \times Etacs \times Steps \\
& + \{napp\} \times EnvStack \times Steps \\
& + \{err\} \\
\vdots & \\
\mathcal{B}\ P\ {}^{\#}E\ [\![\texttt{let}\ i = e_1\ \texttt{in}\ e_2]\!]\ \sigma\ t\ p & \\
& \equiv let\ val(_, (d', \sigma', {}^{\#}E', p')) = \mathcal{B}\ P\ {}^{\#}E\ [\![e_1]\!]\ \sigma\ t\ p; \\
& \qquad val(_, P') = p'; \\
& \quad in\ \ \mathcal{B}\ P'\ {}^{\#}E'\ [\![e_2]\!]\ (pushenv[\![i]\!]d'\sigma')\ t\ p'\ \ end; \\
& \qquad\qquad \text{if } \mathcal{B}\ P\ {}^{\#}E\ [\![e_1]\!]\ \sigma\ t\ p \text{ is } skip \\
& \equiv err \qquad \text{if } \mathcal{B}\ P\ {}^{\#}E\ [\![e_1]\!]\ \sigma\ t\ p \text{ is } napp \\
& \equiv \mathcal{B}\ P\ {}^{\#}E\ [\![e_1]\!]\ \sigma\ t\ p \\
& \qquad\qquad \text{otherwise} \\
\vdots & \\
\mathcal{B}\ P\ {}^{\#}E\ [\![tac\ \overline{e}]\!]\ \sigma\ t\ p & \\
& \equiv (ass,\ ((p \bullet (tac\ \mathcal{E}[\![\overline{e}]\!]\sigma, \mathcal{E}[\![tac\ \overline{e}]\!]\sigma),\ \#,\ \sigma), \\
& \qquad\qquad \mathcal{E}_X(tac\ \mathcal{E}[\![\overline{e}]\!]\sigma, \mathcal{E}[\![tac\ \overline{e}]\!]\sigma, P))) \\
& \qquad\qquad \text{if } tac\ \mathcal{E}[\![\overline{e}]\!]\sigma \text{ is } associated\ with\ t
\end{aligned}$$

$$\begin{aligned}
&\equiv (\mathit{skip},\ \mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma,\ \sigma,\ {}^{\#}E \bullet \mathcal{E}[\![\overline{e}]\!]\sigma,\\
&\qquad ((p \bullet (\mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma, \mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma),\ \#,\ \sigma),\\
&\qquad \mathcal{E}_X(\mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma, \mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma, P)))\\
&\qquad\qquad \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma \text{ is } \mathit{applicable\ in}\ P\\
&\equiv (\mathit{skip},\ \mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma,\ \sigma,\ {}^{\#}E,\ p)\\
&\qquad\qquad \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma\ \in\ {}^{\#}E\\
&\equiv (\mathit{nass},\ \sigma,\ p)\\
&\qquad\qquad \text{if } \mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma \text{ is not } \mathit{associated\ with}\ t
\end{aligned}$$

where the last rule is the base-case, where $\mathit{tac}\ \mathcal{E}[\![\overline{e}]\!]\sigma$ denotes the tactic with the arguments instantiated, $\mathcal{E}[\![\mathit{tac}\ \overline{e}]\!]\sigma$ denotes the formula resulting from the execution on P, and $\mathcal{E}_X$ is the execution function on P. The $\vdots$ stand for rules which are exactly the same as for $\mathcal{A}_{PE}$ except that *ass* stands for *app*, and except of an extended signature (the rule for `let` is an example).

The **locate-function** $\mathcal{O}$ is defined on all kinds of expressions of s:

$$\begin{aligned}
&\mathcal{O}\ :\ \mathcal{P} \to \mathit{Etacs} \to \mathit{DenVal} \to {}^{\#}S \to \mathit{EnvStack} \to \mathit{UTac} \to \mathit{Steps} \longrightarrow\\
&\qquad\qquad \mathit{Steps}\ +\ \mathit{notlocatable}\\
&\vdots\\
&\mathcal{O}\ P\ {}^{\#}E\ d\ [\![\texttt{let}\ i = (e_1\ {}^{\#\wedge})\ \texttt{in}\ e_2]\!]\ \sigma\ t\ p\\
&\qquad\qquad \equiv \mathit{let}\ \mathit{val}(_, p') = \mathcal{B}\ P\ {}^{\#}E\ [\![e_2]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)\ t\ p;\\
&\qquad\qquad\quad \mathit{in}\ \ p'\ \ \mathit{end};\\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{B}\ P\ {}^{\#}E\ [\![e_2]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)\ t\ p \text{ is } \mathit{ass}\\
&\qquad\qquad \equiv \mathit{let}\ \mathit{val}(_, (d', \sigma', {}^{\#}E', p')) = \mathcal{B}\ P\ {}^{\#}E\ [\![e_2]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)\ t\ p;\\
&\qquad\qquad\quad\ \ \mathit{val}(_, P') = p';\\
&\qquad\qquad\quad \mathit{in}\ \ \mathcal{O}\ P'\ {}^{\#}E'\ d'\ [\![e_2]\!]\ (\mathit{popenv}\ \sigma')\ t\ p'\ \ \mathit{end};\\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{B}\ P\ {}^{\#}E\ [\![e_2]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)\ t\ p \text{ is } \mathit{skip}\\
&\qquad\qquad \equiv \mathit{notlocatable}\\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{B}\ P\ {}^{\#}E\ [\![e_2]\!]\ (\mathit{pushenv}[\![i]\!]d\sigma)\ t\ p \text{ is } \mathit{nass} \text{ or } \mathit{err}\\
&\mathcal{O}\ P\ {}^{\#}E\ d\ [\![\texttt{let}\ i = e_1\ \texttt{in}(e_2\ {}^{\#\wedge})]\!]\ \sigma\ t\ p\\
&\qquad\qquad \equiv \mathcal{O}\ P\ {}^{\#}E\ d\ [\![(\texttt{let}\ i = e_1\ \texttt{in}\ e_2)^{\#\wedge}]\!]\ (\mathit{popenv}\ \sigma)\ t\ p\\
&\vdots\\
&\mathcal{O}\ P\ {}^{\#}E\ d\ [\![\mathit{body}]\!]\ t\ p\\
&\qquad\qquad \equiv \mathit{notlocatable}
\end{aligned}$$

where the $\vdots$ stand for rules which are exactly the same as for $\mathcal{X}_{PE}$ except that *ass* stands for *app*, *notlocatable* stands for *helpless* and(!) *finished*, and except of an extended signature.

The last argument $p \in \mathit{Steps}$ serves for appending new list-elements during recursive calls, i.e. the initial call always has $p = [\,]$.

If the search $\mathcal{O}$ for a script-tactic associated to the actually input user-tactic returns *notlocatable*, the search $\mathcal{X}_{PE}$ for the next applicable tactic in

the script cannot be started. As a consequence the script-interpreter is *helpless*. Then a successful continuation depends on the user: He can go back to a previous proof-state (i.e. to a previous formula on the work-sheet), and try the script-interpreter again (which is set to a previous state according to the proof-state), for instance by pushing the button <yourTurn>; at least the first formula of the respective (sub-)proof should lead to a success, provided an adequate specification.

2.4.6 Resume from input of a formula

To resume evaluation of a script after user-input of a formula is very different from resuming from an input user-tactic. The latter is a member of the meta-language concerning scripts, and thus the latter mainly involves searching the resepctive script. Formulas, however, are part of the object-language, and their generation is controlled by the proof-state. Thus resuming from input formulas will mainly involve the proof-tree.

The problem to solve is the following:

1. given:
   (a) the proof-state (P, f) with the given formula f on the work-sheet
   (b) the script s with the set of executed tactics E
   (c) a formula $\widehat{f}$ input by the user
2. find: a derivation from (P, f) to $(P', \widehat{f})$, i.e. a list of tactics $[r_1, \cdots, r_m$ with $f \ \cdots \overrightarrow{r_m} \ \widehat{f}$.

The solution presented below to this problem is, quite in contrary to resume from input tactics, conceptually simple but costly in computation.

Definition 2.4.8: Let (P, f) be a proof-state, s a script with the set of executed tactics according to (P, f), D a canonical simplifier, F a set of formulas, R a set of tactics, and $\widehat{f} \in F$ a formula. $\widehat{f}$ is **locatable** in (P, f) iff

$$\exists\, r_i \in R,\ f_i \in F.\ 1 \leq i \leq m \Rightarrow f \xrightarrow[r_1]{} f_1 \cdots \xrightarrow[r_m]{} f_m \ \wedge \ f_m \approx_D \widehat{f}$$

where the tactics are found by the locate-function on s, i.e. $r_{i+1} = \mathcal{O}\, r_i$ and the formulas $f_i = \mathcal{E}_{Pf}\, r_i$ are generated by the execution-function on (P, f). A formula $\widehat{f} \in F$, locatable in (P, f), is called **located** in (P, f) iff the respective equivalence $f_m \approx_D \widehat{f}$ had been used to establish a new proof-state $(\widehat{P_m}, \widehat{f})$ by adding to (P_m, f_m) proof-objects generated by the list of tactics

$$[\ \mathit{rewrite}\ t_{11},\ \cdots,\ \mathit{rewrite}\ t_{1k},\ \mathit{rewrite}\ \overleftarrow{t_{2l}},\ \cdots,\ \mathit{rewrite}\ \overleftarrow{t_{11}}\ ]$$

where the theorems $t_{ij} \equiv (lhs_{ij} = rhs_{ij})$ and $\overleftarrow{t_{ij}} \equiv (rhs_{ij} = lhs_{ij})$ (note the symmetric versions, with *rhs* and *lhs* exchanged, and in reversed order, in the second half of the list) stem from the respective canonical simplification

$$\begin{array}{ccccc} f_m & \xrightarrow[rewrite\ t_{11}]{} & f_m^{(1)} & \cdots & \xrightarrow[rewrite\ t_{1k}]{} f_m^{(k)} \\ \widehat{f} & \xrightarrow[rewrite\ t_{21}]{} & \widehat{f^{(1)}} & \cdots & \xrightarrow[rewrite\ t_{1l}]{} \widehat{f^{(l)}} \end{array}$$

with the respective normal-forms $f_m^{(k)} \equiv \widehat{f^{(l)}}$.

The concept of locating an input formula $\widehat{f}$ in a given proof-state is simply calculating the example, and comparing each resulting formula for equivalence with $\widehat{f}$. This eventually may be done until the final result is reached – why not should the ME accept the correct final result ? It is the task of the DG, to decide whether to reject this input (asking for some intermediate steps as a justification, or suspecting the student to have been cheating in an examination) !

The equivalence of the formula f_m found by the system, and of the formula $\widehat{f}$ input by the user, cannot be literal equality, as already mentioned. The equivalence is modulo a simplifier; thus it is a crucial difference, whether a *canonical* simplifier exists in a domain or not. The existence of such simplifiers is surveyed in 5.3.

The equivalence $f_m \approx_D \widehat{f}$ is not only shown (by equivalence w.r.t. D, $f_m \approx_D \widehat{f}$). Moreover, in order to have each formula in a calculation derived from some proof-state, a derivation from f_m to $\widehat{f}$ is required. Def.2.4.8 constructs this derivation from the check for the equality of the normal-forms, $f_m^{(k)}$ and $\widehat{f^{(l)}}$ respectively: the rewrites for $\widehat{f}$ are 'inverted' by altering the direction of the rewrite-rule, which is justified by the law of symmetry, $(a = b) \ = \ (b = a)$.
The derivation, i.e. the lists of rewrites, can be cut if their tails are equal: if we have

$$\begin{array}{ccccccc} f_m & \xrightarrow[rewrite\ t_{11}]{} & f_m^{(1)} & \cdots & \xrightarrow[rewrite\ t_{1k'}]{} & f_m^{(k')} & \cdots \xrightarrow[rewrite\ t_{1k}]{} f_m^{(k)} \\ \widehat{f} & \xrightarrow[rewrite\ t_{21}]{} & \widehat{f^{(1)}} & \cdots & \xrightarrow[rewrite\ t_{1l'}]{} & \widehat{f^{(l')}} & \cdots \xrightarrow[rewrite\ t_{1l}]{} \widehat{f^{(l)}} \end{array}$$

with

$$t_{1k'+1} \equiv t_{1l'+1},\ \cdots,\ t_{1k} \equiv t_{1l}$$

then, provided $f_m^{(k)} \equiv \widehat{f^{(l)}}$, this implies that all formulas in the tail are equal

$$f_m^{(k')} \equiv \widehat{f^{(l')}},\ \cdots,\ f_m^{(k)} \equiv \widehat{f^{(l)}}$$

and the following list of rewrites is sufficient for establishing the proof of equivalence, $f_m \approx_D \widehat{f}$, in the proof-tree:

$$[\ \mathit{rewrite}\ t_{11},\ \cdots,\ \mathit{rewrite}\ t_{1k'},\ \mathit{rewrite}\ \overleftarrow{t_{2l'}},\ \cdots,\ \mathit{rewrite}\ \overleftarrow{t_{11}}\ ]$$

2.4.7 Summary and related work

This section developed the language of scripts and its interpreter. Scripts describe how to apply tactics in order to solve the problem specified. Tactics propagate the proof-state, while details of the proof-tree are not visible in the script. Tactics which can be done in parallel, are modeled by disjunct arguments of functions (in the script as functional language), and encoded by let.

The interpreter of the scripts is designed for reactive user-guidance, i.e. before a tactic is being applied the control is passed to the user, and the interpreter tries to resume guidance after input of any tactic which could be done in parallel at the current proof-state.

The algorithms of the script-interpreter are impaired by a limitation of the program language used for implementation: SML does not support parallel processes. [14] Thus the interpreter-state needs to be saved at any 'user-interrupt', and has to be re-established if the user passes control back to the system. A more natural design would (at least) model the dialog-component and the script-interpreter as parallel processes.

Some points of incompleteness are to be noted:

(1) There is no decision yet, which kinds of nesting of script-expression should be excluded. This decision is related to the notion of *helpless* (Def.2.4.4): If one allows a `try` as the out-most script-expression in the body of a script, then the interpreter-state never can become *helpless*, because a 'no applicable tactic' (*napp*) is always turned into *finished* – a behavior which is not desirable.

(2) The consequences of data dependencies in a parallel let (p.87) are not yet clear: The algorithm proposed (Def.2.4.4) re-executes the whole expression of parallel lets on the fly using the list of executed tactics. This algorithm is neither efficient nor proven consistent. Data-flow analysis would be appropriate.

(3) There is no general strategy how to handle applicable tactics on the search $\mathcal{O}$ locating a user-tactic in the script (Def.2.4.7). Only two extreme cases are clear yet: either all tactics are concerned with rewriting (then the

[14] There is a concurrent extension of SML at `http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/doc.html`. But it is impossible to compile Isabelle within this system.

intermediate proof-states are dropped) or otherwise all intermediate proof-states are taken valid.

The major problem with this summary is, that the author sees himself forced to ask the reader to accept - just in this central point of the thesis - the exceptional situation, that no serious reference to related work can be given.
For a non-expert the interpreters task could be compared to the following situation: Given a C^{++} program, eventually containing several threads, running in a debugger. Imagine the program is halted by a breakpoint at the statement $x = x + 1$, and the control is given to the user. The user declares another statement, say $x = h - x$, as to be executed next, and the system has to find out whether this makes sense w.r.t. some constraints, and where to resume execution in the C^{++} program accordingly.
This example is not adequate, neither in analogy (the C^{++} program delivers values, whereas a script delivers tactics, i.e. parts of itself), nor in making sense: The task of locating statements and resuming after assignment of an arbitrary C^{++} statement, does not make any sense. The main difference is, that the semantics of a script is heavily influenced by an instance apart from the scripts language, by the proof-tree.
The proof-tree is consulted for the selection of a tactic (in an `or`, a `repeat`, or a `try` expression of a script). The proof-tree is the source of the value, a tactic delivers; these values become the output to the user. On input of the user, the proof-tree is the first instance deciding upon the applicability of a tactic, before the interpreter tries to locate an associated tactic in the script. The input of a formula requires an even closer cooperation of proof-tree and script.
The three concepts (1) of functional language, (2) of a proof state established by a cooperating deduction machinery, (3) together with resuming from user input, seems to have been combined, to the authors knowledge, for the first time by this thesis; and this special combination leads to a very special purpose language, which is hardly comparable to other languages.

Thus the authors conclusion may be conceded, that there is no special related work to refer to (where the basic techniques used for the interpreter are not considered worthwhile to be mentioned separately).

Chapter 3

REACTIVE USER-GUIDANCE, AND SYSTEM-ARCHITECTURE

This chapter comprises two rather independent sections, the introduction of the dialog component, and the architecture of the overall system.

The dialog-component is extremely separated from the mathematics-engine (ME) introduced in the previous chapter. An author can add mathematics knowledge without being concerned with the dialog at all. This is a great difference to most of other authoring systems !
The interaction between dialog-component and ME is based on a flexible and simple mechanism, featured by symmetric dialog-atoms. *These atoms are symmetric w.r.t. the student and the tutor (the tutoring-system) as partners on an equal base.*
The mechanism on the dialog-atoms is already sufficient to make the prototype work. The major advantages are left open for future development: the dialog-atoms can be combined to 'dialog-patterns' handling typical situations in a dialog.
The symmetric dialog-atoms are claimed to be original work. They allow to configure dialog modes as different as 'stepwise presentation of a calculation by the system' (where the student is passive consumer) or 'perform a calculation in an examination' (where the student works independently).

The system-architecture is presented without discussion and references; the presentation may be regarded as brief documentation of the prototype, and as a prerequisite for estimations on future development, done in the conclusions 4.3. A survey on the knowledge representation is given, and views onto the different kinds knowledge are related to the needs of tutoring and authoring.

3.1 A dialog model for rule based systems

This section develops the feature most essential for the overall functionality of the tutor, the dialog model. The model consequently exploits the existence of a mathematics-engine (ME) capable of acting as an autonomous partner of the student (the ME is able to solve an example on its own, and even can continue an (adequate) calculation proposed by the student).
The partnership is realized by symmetric dialog-atoms, symmetric w.r.t. the interactions of the student and the tutor respectively. The design of this symmetry is claimed as original work.
The model developed so far is presented as a flexible basis for further amendments concerning high-order dialog-patterns and a user-model.

The dialog with a mathematics tutor comprises the wealth of math structures in a calculation, information retrieval from the knowledge base, commands concerning the mode of the dialog, and commands for handling the system. Let us omit commands for handling the system, postpone information retrieval from the knowledge base to 3.2.2, and begin with the most interesting part, the dialog on constructing a calculation on the work-sheet.

3.1.1 The dialog universe

The first decision in approaching the complexity of dialogues in a mathematics tutor is, not to burden the system-component guiding the dialog (DG) with mathematical semantics of a calculation. For this reason the input and output of the math-engine (ME) is abstracted to a model suitable for rule based systems in general. What is called a 'rule-based system' here, is defined comparatively informal by the following input-output-relation:

Tab. 3.1: Input - output of a rule-based system

input	output
tactic	resulting *formula* + next *tactic* or *error*-message
formula	assigned *formula* + next *tactic* or *error*-message

This model consists of the object-types *tactic, formula* and *error*. The individual objects are all predefined, eventual (variable) content of an object is wrapped in the predefined identification, and thus hidden to this model. The internal state by such a rule-based system only becomes visible by the

alternative on the output (a formula plus tactic *or* an error), and by an eventual modification of an input formula to an *assigned* formula.

The mathematics-engine (ME) developed in the preceding chapter may be regarded as such a rule-based system: tactics, as introduced in 2.3.3, may be regarded only with respect to their identifier, neglecting the arguments. The proof-state decides for the output responding to an input tactic: Def.2.3.7 either classifies the tactic 'applicable', then a resulting formula and a next tactic are output; otherwise an error-message is returned.
And formulas of the math object-language, as discussed in 2.1.3, may be regarded as one single kind of object, neglecting any syntactic details. The *assigned* formula may differ to the input formula by its indentation on the worksheet.
Error-messages need not be distinguished in this model, too.

With respect to this model, the design of the dialog becomes manageable. One thing to be added are commands and messages concerning the dialog itself. Thus the universe of the dialog consists of five kinds of objects:

1. tactics: their syntax is defined in 2.3.3, their semantics is neglected by the DG – tactics pass the DG unchanged, or may be stopped on their way from the ME to the user (but never in the opposite direction); for the latter purpose the DG may use a tactics identifier for assigning it to the respective class. W.r.t. the semantics the DG completely relies on the ME, and reacts only due to the MEs feedback given by *error* or not-*error*.

2. formulas: are distinguished from tactics by their appearance; syntax and semantics are completely neglected – formulas pass the DG unchanged, or may be stopped on their way from the ME to the user (but never in the opposite direction). W.r.t. the formulas' semantics the DG completely relies on the ME, and reacts only due to the MEs feedback given by *error* or not-*error*.

3. errors: comes from the ME indicating that the preceding input of a formula or a tactic does not promote the calculation. Beyond this information an error has no semantics, an error passes the DG unchanged (and is never stopped).

4. commands: allow the user to guide the flow of the dialog; the syntax for commands is

$$
\begin{array}{lllll}
\mathit{command} ::= & \mathit{Accept} & \mid \mathit{NotAccept} & & \\
\mid & \mathit{YourTurn} & \mid \mathit{Tactics} & \mid \mathit{Details} & \\
\mid & \mathit{HowCome} & \mid \mathit{WhatFor} & \mid \mathit{DontKnow} & \\
\mid & \mathit{ActivePlus} & \mid \mathit{ActiveMinus} & &
\end{array}
$$

| *WidthPlus* | *WidthMinus*
| *Undo*

The semantics of the commands will be described below.

5. messages: go from the DG to the user; their semantics essentially is equivalent to the commands, just the other way round.

Thus the syntax of the input-language for the student S and the output-language of the tutor T is given by the BNF as follows:

$S ::=$ *tactic* | *formula* | *command*
$T ::=$ *tactic* | *formula* | *message* | *error*

where *tactic* is defined in 2.3.3, *formula* in 2.1.3, and *command* above.

The flow of the objects, as described above, between user (i.e. the presentation-module, the DG and the ME is depicted in Fig.3.1. The division into the three modules presentation, dialog, and application (ME) is according to the Seeheim-model [Pfa85]. The semantics of the commands is sketched as follows.

Fig. 3.1: The flow of the dialog objects

Accept, NotAccept: The user accepts the tutors suggestion (a tactic to apply, a formula to insert in the work-sheet, etc.) or not.

YourTurn: The user rejects a request of the tutor (for the input of a tactic or a formula). The inverse command *MyTurn* is not necessary, because the user is free to suggest a tactic or a formula as the next step at any time.

Tactics: The tutor shall present a list of tactics (eventually) applicable to the current proof-state.

Details: The tutor shall give more information about the current tactic or formula in the work-sheet, or the tutor shall reveal parts of the calculation hidden in folded nesting.

HowCome, WhatFor: Ask the tutor to look back, or ahead, and give reasons for the current step.

DontKnow: The user refuses any action.

ActivePlus, ActiveMinus: Direct access to the variable of the dialog-state deciding on the next dialog-atoms chosen. Active means, the user does steps of calculation himself; the less the tutors help, the more active; and vice versa.

WidthPlus, WidthMinus: Direct access to the variable of the dialog-state deciding on the width of the steps in the calculation.

Undo: Undo.

Having introduced the syntax available in the dialog-universe, and a sketch of the semantics of the universes objects, the next step is to construct the mechanisms implementing that flexible kind of dialog as described in the requirements 1.3.4. Given the ME, which 'knows' what the next steps are (even after the student has done some - successful - steps), the idea is straightforward to view student and tutoring-system as partners on an equal basis. The consequence of this view is, that the dialog can be built up by symmetrical elements. This is done below.

3.1.2 Symmetric dialog atoms

Dsteps, short for 'dialog-steps', map a formula f to a formula f'. There are two intermediate steps in-between f and f' which we call *dphase*s in general: (1) determine a tactic t applicable at f, (2) determine the tactics arguments a, and (3) determine f' resulting from application of $(t\ a)$, i.e. complete the dstep itself. We use BNF, non-terminals start with a lower-case letter, terminals start with an upper-case letter:

dstep ::= putRes
| *fillRes*
| *((putTactic*
| *selTactic*
| *fillTactic)*
| *((putTac*
| *selTac) (putArg*

| *fillArg*)) (*putTacticRes*
| *fillTacticRes*))

dphase - - - - *(1)* - - - - |- - - *(2)* - - -| - - - *(3)* - - -

The parentheses show that a dstep can be done in *one* go (by 'putRes' or 'fillRes'), or by two intermediate steps, e.g. 'putTactic' – 'putTacticRes', or at most by three intermediate steps, e.g. 'putTac' – 'putArg' – 'putTacticRes'. We call the intermediate steps 'datoms'.

Datoms, short for 'dialog atoms', can be performed by one of the two dialog partners, the student S or the tutor T (i.e. the tutoring system). In order to describe the partner we extend BNF: $datom\langle S,T\rangle$ means that S is the 'active partner', with $datom\langle T,S\rangle$ the active partner is T, and $datom\langle a,b\rangle$ can take a as active partner S or T [1] . Using this notation, and connecting the partner and the action performed by ':', we describe the interactions following the wellknown hand-shake structure *request – data – acknowledge*:

putRes ⟨ *S, T* ⟩ ::=
T: "input the resulting formula"
S: (Form | dBreak)
T: (NotAccept MEmsg | ϵ)

putRes ⟨ *T, S* ⟩ ::=
S: ϵ
T: Form
S: (Accept | aBreak)

fillRes ⟨ *S, T* ⟩ ::=
T: "fill the gaps in the current formula"
S: (Form | dBreak)
T: (NotAccept MEmsg | ϵ)

putTactic ⟨ *S, T* ⟩ ::=
T: "apply a tactic to the current formula"
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

putTactic ⟨ *T, S* ⟩ ::=
S: ϵ
T: Tactic
S: (Accept | aBreak)

selTactic ⟨ *S, T* ⟩ ::=
T: SelectList "select a tactic for application to the current formula
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

fillTactic ⟨ *S, T* ⟩ ::=
T: "fill the gaps in the tactic to apply"
S: (Tactic | dBreak)

[1] This notion of 'active' is related to the commands *ActivePlus, ActiveMinus.*

T: (NotAccept MEmsg | ϵ)

putTac ⟨ S, T ⟩ ::=
T: "input a tactic for application to the current formula"
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

putTac ⟨ T, S ⟩ ::=
S: ϵ
T: Tactic
S: (Accept | aBreak)

selTac ⟨ S, T ⟩ ::=
T: SelectList "select a tactic for application to the current formula
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

putArg ⟨ S, T ⟩ ::=
T: "complete the tactic for application to the current formula"
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

putArg ⟨ T, S ⟩ ::=
S: ϵ
T: Tactic
S: (Accept | aBreak)

fillArg ⟨ S, T ⟩ ::=
T: "fill the gaps in the tactic to apply"
S: (Tactic | dBreak)
T: (NotAccept MEmsg | ϵ)

putTacticRes ⟨ a, b ⟩ ::= putRes ⟨ a, b ⟩
fillTacticRes ⟨ a, b ⟩ ::= fillRes ⟨ a, b ⟩

The datoms serving dphase (1) rely on the precondition, that the current formula (i.e. the *given* formula in the proof-state) is being displayed.
Most of the datoms can have either, S or T, as an active partner. The exceptions are 'fill_' and 'sel_': it does not make sense for the student to ask the system: S: "fill the gaps in ...", and it is the same with S: (SelectList "select a ... for application").

Chaining of the datoms concerns the question: Given the hand-shake structures T–S–T (where S is active) and S–T–S (where T is active), how do we get a chain of interaction ...–S–T–S–T–... ?

Looking at these structures, alteration of activity with each datom seems to be the only straight-forward possibility of chaining; however, this case

should be *one* possibility among others. The complete collection of cases in chaining is

1. T remains active
2. S remains active
3. change from T active to S active, initiated by T
4. change from T active to S active, initiated by S
5. change from S active to T active, initiated by T
6. change from S active to T active, initiated by S

We show in the sequel: *all cases of chaining can be constructed* by the mechanisms of skipping, combining and pattern-breaking.

Skipping the students requests and acknowledges: [2] The reader may have noted, that all datoms with T active are defined with an empty request for the student, $S : \epsilon$. This is consistent with the view of the tutor (i.e. the tutoring-system) as a partner of the student on an equal base: the tutor is expected to cooperate actively – why should the student request for an activity of the tutor in general ? The student does what she wants, and the tutor reacts (eventually with a request to the student).
This design-decision for the datoms allows to construct chaining case (1.):

$$\cdots - \; T_{i-1} \; - \overbrace{\; S_i \; - \; T_{i+1} \; - \; S_{i+2}} \; - \underbrace{\; T_{i+3} \; - \; S_{i+4}} \; - \cdots$$

The acknowledge of the preceding datom in S_{i+2} is combined with the request in the subsequent; this is consistent with the empty student request, $S_{i+2} = \epsilon$.

Skipping is the general mechanism to make the dialog more fluent. Consider the example:

	$\ldots A_1 = 2(2r\sin\alpha)b - (2r\sin\alpha)^2)$
putTactic $\langle$ *T, S* $\rangle$ ::=	*S:* ϵ
	T: we substitute $b \mapsto 2r\cos\alpha$
	S: ϵ
putRes $\langle$ *S, T* $\rangle$ ::=	*T: "input the resulting formula"*
	S: $2(2r\sin\alpha)(2r\cos\alpha) - (2r\sin\alpha)^2)$
	T: Accept

[2] Skipping relates to the commands *WidthPlus, WidthMinus.*

There are two places S:ϵ where the students input is being skipped. The first one is at the beginnin of a datom with T active. In this case we *never* wait for a *request* by the student.
The second one at the end of a datom with T active is an option the DG may choose if the current dialog-state indicates to proceed quickly in the dialog. The DG is even free to decide not to show certain dsteps, or to collect the data from several datoms with T active into one single output. For instance in continuing the example from above, the DG may collect

	$\ldots A_1 = 2(2r\sin\alpha)(2r\cos\alpha) - (2r\sin\alpha)^2)$
putTactic $\langle$ *T, S* $\rangle$ ::=	*S:* ϵ
	T: subproblem $(\mathcal{R}, \mathit{derivative}, \_)$
	S: ϵ
putRes $\langle$ *T, S* $\rangle$ ::=	*S:* ϵ
	T: $\frac{d}{d\alpha}(2(2r\sin\alpha)(2r\cos\alpha) - (2r\sin\alpha)^2))$
	S: ϵ
selTactic $\langle$ *S, T* $\rangle$ ::=	*T: ("diff_boundvar",* $\frac{d}{dx}x = 1$*)*
	("diff_constant", $\frac{d}{dx}c = 0$*)*
	("diff_sum", $\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$*)*
	...
	"select a tactic for application
	S: ...

to one interaction

selTactic $\langle$ S, T $\rangle$ ::=	S:ϵ
	T: subproblem $(\mathcal{R}, \mathit{derivative}, \_)$
	$\frac{d}{d\alpha}(2(2r\sin\alpha)(2r\cos\alpha) - (2r\sin\alpha)^2))$
	("diff_boundvar", $\frac{d}{dx}x = 1$)
	("diff_constant", $\frac{d}{dx}c = 0$)
	("diff_sum", $\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$)
	...
	"select a tactic for application
	S: ...

This is already an extended example for the following mechanism:

Combining the tutors acknowledges and requests This is the means to construct chaining case (2.):

$$\cdots - \; S_{i-1} \; - \overbrace{\; T_i \; - \; S_{i+1} \; - \; \underbrace{T_{i+2}}} \; - \; S_{i+3} \; - \underbrace{\; T_{i+4}} \; - \cdots$$

The student is active, according to the tutors requests; and the tutors acknowledge of the preceding datom in T_{i+2} is combined with the request of the subsequent datom.

Braking datoms At any S the student is free to input arbitrary commands, tactics or formulae at any dstep displayed on the screen. Thus in general, the DG has to expect the datom to be broken; the datoms defined above have the following structure in common:

$$\begin{array}{ll} \mathit{datom}\ \langle\ S,\ T\ \rangle\ ::= & \mathit{T}\text{: (}\ \mathit{Msg}\ \mid\ \mathit{SelectList\ Msg}\ \text{)} \\ & \mathit{S}\text{: (}\ \mathit{Form}\ \mid\ \mathit{Tactic}\ \mid\ \mathit{dBreak}\ \text{)} \\ & \mathit{T}\text{: (}\ \mathit{NotAccept\ Msg}\ \mid\ \epsilon\ \text{)} \\ \\ \mathit{datom}\ \langle\ T,\ S\ \rangle\ ::= & \mathit{S}\text{: }\epsilon \\ & \mathit{T}\text{: (}\ \mathit{Form}\ \mid\ \mathit{Tactic}\ \text{)} \\ & \mathit{S}\text{: (}\ \mathit{Accept}\ \mid\ \mathit{aBreak}\ \text{)} \end{array}$$

The *dBreak* and *aBreak* mark the points where the student can input anything, where in *dBreak* the datom would have expected data from the student, and in *aBreak* the datom would have expected an acknowledge.

Equipped with the mechanism of pattern-braking, chains including alteration of activity by the student can be constructed as follows. Case (4.) in chaining is

$$\cdots - \ T_{i-1}\ - \overbrace{S_i\ - \underbrace{T_{i+1}\ - \underline{S_{i+2}}_a}\ - \ T_{i+3}}\ - \ S_{i+4}\ - \cdots$$

where the hand-shake structure of the first datom expects an acknowledge (or at least some command) in S_{i+1}, the student however inputs a tactic or a formula. This requires the DG to enter the datom for $\mathit{putTactic}\langle S,\ T\rangle$ (or $\mathit{putArg}\langle S,\ T\rangle$ depending on the dphase) or to enter the datom for $\mathit{putRes}\langle S,\ T\rangle$ respectively.

Case (6.) concerns a datom with the student active; thus the student is expected to input a tactic (or its argument dependent on dphase) or a formula. However, the student breaks the datom by input of a command, for instance *YourTurn* or *DontKnow*:

$$\cdots - \ S_{i-1}\ - \overbrace{T_i\ - \underbrace{\underline{S_{i+1}}_d\ - \ T_{i+2}}\ - \ S_{i+3}}\ - \ T_{i+4}\ - \cdots$$

Chains where the tutor alters activity concern case (3.) and (5.). In case (3.) the tutor is active in the preceding datom, and the DG decides not to output a tactic (or a formula dependent on the datom), but to shift activity to the student by *YourTurn* combined with a request entering the subsequent datom. That means that this case (3.) is covered by a special kind of datoms:

$$\cdots - \; T_{i-1} \; - \overbrace{\; S_i \; - \underbrace{\underline{T_{i+1}}_d \; - \; S_{i+2}}} \; - \; T_{i+3} \; - \; S_{i+4} \; - \cdots$$

The last case (5.) occurs very naturally if the ME responds with *error* to a tactic or formula input by the student in S_{i+1}; the DG passes this error-message through to T_{i+2}, and eventually gives the correct result in T_{i+2}, too:

$$\cdots - \; S_{i-1} \; - \overbrace{\; T_i \; - \underbrace{S_{i+1} \; - \; T_{i+2}}} \; - \; S_{i+3} \; - \; T_{i+4} \; - \cdots$$

Another decision of the DG could be, to request another trial from the student; this would be chaining case (2.).

This completes the demonstration how all cases (1.)...(6.) of chaining datoms can be constructed. The possibility to freely compose datoms raises the question which datoms should be chained in which situation. This question will be answered within the next subsection.

3.1.3 Chaining atoms for reactive user-guidance

Reactive user-guidance is constituted by the cooperation of DG and ME, which is comparably simple due to the allocation of the respective duties: The DG chooses datoms according to the dphase, the students input (tactic/formula/command) and output of ME (tactic/formula/error). The ME checks the students input passed through the DG and responds with (tactic + formula) or error. This simple structure becomes possible by the mechanisms of *skipping* and *combining* introduced above. Let us give an example in Tab.3.2.

This example shows that the structure of the dialog may be rather blurred in the presentation (all S:ϵ are skipped, and thus *three* consecutive T: are combined (T: Accept, result f_i, "apply a tactic"). What the user gets presented in the dialog above, is:

Tab. 3.2: Cooperation of DG and ME

presentation	dialog-guide (DG)		math-engine (ME)
...	...	...	...
T: "apply a tactic" S: r_1 T: Accept	putTactic ⟨S,T⟩	$\longrightarrow$ $\longleftarrow$	check,apply r_1 f_1, r_2
S: ϵ T: result f_1 S: ϵ	putTacticRes⟨T,S⟩		
T: "apply a tactic" S: r_2' T: Accept	putTactic ⟨S,T⟩	$\longrightarrow$ $\longleftarrow$	check,apply r_2' f_2, r_3
S: ϵ T: result f_2 S: ϵ	putTacticRes⟨T,S⟩		
...	...	...	...

T: ..., "apply a tactic"
S: r_1
T: result f_1 *Accept*
"apply a tactic"
S: r_2
T: result f_2
...

The *internal* expansion of the dialog to datoms is the key idea which then provides for a well-structured and flexible interaction between DG and ME.

The design-decision for *skipping* and *combining* leads to overlapping datoms, instead of the need for glue between datoms. Thus a simple list of the datoms executed already creates a history, which describes very well what actually happened in the dialog.

A history maintains the information necessary to design some kind of *dialog-patterns* consisting of a special combination of datoms for special purposes, for instance for reacting on common errors. Such dialog-patterns have not yet been designed; they should be reflected in the dialog-state and in a user-model not yet designed, too.

The dialog-state, which makes the prototype working, consists of a list of tactics to be done hidden from the user, the dphase, the current datom and the history. From the history two integer values are extracted, the 'activity' and the 'step-width'. In order to influence the DGs selection of datoms,

these values can be set by the user – a primitive choice, not satisfactory in future versions.

3.1.4 Summary and related work

This section developed a comparably informal dialog model which exploits the existence of the mathematics-engine capable of autonomously solve problems, eventually continuing the users solutions. This dialog-model views the student and the tutoring-system as partners on an equal base.

The model provides for highly flexible dialogs by expanding the externally visible interaction to internal dialog-atoms built upon the hand-shake mechanism. These dialog-atoms are symmetric w.r.t. the two partners, they can be chained to manifold patterns, and they are generally usable for rule based systems.

The symmetric dialog-atoms form the basis for future development of high-order dialog-patterns, hand-in-hand with the development of a user-model. A formalization of the dialog-model might lead to a kind of algebra of dialog-atoms.

The search for related work, related to the symmetric dialog-atoms, is successful already in the stormy decade of artificial intelligence – [Pre71] coins the notion of 'balanced man-machine systems' at the beginning of the seventies. A balanced system is 'where both partners are active, where either may suggest operations or execute them, and where the order of the steps in the problem-solving process is jointly determined'. Such a system has been shown to be useful for the analysis of multivariate data, already at that time. To the best knowledge of the author, the term 'balanced man-machine system' has not been used any more.

A recent survey on human-computer interaction technology [Mye98] identifies six up-and-coming areas, five of which are concerned with a 'tool based paradigm': gesture recognition, multi-media, 3-D, virtual reality, natural language and speech (the sixth area is computer supported cooperative work, CSCW, to be discussed later). The tool based paradigm, of which direct manipulation is an example, has become synonymous with many of todays graphical user interfaces. Direct manipulation interfaces aim to amplify the user's knowledge of a domain by allowing them to *think in familiar terms*, rather than those of the computational medium. They present the user with the illusion of interaction in an environment populated by familiar task related tools which reflect their goals and intentions, and they fail to utilize the potential of technology to provide novel ways to think of, and interact with, a task domain [Hut89].

Thus the search for related work has to leave the main-streams of HCI, where an explicit 'cooperative interaction paradigm' is being pursued in sev-

eral sub-domains and application areas. The paradigm is nicely described by [3] as undertaking which 'seeks to integrate the benefits associated with direct manipulation and agent-based approaches by enabling a more equitable division of labour between the user and technology. It is proposed to achieve this by viewing human computer interaction as a cooperative partnership between the human and the computer and by the explicit acknowledgment that each bring different but necessary skills to the task domain. For example, repetitive or computationally intensive aspects of a task could be assigned to the system, thereby releasing the user to engage in more qualitative aspects of the task'.
Let us scan the few references which are explicitly dedicated to this paradigm in their respective field of application.

Computer supported cooperative work (CSCW), i.e. CSCW-references use 'cooperative' in a way misleading our search. A major conference on [4] describes its dedication as 'to contribute to the solution of problems related to the design of cooperative systems, and to the integration of these systems in organizational settings'. A look at the publications there shows that the tool based paradigm [Hut89] is prevalent. An exception is [DFR93] who pursue the idea that the design of better adapted intelligent systems can benefit from a better knowledge of the natural cooperative behavior between humans. Two studies of natural situations of cooperation are presented, focusing on the type of intervention of each partner in the dialogue.
This work pays specific attention to the methodology of analysis, and does not construct any model.

Knowledge based systems (KBS) are another candidate to be approached by the paradigm, because both, the system and the user have (some kind of) knowledge. [KSF93] states, that such systems incorporate the user in the problem solving and decision process. As the user guides and participates in the problem solving, it becomes essential to turn the simple communication with the system into a real cooperation between the two partners'. The work proposes a Man-System Cooperation Model (MSCM) and a new software architecture model for Cooperative KBS.
These models are much more abstract than the concrete model presented in this thesis; thus they hardly can be compared. A severe additional difference is, that a KBS is not meant to find a solution without the users intervention. The tutor, however, is meant to be able to demonstrate on its own, how an example can be solved.

[3] http://www.dcs.napier.ac.uk/∼michael/phd/summary.html
[4] COOP, http://www-sop.inria.fr/acacia/Coop/Coop2000/

Computer aided design (CAD) as a tool for technicians is a little surprising application area. [CS93] developed a prototype system in order to investigate the utility of cooperation as a novel paradigm for human computer interaction. The task domain selected in order to exemplify the approach was spatial design. As part of the development cycle the prototype was evaluated by two expert designers. Their responses were generally favorable concerning the nature of the interaction and in particular the fact that the system acted as a prompt to their own imaginations during problem solving. However, the poverty of the underlying rule base resulted in the system providing the designers with overly simplistic design alternatives. The fact was considered to be the single most important failing of the prototype system as the feature was critical to the cooperative paradigm.
Again, the relation of this work to the thesis' dialog model is weak. Spatial knowledge and spatial reasoning is too different from mathematical knowledge and reasoning.

Thus, the final conclusion is, that no reference in HCI covers the dialog model presented in this thesis.

3.2 System-architecture

The system-architecture is described as already provided for the prototype implementation, and related to organizational and technical requirements.
The math knowledge is recalled after their introduction in the previous chapter, and surveyed together with all other kinds of data, an author has to provide, and to implement into certain system-components.
The various views onto the knowledge is listed and related to their respective use by students and by authors. Finally special requirements, to be met for authoring, are collected.

3.2.1 A web-based multi-user system

Within the last two decades *personal* computing pushed forward the development of hardware and software; during the last five years the web took over the technology-driving rôle. This rôle applies to educational software, too, as has been documented by [Aus00] recently.

When designing a mathematic tutor it is advisable to take into consideration the advantages of both, of the decentral concept of personal computing, and of the centralized concept of a web-based service. This consideration concerns the following points:

1. The availability of such a software-system for anybody, at any place, any time, is the first point, realizable by both, a stand-alone application on a PC, or a web-based service.

2. The availability for students concerns classroom activities of different kinds, homework, and distance learning; there are also persons interested in mathematics, however, not organized in some educational institution.

3. The availability for teachers and course designers concerns adding lessons for their courses, eventually adding new knowledge for the system mastering a new course, administering the courses, analyzing the students progress, and performing examinations.

The first two points indicate a decentral concept, the last a concept centered within an educational institution. Free down-load of the software for educational purposes is desirable. The following points indicate a centralized, web-based concept:

1. Mathematics tutoring requires much computational power. A typical Isabelle session (without tutoring system) involves a core image inbetween 50MB and 100MB. Presently this exceeds the computational power of a typical PC at high-schools.

2. All users should take advantage of already implemented knowledge. As mathematics knowledge is structured hierarchically, adding new example collections would not require adding new knowledge after some years of summarized efforts.

The last point is the most important: If there are stand-alone versions of the authoring-system, then there must be a well-organized team for integrating the individual developments into new releases. Isabelle [5] is a good example how to organize such a development.

The prototype of the tutor has been designed with respect to the points above. The prototypes architecture is depictured in Fig.3.2. Its structure is

Fig. 3.2: The components of the prototype

according to the Seeheim-model [Pfa85].

The presentation component is given by the note-book *NB* for each user. The term note-book refers to the kind of user-interface introduced by Mathematica [Wol96].

The dialog component is given by the dialog-guide DG holding the dialog-states *DS1...DSn* for each user respectively.

The application is given by the mathematics-engine ME operating on Isabelle, and maintaining proof-states *PS1...PSn*. ME uses Isabelle as stateless knowledge-base.

[5] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/

The middled-ware, not included in the Seeheim-model, is represented by the bridge , which provides for networking and for the connection between Java and SML.

Java is the language of choice as for many inter-net-applications, and SML is the language used for the implementation of the theorem prover Isabelle, and consequently used for the tutors implementation. SML is efficient enough to implement the multi-user system.

3.2.2 Survey on the knowledge representation

Within the development of the mathematics-engine various kinds of mathematics knowledge have been introduced. Fig.3.3 depictures all kinds of math knowledge. In the sequel all other kinds of knowledge are surveyed,

Fig. 3.3: Interfaces of the mathematics engine

too, in order to give an idea what 'instantiate the generic system' means for an author.

There are three groups of knowledge, (1) the mathematics knowledge, (2) the dialog-settings, and (3) the example collections.

Mathematics knowledge is represented by the 3D-universe Fig.2.1 on p.64 of domains, problem-types, and methods.

The domains are completely covered by Isabelle [Pau94] which itself is generic, with a kernel called 'Pure' capable only of inference based on

high-order natural deduction. In Isabelle the knowledge on domains is represented by so-called theory-files, which have an ML-file attached and which are organized in a hierarchy (actually a directed acyclic graph [6])

- Theory-files contain
  - function constants defined together with their arity, their type and eventual concrete syntax
  - definitions, including recursive, and mutual recursive functions, as well as (ev. recursive) data-type definitions
  - axioms.
  - One special theory-file contains function-constants for script-expressions Def.2.4.1 and tactics.
  - One special theory-file contains function-constants for descriptions Def.2.2.2 in problem-types.
- The attached ML-files contain
  - proof-scripts for theorems, by convention restricted to those based on the definitions in the related theory-file
  - ML-code for predicates (used in scripts and problem-types) where the function-constant has been defined in the respective theory-file
  - rule-sets containing theorems defined in the related theory or a parent theory. A rule-set contains
    - the list of theorems (rewrite-rules)
    - the list of calculations
    - the term-order to be used in case of ordered rewriting
    - the list of conditional rewrite-rules storing their conditions in the assumptions.
  - One special ML-file contains the code of the functions for numerical evaluation.

All these data are permanently held in the SML-core-image.

The problem-types give a description for a collection of examples which can be solved by one or more methods. For each example prepared for tutoring, a problem-type must be prepared, too, to be instantiated (Def.2.2.5) by the example. The problem-types are assembled in

- a problem-tree Def.2.2.7 induced by the refine-relation Def.2.2.6

[6] http://isabelle.in.tum.de/library/graph/data/HOL/large.html

- the problem-types with the fields *given, where, find, with, relate* as defined in Def.2.2.3
- for each problem-type a list of methods for solving this problem-type.

This problem-tree is stored in an ML-source-file, i.e. it is permanently held in the SML-core-image.

The methods solve a problem-type; they are guarded by a data-structure equal to problem-types. Thus each method consists of

- the script Def.2.4.1
- the guard equal to Def.2.2.3
- a (canonical) rule-set necessary for associating rules with tactics Def.2.4.2 and for locating a formula Def.2.4.8.

The methods are held in a list stored in an ML-source-file, i.e. they are permanently held in the SML-core-image.

Dialog-settings presently are a minimal version for the prototype, concerning

- activity, i.e. the DGs inclination to select dialog-atoms with the student or the tutor active 3.1.2
- step-width, i.e. the DGs inclination to skip requests to the student and to combine output of the tutor p.117
- a list of rules, which should be done only between DG and ME, i.e. hidden from the student
- a list of theorems, which should be used for rewriting only between DG and ME, i.e. hidden from the student
- a list of theorems to be hidden in repeated rewritings (e.g. the law of distributivity in $a \cdot (b + c + d + \cdots)$).
- a list of problem-types to be solved hidden from the student.

These few preliminary values are stored in the respective users dialog-state in the DG Fig.3.2, i.e. they are values of the SML-top-level environment. The future settings, which will be more abstract and closer to terms of cognitive psychology and of user-modeling, will be stored together with administrative data somewhere in the Java-part.

Example collections provide the student with stuff to exercise, and the tutor with hidden knowledge necessary to assist in the model- and specification-phase. Each example is given by three kinds of data in such a collection:

- the textual description, eventually augmented with a drawing, e.g. 11
- the hidden formalization Def.2.2.1
- the hidden specification Def.2.2.9.

Example collections are stored within the administrative framework in the Java-part of the system.

Summarizing the locations of the different kinds of knowledge presented above, the design follows Isabelles policy, and holds all knowledge in the core. The only, but essential, exception are the example collections, which are held in the Java-part and thus can be extended without limits.
As the mathematics knowledge may become more and more complete during the systems development, the run-time system (containing this knowledge) will become stable against adding example collections.

Multi-lingual usage of the system would require different versions of the SML-core. Language constructs visible for the user are contained in domains (Isabelle theories), including names of predicates in the problem-types (and names of tactics in the script, which does not matter). This kind of inflexibility has to be accepted with the choice for Isabelle. Another part of language-dependent constructs are requests, responses and messages delivered by the DG (at least for the time being); this part can be implemented multi-lingual.

3.2.3 Views for tutoring and authoring

In principle, a students access to knowledge should be as free as an authors access – empowerment is the best motivation for learning, and individual restriction only may protect from unnecessary confusion. Thus the discrimination between the students view and the authors view below is just for partitioning the presentation; sometimes there are technical reasons to be discussed at the end of the section.

Views onto domains, and operations on domains,

1. for the student are
   (a) survey statically the hierarchy for definitions, axioms, syntax and theorems (implemented by Isabelle)

   (b) show dynamically a definition, an axiom or a theorem, marked on the work-sheet, in its respective theory-file (tools for searching theories are provided by Isabelle [Pau97a])
   (c) select by pointing, and drag and drop, a theory into the work-sheet
   (d) select by pointing, and drag and drop, an axiom or a theorem from a theory-file into the work-sheet

2. for the author are
   (a) edit a theory-file, evaluate it and thus change the SML-top-level environment (implemented by Isabelle conveniently administering the update of theory dependencies)
   (b) edit an attached ML-file, evaluate it and thus change the SML-top-level environment (implemented by Isabelle conveniently administering the update of theory dependencies)

Views onto problem-types, and operations on problem-types are more often of dynamical nature, depending on the current proof-state. They are

1. for the student
   (a) survey statically the hierarchy of problem-types without or without details, i.e. problem-identification Def.2.2.8 only, or the fields *given, where, find, with, relate*, too
   (b) search dynamically the hierarchy while instantiating (Def.2.2.5) a selected problem-type with the formalization of the current example, or a subproblem of the example marked on the work-sheet; display with or without details, i.e. problem-identification 2.2.8 only, or display missing or superfluous objects, predicates not met by the example etc., too
   (c) select by pointing, and drag and drop, a problem-type from the presentation of the hierarchy into the work-sheet
   (d) search the hierarchy for identifiers Def.2.2.8 of problem-types
   (e) search the hierarchy for predicates used in the *where* and *with*-fields

2. for the author
   (a) insert problem-types into the hierarchy, delete from the hierarchy
   (b) edit problem-types in the hierarchy (this leads to a change of the SML run-time system)
   (c) (for test-facilities see p.133)

Views onto methods, and operations on methods

1. for the student are
   (a) survey statically the list of methods without or with details, i.e. the identification only, or the script or the guard, or both, too
   (b) select by pointing, and drag and drop, a method from the presentation of the list into the work-sheet
   (c) search dynamically the list of methods while instantiating the guard with the current example (or the formalization of a marked subproblem of the example) and display missing or superfluous objects, predicates not met by the example etc.
   (d) display the relation 'is subproblem of' p.53 for the current example
   (e) display a script while marking the currently evaluated tactic
2. for the author are
   (a) insert methods into the respective list, delete from the list
   (b) edit scripts and guards in the list
   (c) generate statistics on the relation 'is subproblem of': what are the prerequisites for a given example-collection ? or: which subproblems are employed most or least ?
   (d) (for test-facilities see p.133)

Views onto example collections, and operations on example collections

1. for the student are
   (a) survey the (collections of) collections of examples, with and without details, i.e. with the identification of the examples only, or with the descriptions, too
   (b) select by pointing, and drag and drop, an example from the collections representation into the work-sheet
   (c) show statistics on the problem-types, methods, definitions, theorems most often used in an example collection
   (d) display an examples hidden formalization and specification Def.2.3.3 (to be controlled by the DG ?!)
2. for the author are
   (a) insert examples into collections, delete from collections

(b) edit examples: their textual description Def.2.2.2 (ev. including drawings !) and the hidden formalization and specification (syntax-check, and availability of the elements of the specification Def.2.2.9)

(c) show the result of an example, i.e. calculate automatically without stepwise user-interaction

(d) show the results of an collection, i.e. indicate examples not solvable by the methods supplied

(e) generate lists of (sub-)problems and methods necessary as prerequisites for mastering an example collection.

Restrictions for authoring are twofold. One kind of technical restriction is given by the fact, that values of the SML-top-level environment have to be altered. As the tutor is designed as a multi-user system, authoring cannot be done in the production system.

The other kind of restriction is more of organizational concern. The 3D-universe of mathematics as presented in Fig.2.1 is a highly complex structure. Alterations in one part can introduce conflicts with other, very distant parts in the universe. If authors add some parts to the math knowledge, and make their example collections run, this does not mean, that these additions can immediately be integrated into a 'summarizing version'; rather, in general, conflicts need to be resolved (if not fore-seen and prohibited), and this is the task for real experts – a task of essential importance for the development of the tutoring system.

Chapter 4

CONCLUSIONS AND FUTURE WORK

The previous main chapters, developing the concepts and techniques, are now concluded by the inquiry, if the development has achieved the goals stated by the requirements in 1.3 at the end of the introduction.

The arrangement of the main chapters has more or less corresponded to a systematic walk through the development of concepts and techniques. Sometimes implementation issues were discussed or not discussed, not on the grounds of their intrinsic interest or lack of it, but because they were or were not a convenient peg from which to demonstrate key mechanisms.

Now in the review, the points are checked w.r.t. their relevance in fulfilling the requirements stated. Some of the checkpoints concern theory development, some points concern the prototype implementation. Some checkpoints are proven to be done in the thesis or in the prototype completely, some are prepared for immediate implementation, some others concern future research and development.

In order not to confuse the actual achievements of this thesis with future development, the key contributions of the thesis are mentioned first.

Then a systematic check over the points of the requirements follows.

Finally the checkpoints left open are collected and estimated for the effort necessary for their implementation.

4.1 The key contributions of the thesis

Given a rough idea of the user requirements, it took several years of preparatory work to come up with a requirements specification, which (1) describes features interesting and useful for math education, and (2) encounters a set of open subproblems solvable by one person within limited time. From the beginning it was apparent, that there were considerable large and swampy areas of unclear questions on the way to the goal, as well as clear, but unsolvable hurdles (for one person, e.g. re-engineering CAS for stepwise execution and reflection). The way found in this thesis is a narrow and shaky foot-bridge from the requirements to a solution, based on three very distant supporting columns: *problem-types* within the realm of formal methods, *scripts* within compiler construction, and *dialog-atoms* within human computer interaction, are the key contributions of the thesis.
The appropriateness and load-carrying capacity of the three columns is inquired first, followed by a scan over the whole arch of the foot-bridge w.r.t. the requirements.

4.1.1 Mechanical search on problem-types

The requirement of a logical framework for calculations lead to the concept of problem-types. The original character of the implementation of this concept has bee under-pinned in 2.2.4.
This concept, originally required for checking the correctness of the solution of problems, turned out to bring along remarkable additional value: problem-types can be structured in hierarchies by the 'refinement'-relation (Def.2.2.6); and then this hierarchy can be mechanically searched for the problem-type most appropriate for a given example.
Specifying problems is an important part of applied mathematics; the selection of appropriate methods, including the re-specification of a problem in order to apply an even more appropriate method, are also important in high-school math. The mechanical support given in searching the knowledge base (i.e. the hierarchy of problem-types) makes this task more enjoyable and efficient.

The introduction of the concept of problem-types considerably influences the kind, calculations are done, too. At the first sight it seemed, that a mere 're-engineering of CAS' would be necessary in order to obtain stepwise execution according to the requirements. But good use of subproblems leads to another structure of calculations.
For instance, a CAS solves the equational system $3x^2-3=0 \wedge -3y^2+12=0$ by Buchbergers algorithm [Buc65], presumably. A well-designed hierarchy of problem-types, however, could prevent shooting with such a powerful gun at this equation, and find out that it can be solved by some substitution; and this, in addition, is the right method for high-schools.

4.1.2 Scripts resuming from user-input

The concept of scripts and its interpreter establishes the prerequisites for meeting the most interesting requirements as defined: (1) the tutor knows how to do mathematics, i.e. he proposes the next step towards a solution, and (2) the tutor continues to do so after the student eventually has applied a rule (eventually another one as the tutor would have chosen).
A script is the instance in the tutor, which 'knows' which step has to be done in the solve-phase in order to solve a whole class of problems; this is not the challenge. The challenge is introduced by the additional condition (2) above:
If the student inputs some rule, and the deduction mechanism accepts this rule as applicable to the current proof-state: where is the associated tactic in the script ? Only if such a tactic can be identified in the script, the interpreter can continue to evaluate the script and find a next tactic to be applied. This is called 'resuming from user-input'. If the interpreter cannot resume, because the rule chosen by the student does not fit into the script at all, the tutor is called 'helpless'. The algorithm given in 2.4.4 solves the problem how to resume from user-input of rules.

That means, script can do both: given a proof-state, the script outputs a rule promoting the proof-state, *and* handles the input of rules while continuing that promotion. Put into a more general picture, the *direction of the input-output relation is kept variable !*
The direction of the input-output relation upon interpreting a *set* of equations and in-equations can be kept variable, as demonstrated by constraint solvers: variables input in one session can be part of the output in another session on the same set of knowledge.
The difference to constraint solvers is great and evident: Whereas the former operate on (in)equations only, *the scripts interpreter has to cope with sets of equations (by solving word problems w.r.t equivalent formulas) as well as with control structures typical for program languages, when keeping the input-output relation variable.*

Resuming from input of a formula concerns the object-language of scripts; the technique to cope with this issue is less inventive, and described in 2.4.6. The overall capability of resuming guidance after user-input and of 'knowing' the next rule to be applied in the calculation, brought into cooperation with the deductive mechanisms developed, leads to a kind of mathematics-engine, which here is called an 'autonomous math engine'. This new kind of engine is consequently exploited by the following point.

4.1.3 Symmetric atoms for human-computer-interaction

The autonomous mathematics engine is being exploited by a straight forward development of high interactivity, where the math engine in the tutor is established as a *partner* of the student.
Partnership here means that both, the tutor and the student, can contribute in solving an example with equal chances to propose the next step towards the solution, where the step can again concern an element of the object-language, a formula, or an element of the meta-language, a rule: The student can propose the next step — and the tutor checks the step and eventually asks the student for a justification of the step, or the tutor proposes the next step — and the student acknowledges this step and eventually requests an explanation.

The possibility of this view of the user and the system as equal partners, lead to the design of *symmetric dialog-atoms*. The novelty of this design is discussed in 3.1.4.
The symmetric dialog-atoms are shown to have appealing properties: They can almost arbitrarily be combined, sequences of them can skip certain interactions in order to increase or decrease the width of steps in solving an example, special combinations of atoms can be (re)used as a kind of 'higher-level' dialog-patterns, and the list of such atoms used along a dialog form a very instructive history.
This makes dialog-atoms a promising base for the development of 'higher-order' structures in guiding the dialogue between mathematics-engine and the student. The dialog-guide implemented in the prototype already produces data which might be interesting for didactic field studies.

4.2 Check for the initial requirements

This section closes the bracket opened by the statement of requirements in the introduction 1.3.: it checks whether the requirements are met by the concepts presented in chapters 2 and 3, it mentions open questions, and remarks the state of development of the respective part of the prototype.

This section does *not* repeat the references to related work; those have been given within the respective sections, or have been collected in separate subsections (2.2.4, 2.4.7 and 3.1.4).

4.2.1 The realization of the logical framework

As mathematics is reasoning, a proper foundation by formal logic is essential. Isabelle cannot be beaten in conciseness and clarity in this respect: Its implementation of the logical kernel is minimal, built upon a dozen of meta-rules, essentially contained in the file `Pure/thm.ML` of 2398 lines of ML-code. The implementation is based on a well settled theoretical foundation

dating back to work of Dana Scott first implemented by [Mil72], and this implementation already formally under-pinned [Mil73].

The concepts presented in this thesis concern example construction problems, which, in principle, can be regarded as a special case of theorem proving: 'solving (example construction problems) is proving existence by constructing an example'. [1] A logical foundation for this case could be given by [Bee84a], an implementation could be under-pinned according to [Bee88]. The practical implementation following the definition of the proof-tree, Def.2.3.1, however, does not yet have an elaborated foundation on logic. The thesis' concepts *extend* Isabelles meta-logic, where a *generalization* would be desirable.

This is the most serious open question, which should not be overridden by particular achievments gained by the concepts presented in the thesis and the features implemented in the prototype ! The acchievments are as follows:

Deductive rules are based on Isabelle, in particular rewriting is done exclusively by theorems introduced by Isabelle theories. Most of these theorems, however, are still given as axioms in the prototype implementation of the tutor, and are waiting for mechanized proof. Some theorems are built into the branch-types of the proof-tree, for an example see p.70.

The generality in the formulation of knowledge used by the mathematics-engine seems quite appealing: (1) domains (i.e. Isabelle theories) contain axioms and definitions of function constants using Isabelles powerful and simple technique to introduce 'syntactic sugar', (2) problem-types, and (3) methods guiding proofs and calculations build the base of the math knowledge. The knowledge is arbitrarily extensible, and reusable in many ways: theorems for methods, methods for problem-types etc.

The external representation of calculations is done by copying the relevant parts of the respective proof-trees to a worksheet on the screen. Promising examples are given in 2.3.2 and in the appendix. Whether the underlying structure really allows for simple representation of complicated structures in calculation (e.g. example on p.70), and help the student with easy comprehension, will be shown by experience in future class-room usage.

The structuring of large calculations can be done by folding-in selected parts of the calculation along the underlying structure of the proof-tree. Subsidiary calculations reside (instead at an extra sheet of paper) deeper in the nesting of that structure.

[1] due to verbal communication with Bruno Buchberger.

The design of the set of tactics for operating on the proof-tree, i.e. on its representation on the worksheet, can be found in 2.3.3. These tactics are rather different in comparison to the input-language of CAS as well as of CTP; experience in class-room usage will show the adequacy w.r.t. handling, and the implementation of larger problem classes in the tutor will exhibit, what kind of rules are missing.

Postconditions in the specification of an example have two rôles, both of which raise open questions: (1) a postcondition is characteristic for a problem-type, but there is neither a sufficiently general language for automatic instantiation of a 'postcondition-template' in the problem-type to a postcondition in the problem, nor an idea how to exploit postconditions in a search for appropriate problem-types. (2) a postcondition should allow to check a result; this is only possible for some cases (see equations p.58); other cases (see the maximum-example on p.49) are challenging tasks for theorem proving, not yet tackled. Here the high-level relatives of example construction problems, the implicit computation problems' come in through the back-door.

Many details of the logic are hidden from the student, as long as not asked for. For instance: indispensable to obtain correct solutions of equations (see the example on p.73), but usually hidden, is the decision whether to check the assumptions of a theorem at the spot, or to add them to the assumptions of the calculation, as implemented in the prototype. There are many further details, e.g. the selection of the term-order in rewriting, which is done automatically by the invocation of a method.
The design of the presentation of such details is still open: how to present wich information due to which request of the user.

4.2.2 The realization of autonomous problem solving

CAS are able to automatically solve problems because of their narrow view of problem; CTP have a broader view and encounter clear limits in automation. This thesis confined itself to example construction problems, to be automatically solved by knowledge separated from the meta-system. Several questions are open, and shall be answered by future experience:
(1) Can the hierarchy of problem-types modeled in a way which allows automated refinement over larger problem-trees (e.g. over all equations) ? How efficient is the refinement ? And how can the variety of problems in high-school math structured in order to really assist the student in problem solving ?
(2) How general can methods be described by the scripts ? How many problem-types are to be solved by more than one method, and how many

methods solve more than one problem-type ?
(3) Where are the draw-backs of separating the math knowledge from the math engine ? What kind of shortcuts will be enforced in special cases (inverse rewriting, partial terms, etc.) ?

On the strength of the concepts developed in the thesis the following remarks can be given:

All phases of problem-solving are supported in such a way, that the tutor can do the respective steps towards the solution autonomously, and the tutor can assist the student in trying her or his own steps.

The model-phase is supported least. If the tutor 'knows' the example to be solved (by a hidden formalization prepared by an author), it can tell the student, whether an item is missing or unknown.
One could imagine a component for modeling, which operates on a model covering the notions contained in the textual or graphical description of an example (like 'circle', 'rectangle', 'inscribed', etc.), as already done for geometry by [ABY85].
In the prototype, the feed back on syntax and type errors in the input is still insufficient.

The specify-phase is covered completely: if the specification of the example is known to the tutor, autonomous solving the example and assisting the student starts immediately, otherwise it starts after specifying the domain (for correct parsing of the input formulas) and the problem-type. Assistance is given by marking formulas already input with 'syntax', 'false' (w.r.t. a predicate of the precondition), 'superfluous' (i.e. not necessary for solving the current example) and 'incomplete' (if there is a list of formulas). The hierarchy of problem-types is not yet implemented in the prototype; thus the refinement of problems cannot yet demonstrated.

The solve-phase is covered by the scripts. The review has to add the remark, that the property of resuming after user-input still needs experimentation, in order to improve the heuristics for resuming from input formulas.

4.2.3 The realization of reflection

Reflection depends on the possibility to explicitly describe the essential parts of the mathematics knowledge in a representation readable for students. A closer look at Isabelles theories reveals many technicalities confusing for a non-expert. How will larger problem-trees look like, and special scripts ?

Also the prospectively large amount of knowledge and the highly structured interconnections between various parts are an issue. Which details and relations should be shown when ? This gives a bulk of detailed open questions to be tackled in practical implementation and experimentation.

The design decision, to exclude didactic considerations, turned out to be extremely helpful; it lead to a system not hampered by any pedagogical attitude.

The concepts of the thesis, and the state of the prototypes development contribute to the realization of reflection as follows:

Domains, definitions, axioms and theorems and their human readable presentation are already done by Isabelle theories, perfectly in comparison to other parts of the prototype. Direct access of the related theory by name of one of the elements is prepared by Isabelles user-interface (and exploited by Proof-General and by Isar), but this is not implemented in the prototype.

Problem-types and their hierarchy are prepared conceptually, but not implemented in the prototype. Direct access of the current problem-type in the hierarchy, immediate instantiation of problem-types with the actual example during a search in the hierarchy, and indication of possible refinements, would be a major project for the future.

Scripts are readable representations of algorithms. Two future amendments can be imagined: (1) Dynamical marking of the tactic currently executed. (2) Translation of expressions in the script into natural language, e.g. translate the line of code in a script
`main_equation = (hd o (filter (is_contained var))) equations`
into
in order to get the `main_equation`*, filter* `equations` *for* `var is_contained` *and take the first of them.*

Showing subproblems according to the black-box-white-box principle is done by consulting a list of such problems in the dialog-state, presently in the prototype. This decision should become part of the user-model in the future.

Use rewrite rules according to the black-box-white-box principle is done by the same list in the dialog-state. The list could be amended by a hierarchy associated with a user-model, and adapted dynamically.

Explain rewriting by animation. Rewriting is the basic mechanism used for deduction by the tutor, thus it should be explained as clearly as possible. [Aus00] demonstrated the feasibility of such an animation.

The worksheet represents calculations very close to handwritten ones. The selection of already existing formulas for continuing calculation in that point (for correction or for a variant), is prepared in the concept of the proof-tree, but not implemented in the prototype.

The flexible dialog facilitated by dialog-atoms allows to view related knowledge at the spot, in principle. Actual support of this possibility by an appropriate design of the front-end still needs lots of consideration.

4.2.4 The realization of reactive user-guidance

User-guidance is a concern of dynamically adapting a system to the user in a way which makes work pleasant and efficient. A tutoring system, additionally, underlies the seduction to compare with a human tutor who is able to provide for appropriate variation in the flow of interaction.

Reactive user-guidance is not only concerned with guiding the steps of problem solving, but also with adapting to the students individual preferences and to the actual situation in accessing the mathematics knowledge base.

The concepts presented so far realize reactive user-guidance by combination of the autonomous math-engine (scripts !) and flexible dialogs (dialog-atoms !). Only basic features are implemented in the prototype.

Free and seamless change of dialog modes from an 'active' user which inputs rules and formulas, to a 'passive' user listening the the tutors proposal for the next step, is implemented.

Interrupt a calculation for questions is prepared, but not yet implemented; also answering questions is not implemented.

The dialog-state in the present prototype comprises: a list of rules to be done hidden from the user, an integer denoting the 'activity' of the user, and an integer denoting the 'step-width', i.e. the inclination of the user to skip interactions. Further enhancements depend on the following two points.

High-level notions for the dialog-state need to be developed. These should comprise high-level dialog-patterns consisting of several dialog-atoms, eventually including conditional variants, and notions facilitating the selection among dialog-patterns.

High-level notions for the user-model need to be developed in correlation to the dialog-state. Both notions should include information about exercises already done, interactive attitudes while solving these exercises (initiative, step-width, errors, etc.).

4.3 Estimation of effort for future development

During the year 1999/2000 a prototype has been implemented by a team of three persons at IST, TU-Graz. This version of the prototype is a *minimal* demonstration version: the thesis developed several concepts which are rather new for the users aimed at, and also for the administrators responsible for the introduction of software in education. Thus this version should exhibit the concepts of stepwise calculation based on logic, and flexible dialogs.

The prototyping showed the feasibility of some key-concepts, gained valuable experiences on memory and computation-time requirements, and allows for reliable estimations of effort for future development. The next steps of development can be based on the concepts presented in this thesis without alteration. Three goals are proposed for the next steps, which can be tackled in one go, or scaled into three subsequent phases (a forth point is added in order to have a place for all requirements mentioned in this thesis):

1. **mathematics knowledge:** the minimal demonstration version contains almost no math knowledge. Adding simplifiers for I, Q, C, adding basic problem-types like equations, and adding methods solving those types, is the most urgent task.

2. **interactivity in tutoring:** this version should allow practical use in test-classes, i.e. it should provide for multi-user capability and at least one practically useful math knowledge base.

3. **authoring system:** this version should allow authoring by selected authors external to the developers team; the introduction into authoring the system should be done by a formalized course.

4. ideas postponed beyond the first three phases.

These three goals need work on the components as estimated in the following. Most of the work is development following the concepts presented in this thesis; this is marked by a 'D:' for development, i.e. it concerns mere implementation; research is marked by 'R:', that is all work due to concepts not described in this thesis. The unit for estimation is 'man-month' (MM); a diploma thesis is 4 MM, a project/seminar is 2 MM. About 70% of the tasks can be done by students withing project/seminars and diploma theses. Work

is subsumed under the component involved most: the estimation includes related work to be done in other components. Work has to be done on the following components or functions:

1. mathematics-engine
2. dialog-guide
3. work-sheet: this will integrate with the subsequent views; the front-end uses the SWING-library and the WebEq-library for 2-dimensional formulae
4. views onto math knowledge: notions on this work can be found in 3.2.3
5. general math tools: the related work requires special mathematics knowledge
6. middle-ware and system: networking and the interface between Java and SML

Tab. 4.1: Estimation of man-months for ME

Math-engine R&D-topics	man-months for goal			
	1	2	3	4
D: clean up code (modules, arguments, etc.)	0.5			
user-input of a formula in the model-phase and in the solve-phase of a calculation	2.0			
D: continue calculation at any previous position on the work-sheet (cut ptree and script)	1.0			
D: script-interpretation for subproblems, and for rule-sets on request of *Details*	1.0			
D: complete the set of rules and the branch-types of the proof-tree		2.0		
D: speed up rewriting: fast discrimination nets, terms instead cterms, numerals		2.0		
D: handle variants in formalization and specification w.r.t. solve-phase		2.0		
D: perform type-inference on user-input formulas (different domains !)		0.5		
D: optimize the script-interpretation (mixed list-tactic expressions, let)		1.0		
R&D: optimize handling of language-levels (access to SML-system instead of association-lists)			2.0	
D: clean up operator-precedences for scripts etc. in Isabelle			0.5	
D: instantiate rules in the select-list on request				
D: rewriting on sub-terms of formulae (a prerequisite for the animation of rewriting)				2.0
R: concept of pre- and post-condition for rule-sets				2.0
R: concept for inconsistent proof (during trial and error)				2.0
	4.5	7.5	2.5	6.0

Tab. 4.2: Estimation of man-months for DG

Dialog-guide R&D-topics	man-months for goal			
	1	2	3	4
D: select dialog-atoms by activity and step-width	0.5			
D: complete the set of datoms	1.0	2.0		
D: handle not yet accepted proof-states (e.g. after input of a formula)		3.0		
R: develop dialog structured by high-level dialog-patterns				x.0
R: develop user-model related to the high-level dialog				x.0
	1.5	5.0	0.0	x.0

Tab. 4.3: Estimation of man-months for work-sheet

Work-sheet R&D-topics	man-months for goal			
	1	2	3	4
R: input formulae on the work-sheet, also complete formulae partially given by the system	20			
D: input formulae by menus		2.0		
D: copy, cut and paste formulae			1.0	
D: edit work-sheets for printout			2.0	
D: handle sub-terms of formulae and animate rewriting				4.0
D: store calculations and replay them				1.0
	2.0	2.0	3.0	5.0

Tab. 4.4: Estimation of man-months for views

views on the knowledge R&D-topics	man-months for goal			
	1	2	3	4
domains: p.131				
D: make views dynamic				2.0
D: drag and drop into the work-sheet				1.0
problem-types: p.132				
D: survey hierarchy 1a	1.0			
D: instantiate hierarchy 1b		1.0		
D: select by pointing, drag and drop 1c		1.0		
D: edit and insert in the hierarchy 2a,2b			2.0	
D: search the hierarchy for identifiers, predicates 1d				1.0
methods: p.133				
D: survey statically 1a		1.0		
D: select by pointing, drag and drop 1b		0.5		
D: edit script and guard, and insert into list 2a,view-m-edit			2.0	
D: survey while instantiating 1c				1.0
D: show relation 'is subproblem' 1d				2.0
D: mark the currently evaluated tactic in the script 1e				1.0
R&D: generate statistics on the relation 'is subproblem of' 2c				4.0
example collections				
D: survey with/out details 1a		1.0		
D: select by pointing, drag and drop 1b		0.5		
D: edit examples (text and formalization), insert into collection 2a,2b			2.0	
D: check the solvability of the examples 2c,2d		2.0	2.0	
R&D: show statistics on the problem-types, methods, definitions, theorems etc. 1c				x.0
R&D: compile the prerequisites for a collection 2e				x.0
	1.0	8.0	8.0	12.0

Tab. 4.5: Estimation of man-months for math tools

General math tools R&D-topics	man-months for goal 1	2	3	4
R&D: factorization of integer terms	2.0			
D: demo knowledge (English)	1.0	1.0		
R&D: generic pre- and post-conditions for problem-types Def.2.2.5		4.0		
R&D: canonical simplifier for rationals		2.0		
R&D: canonical simplifier for radicals		2.0		
R&D: floating point numerals				4.0
	3.0	9.0	0.0	4.0

Tab. 4.6: Estimation of man-months for middle-ware

Middle-ware and System R&D-topics	man-months for goal 1	2	3	4
D: speed-up the bridge Fig.3.2		1.0		
R&d: integration into S.E.A.L. and Hyperwave		2.0		
D: installation for the system			1.0	
	0.0	3.0	1.0	0.0

Tab. 4.7: Survey on man-months for future development

Survey on man-months for the components / functions	for goal 1	2	3	sums
math-engine	4.5	7.5	2.5	14.5
dialog-guide	1.5	5.0	0.0	6.5
work-sheet	2.0	2.0	3.0	7.0
views on the knowledge	1.0	9.0	6.0	16.0
general math methods	3.0	9.0	0.0	12.0
middle-ware and system	0.0	3.0	1.0	4.0
sums	12.0	35.5	12.5	60.0

Chapter 5

CASE STUDIES

The work on the thesis comprised several years of exploring possibilities of tutoring mathematics, of studying concepts and techniques required, and several stages of narrowing down the field of investigation. From this work two early feasibility studies on Isabelle are collected in this chapter, together with a survey on mathematics topics suitable for tutoring, and further example-calculations.

The first feasibility-study documents the first approach to Isabelle, raising the question: can Isabelle calculate ? Isabelles capabilities as generic logical framework are evident, but is it suitable for calculations in high-school mathematics ?

The second study investigates Isabelles tools for handling example construction problems, establishes requirements on interactivity in the modeling and specification phase, and by the way makes clear the difference to `solve` *in CAS.*

These two studies on Isabelle lead to the decision to prefer Isabelle to Redlog and to Mathematica in the first phase of prototyping. The second study has been documented as technical report [Neu99a], the first one is part of [Neu99b].

A survey on high-school mathematics collects typical examples most suitable for the basic mechanism of the tutor, for rewriting, and gives an estimation for the extent the tutor covers 'all' high-school mathematics.

Finally a collection of examples is given, which drove the development of the thesis and the prototype. Some of them mark the limit of the tutors present scope.

5.1 Can Isabelle calculate ?

Isabelle is well-known and esteemed as generic logical framework, and the development of theories covers most of the mathematics which is the theoretical basis of what is taught at high-schools (and the development is still ongoing with much effort !).
However, Isabelle is not meant for calculations as occurring in solving equations etc. This study raised the question in the headline at a time, when Isabelle had no numerals except the natural numbers $0, 1, 2$ *and an attached theory for integer numerals. The development of numerals went on steadily, and Isabelles latest version has generic numerals already.*
This study is still of interest, because it describes the interfaces to Isabelles internals used in the prototype, and because the design-decisions made for the prototype are different from those made by Isabelle.

Currently there are many projects narrowing the gap between computer theorem provers and computer algebra systems. Theorem provers are intended to do what their name tells, thus originally they were concerned with logic formulae, and afterwards strengthened their calculational features. Modern CTPs are rather different in how they implement calculations, and several arithmetic tools are still under construction. HOL [GM93] has the integers incorporated into its logic, and calculates by use of a tactic implementing Presburger Arithmetic. PVS [ORR+96] as well has the logical representation for integers as part of the standard logic, and numeric simplification embedded into the standard tactics. Isabelle, however being generic and extensible, started with natural numbers [Pau94], added integers in the last version, and recently a very complete arithmetic toolkit [BNP98] for integers has been implemented. Moreover, decision procedures fit well into theorem provers' features [AG93]. In particular, there are implementations of decision procedures for real algebra in HOL [Har97], for integer expression based on [Sho79] in PVS, and recently in Isabelle [BNP98].

5.1.1 Simplification of $2 - a + 1 - 2a$ by trial and error

Isabelle is a large software system, which provides simple access for the naive user as well as highly sophisticated tools for experts. Let us approach as naive users by trial and error and start with Isabelle as it comes in its standard distribution with high order logic, HOL, (but without its emacs front-end in order to obtain plain ascii output).

```
unixprompt:isabelle
val it = false :  bool
> thy;
```

```
val it =
 {ProtoPure, CPure, HOL, Ord, Set, Fun, subset, equalities, mono, Prod, Lfp,
  Relation, Trancl, WF, NatDef, Nat, Arith, Divides, List, Option, Map}
 :  theory
```

where > is Isabelles prompt and the evaluation of the global `thy` shows Isabelles `theories` available, which contain the hierarchy of definitions, axioms and theorems. [1] Now let us try to simplify $2 - a + 1 - 2a$ using Isabelles inference mechanisms as simply as possible.

```
> goal thy ''2-a+1-2*a=?z'';
 1.  2 - a + 1 - 2 * a = ?z
val it = [] :  thm list
> by(simp_tac simpset() 1);
2 - a + 1 - 2 * a = Suc (2 - a) - (a + a)
No subgoals!
val it = () :  unit
```

We formulate the calculation as a proof-`goal` and use the built in simplifier, i.e. we apply the tactic `simp_tac` to subgoal 1, which rewrites by theorems from a global rule set `simpset` and by use of several techniques [Pau97a]. The simplifier works well for complicated predicates in proofs, but here the output $Suc(1 - a) - (a + a)$ is not what we want — Isabelle works on natural numbers in HOL, which ist most appropriate for theorems usually proved.

Isabelle contains an implementation of integers, too. We load the respective theory by setting the path

```
> loadpath:=[''.'',''.../Isabelle98/src/HOL/Integ''];
> use_thy''Bin'';
...
val it = () :  unit
> goal thy ''#2-a+#1-#2*a=?z'';
 1.  #2 - a + #1 - #2 * a = ?z
val it = [] :  thm list
> by(simp_tac simset() 1);
#2 - a + #1 - #2 * a = #2 + $~ a + #1 + $~ (#2 * a)
No subgoals!
val it = () :  unit
```

The # distinguishes the integer numerals from natural numbers, the $~ is the unary minus sign — this format of numerals is not nice [2] , but there are more urgent problems. Theory `Bin` calculates numerals, but only if we have joined them in the term before. For this purpose we construct an AC-operator by gathering the laws of associativity, commutativity

[1] Isabelles' theories are viewed best in their HTLM-format as presented in the distribution package.

[2] There is much ongoing work on numerals in the Isabelle development groups. The discussions whether to overload the numerals or to use (ugly) embedding functions are not finished at the date of writing.

and leftcommutativity already present in the system. Leftcommutativity $f(x, f(y, z)) = f(y, f(x, z))$ is necessary in order to obtain confluence, i.e. to complete the rule set.

```
> val ac = [zadd_assoc,zadd_commute,zadd_left_commute,
            zmult_assoc,zmult_commute,zmult_left_commute];
val ac =
  [''?m + ?n + ?k = ?m + (?n + ?k)'',''?m + ?n = ?n + ?m'',
   ''?x + (?y + ?z) = ?y + (?x + ?z)'',''?m * ?n * ?k = ?m * (?n * ?k)'',
   ''?m * ?n = ?n * ?m'',''?x * (?y * ?z) = ?y * (?x * ?z)''] :  thm list
> val AC = HOL_basic_ss addsimps ac;
val AC =
  Simpset
   {finish_tac=fn,loop_tac=fn,
     mss=Mss bounds=#,congs=#,mk_rews=#,prems=#,procs=#,rules=#,termless=#,
     subgoal_tac=fn,unsafe_finish_tac=fn} :  simpset
```

The infix function `addsimps` generates the record Isabelle uses as simplification set, including a termorder `termless` we will need below. Up to this point `by(simp_tac simpset() 1)` caused the proof to stop. In the sequel we shall apply several simplification sets in sequence, and we do not want to stop the proof after one application. Thus we need the following tactical using the theorem of transitivity `trans`.

```
> fun simp_cont s i = (rtac trans i) THEN (simp_tac s i);
val simp_cont = fn :  simpset -> int -> tactic
```

Now we can arrange the subterms in such an order that theory `Bin` eventually allows to simplify `#2 + #1 = #3`.

```
> goal thy ''#2-a+#1-#2*a=?z'';
 1.  #2 - a + #1 - #2 * a = ?z
val it = [] :  thm list
> by(simp_cont (simpset()) 1);
 1.  #2 + $~ a + #1 + $~ (#2 * a) = ?z
val it = () :  unit
> by(simp_cont AC 1);
 1.  $~ a + (#1 + (#2 + $~ (a * #2))) = ?z
val it = () :  unit
> by(resolve_tac [refl] 1);
#2 - a + #1 - #2 * a = $~ a + (#1 + (#2 + $~ (a * #2)))
No subgoals!
val it = () :  unit
```

We are drawing closer to our goal, but still `(#1 + (#2 + t))` cannot be calculated. It would help to have `(t + (#1 + #2))`, the numerals shifted backwards. Isabelles default term order is order by number of subterms, and we have to change it for our purposes. Thus we leave the view of a completely naive user and dig into Isabelles internals. Isabelle wraps its terms very comfortably. We can decompose for instance theorem `refl` by the following functions.

```
> refl;
val it = ''?t = ?t'' :  thm
> (#prop o crep_thm) refl;
val it = ''?t = ?t'' :  cterm
> term_of ((#prop o crep_thm) refl);
val it = Const (''Trueprop'',''bool => prop'') $
 (Const # $ Var # $ Var (#,#)) :  term
```

ML compresses the output of large datatypes by #, as of the term above. A term (a simply typed λ-term [NPS90]) is the following ML datatype. [3]

```
datatype term =
 Const of string * typ |
 Free of string * typ |
 Var of indexname * typ |
 Bound of int |
 Abs of string * typ * term |
 $ of term * term
```

We define a new order on this datatype, which makes numeral constants greatest: function `termless_const : term * term -> bool`. We suppose the source contained in the file `termorder.ML` and add the order to the simplifier `AC`.

```
> use''termorder.ML'';
...
> val AC = HOL_basic_ss addsimps ac settermless termless_const;
val AC =
Simpset
 {finish_tac=fn,loop_tac=fn,
  mss=Mss {bounds=#,congs=#,mk_rews=#,prems=#,
  procs=#,rules=#,termless=#},
  subgoal_tac=fn,unsafe_finish_tac=fn} :  simpset
```

Now let see what we have gained:

```
> goal thy ''#2-a+#1-#2*a=?z'';
 1.  #2 - a + #1 - #2 * a = ?z
val it = [] :  thm list
> by(simp_cont (simpset()) 1);
 1.  #2 + $~ a + #1 + $~ (#2 * a) = ?z
val it = () :  unit
> by(simp_cont AC 1);
 1.  $~ a + ($~ (a * #2) + (#2 + #1)) = ?z
val it = () :  unit
> by(simp_cont (simpset()) 1);
 1.  $~ a + ($~ (a * #2) + #3) = ?z
val it = () :  unit
> by(resolve_tac [refl] 1);
#2 - a + #1 - #2 * a = $~ a + ($~ (a * #2) + #3)
```

[3] We omit the definition of types, which is comparably simple.

```
No subgoals!
val it = () :  unit
```

Now, `#2 + #1 = #3` has worked, but the resulting term is not nice: terms are ordered ascending by size as long as they contain variables, and after the largest summand we find the atomic numerals. Moreover `#2 * a` would be definitely nicer than `a * #2`. We can shorten the simplification by a tactic `rewrite`:

```
> goal thy ''#2-a+#1-#2*a=?z'';
 1.  #2 - a + #1 - #2 * a = ?z
val it = [] :  thm list
> by(rewrite [simpset(),AC,simpset()] 1);
#2 - a + #1 - #2 * a = $~ a + ($~ (a * #2) + #3)
No subgoals!
val it = () :  unit
```

But let us turn to the more urgent problem, that `$~ a + ($~ (a * #2)` still does not simplify to `$ ~3 * a`. We look for another confluent simplifier. Algebra systems, for instance [Wol96], follow the strategy to factor out the unary minus. In our example we would like to have

```
$~ a         →  ($~ #1) * a
$~ (a * #2)  →  ($~ #2) * a
```

as abstract syntax, [4] which allows to factor out the constant coefficients

```
$~ 1 * a + ($~ 2) * a = ($~ 1 * $~ 2) * a = $~ 3 * a
```

It is not desirable to factor all terms $a \cdot c + b \cdot c = (a + b) \cdot c$ in general. Thus we need to apply the (reverse) distributive law to terms with a, b numerals and c a subterm *with variables only*. This can be implemented by simplification procedures in Isabelle [Pau97a] p.121. But this is already highbrow usage of Isabelle, and we stop our trial and error trip at this point.

The reason to stop before approaching the details of Isabelles simplifier is, that we plan to develop our own simplifier in order to meet requirements of interaction.

Another specialty of numerals in theorems provers can be show here briefly. By use of some functions we can look at the internal representation of integer numerals

```
> term_of (parse thy ''#9'');
val it =
 Const (''Bin.integ_of_bin'',''Bin.bin => Integ.int'') $
 (Const # $ (# $ #) $ Const (#,#)) :  term
> atomize (term_of (parse thy ''#9''));
***  Const ( Bin.integ_of_bin, Bin.bin => int)
***    Const ( Bin.Bcons, [Bin.bin, bool] => Bin.bin)
```

[4] This internal format, of course, should be presented to the user by prettyprinting as `-a` and `-2a`.

```
***      Const ( Bin.Bcons, [Bin.bin, bool] => Bin.bin)
***        Const ( Bin.Bcons, [Bin.bin, bool] => Bin.bin)
***          Const ( Bin.Bcons, [Bin.bin, bool] => Bin.bin)
***          Const ( Bin.PlusSign, Bin.bin)
***        Const ( True, bool)
***      Const ( False, bool)
***    Const ( False, bool)
***  Const ( True, bool)
val it = () :  unit
```

given by a binary representation `True, False` which does not suffer from limitations in range, but have some disadvantages for our algebraic purposes. In Isabelle the numerals can be implemented much more adequate for algebra, if one does not regret the loss of logic exactness. We will show how to implement numerals in the sequel.

5.1.2 Implementation of numerals in Isabelle

If we implement numerals in Isabelle we want to build upon mathematical theories already established in the system. The Isabelle theory of integers is `Integ.thy`. We go without the binary representation of integer numerals as for instance [BNP98], take Isabelles 'raw' numbers of type `xnum` and provide for an embedding into the integers:

```
Num = Integ +
consts
  ''int_num'' ::  xnum => int (''_'')
end
```

We store the above text in file `Num.thy`, load it by `use_thy''Num''` and are ready to parse numerals with this information in the current theory `thy` and to prettyprint them due to the syntax definition `(''_'')` described in [Pau97a] p.73. [5]

```
> use_thy''Num'';
val it = () :  unit
> val ct = parse thy ''#123'';
val ct = ''#123'' :  cterm
> term_of ct;
val it = Const (''Num.int_num'',''xnum => Integ.int'')
          $ Free (''#123'',''xnum'') :  term
val it = () :  unit
```

By this way we inherit all theorems, definitions and operators from `Integ.thy`. We regard the implementation of the unary minus first.

```
> val minus_num = parse thy ''$~ #123'';
val minus_num = ''$~ #123'' :  cterm
```

[5] We take the #-sign to identify integers in order to avoid parse-AST translations at the first approach.

```
> term_of minus_num;
val it =
  Const (''Integ.zminus'',''Integ.int => Integ.int'')
    $ (Const (#,#) $ Free (#,#))
    :  term
> atomize( term_of minus_num);
*** Const ( Integ.zminus, int => int)
***   Const ( Num.int_num, xnum => int)
***   Free ( #123, xnum)
val it = () :  unit
>
> val minus_var = parse thy ''$~ a'';
val minus_var = ''$~ a'' :  cterm
> term_of minus_var;
val it =
  Const (''Integ.zminus'',''Integ.int => Integ.int'')
    $ Free (''a'',''Integ.int'')
    :  term
>
> val neg_num = parse thy ''#~123'';
val neg_num = ''#~123'' :  cterm
> term_of neg_num;
val it = Const (''Num.int_num'',''xnum => Integ.int'')
    $ Free (''#~123'',''xnum'')
    :  term
```

The functions for evaluating the numerals are simple. The representation of `neg_num` is what we shall use for a normal form. The transformation to normal forms will be considered in 5.1.3. Here we emphasize that, all we need for those transformations, is already in the knowledgebase of `Integ.thy`, for instance

```
> zminus_zadd_distrib;
val it = ''$~ (?z + ?w) = $~ ?z + $~ ?w'' :  thm
> zmult_zminus;
val it = ''$~ ?z * ?w = $~ (?z * ?w)'' :  thm
> zminus_zminus;
val it = ''$~ $~ ?z = ?z'' :  thm
```

These theorems are proven in `Integ.thy` wrt. the definition of integers as equivalence classes of naturals with equal difference (which is the usual way of intorducing integers by the way of natural numbers). What we even more welcome than the theorems being proven is, that all the machinery dealing with operator precedence, omitting superfluous parenthesis, infix notation, the notion of bound variables (`{x.  x + #1 = #2} = ...`) etc. etc. is all available.

Now let us turn to calculating the numerals. Let us assume (see 5.1.3) a normal form for terms assuring that all numerals are adjacent in any term

after rewriting a particular simplifier. Then we have to detect subterms

$$n_1 + n_2, \qquad n_1 + (n_2 + t)$$

$$n_1 \cdot n_2, \qquad n_1 \cdot (n_2 \cdot t)$$

in arbitrary terms, where n_i are numerals and t is another arbitrary subterm. A function `get_cpair` scanning a term for such pairs of numerals is not hard to do. If we have extracted such adjacent pairs of numerals, what then ?

Our view of calculation is based on proving, i.e. the process where a given formula f is transformed to another formula f' by application of an appropriate theorem t. Thus we are obliged to generate theorems, even for $1+2=3$ for instance. Such theorems are called [Har97] proforma theorems.

As we go without a binary representation of integer numerals (and we even need real numerals), we leave rewriting and thus the scope of a theorem prover, and calculate sums and products outside the provers logic. Isabelle provides a special mechanism, *oracles*, for calling external reasoners like model checkers or algebra systems. The input from such an oracle is made a theorem without any further check by Isabelle, the responsibility is with the programmer of the oracle.

In our application we can take the responsibility. We define the oracle by extending the theory already defined.

```
Num = Integ +
consts
  ''int_num'' ::  xnum => int (''_'')
oracle
  calc = mk_calc_oracle
end

ML
exception CalcExn of term;
fun mk_calc_oracle (sign, CalcExn t) =
  calc_prop t handle _ => raise CalcExn t;
```

where the function `calc_prop` generates a proposition which states that the pair of numerals found (as subterm connected by + or ·) is equal to the sum or product. And now we 'really can calculate' numerals in terms containing numerals and variables intermixed:

```
> goal thy ''#1 + (#2 + ( $~ a + #~2 * a)) = ?z'';
1.  #1 + (#2 + ($~ a + #~2 * a)) = ?z
val it = [] :  thm list
> val num_pair = get_cpair( concl_of( topthm()));
val num_pair = Some ''#1 + (#2 + ?a)'' :  cterm option
> val calc_thm = calc_oracle( term_of( the num_pair));
val calc_thm = ''#1 + (#2 + ?a) = #3 + ?a'' :  thm
> br calc_thm 1;
#1 + (#2 + ($~ a + #~2 * a)) = #3 + ($~ a + #~2 * a)
```

```
No subgoals!
val it = () :  unit
```

After application by tactic `br` (by resolution) the proforma theorem `calc_thm = ''#1 + (#2 + ?a) = #3 + ?a''` can be dropped. How to simplify the remaining part `$~ a + #~2 * a = #~3 * a` is concern of rewriting techniques which will be discussed below.

5.1.3 Normal forms and simplifiers

Normal forms as defined in 2.1.2 are important in order to get some standardized representations of terms as an indispensible prerequisite for the application of any further methods.

The minus-operator is the first point do deal with in integer terms. The ambiguity in the representation $a - b = a + (-b)$ needs to be resolved [6] . The right-hand side is the usually employed for several reasons. Another ambiguity concerns the minus-sign itself, the binary operation minus $-$ and the unary minus, which we denote by $\sim$. We employ the following ruleset:

simplifier Minus

(1)	$a - b$	$=$	$a + (\sim b)$	
(2)	$\sim(a + b)$	$=$	$(\sim a) + (\sim b)$	*zminus_zadd_distrib*
(3)	$\sim(a \cdot b)$	$=$	$(\sim a) \cdot b$	*zmult_zminus*
(4)	$is\_atom\ a \Rightarrow \sim a$	$=$	$(\sim 1) \cdot a$	
(5)	$is\_num\ a \wedge is\_num\ b \Rightarrow$			
	$\Rightarrow a \cdot c + b \cdot c$	$=$	$(a + b) \cdot c$	*zadd_zmult_distrib*

The identifiers in the rightmost column refer to Isabelle theorems already present. The important point in this list is that it contains conditional equalities. Conditional rewriting (definition 2.1.2) is considered to show up with 'nasty surprises' [NB98] p.270.

But we cannot renounce the conditions: In (4) the rule $\sim a = (\sim 1) \cdot a$, helpful for preparing the collection of variables, and for calculating the values of negative numbers, Obviously, the condition *is_atom* ensuring that the rule is not being applied to structured terms, is necessary for termination. Eventually the condition could be dropped, if we could find an appropriate order with $\sim > \cdot$. In (5) the condition ensures that the (reverse) distributive law is applied to numerals only, thus joining them to an 'adjacent pair' which can be evaluated as shown in the previous subsection. Omitting the condition would transform the term more than necessary

Weak normal forms already provide for some standardization of terms. The simplifier *Minus* produces a weak normal form substituting the binary

[6] Isabelle99 has a unary minus-sign denoted by $-$ and this works in spite of the ambiguity mentioned.

minus by unary minus and by moving the unary minus to the leaves of the term. A stronger normal form can be produced by ordering the term. Terms can be ordered by recursive path orderings, for our case the lexicographic path order (definition 2.1.4) is sufficient. We extend this definition, i.e. *LPO1 ... LPO2c*, by *LPO0* in order to cope with numerals as follows:

$$\begin{aligned} s >_{lpo} t \iff & \textit{(LPO0)}\ s \in \mathit{Num} \wedge t \in \mathit{Num} \wedge s >_n t, \text{ or} \\ & \quad\ s \notin \mathit{Num} \wedge t \in \mathit{Num}, \text{ or} \\ & \textit{(LPO1)} \ldots \end{aligned}$$

where *Num* is the set of numerals and $>_n$ the order on *Num*. The implementation of an according function $termless : Term \times Term \to Bool$ is implemented easily. We use this order for ordered rewriting (definition **??**) together with the following AC-operators.

simplifier AC-add

(1)	$b + a$	$=$	$a + b$	*zadd_commute*
(2)	$(a + b) + c$	$=$	$a + (b + c)$	*zadd_assoc*
(3)	$a + (b + c)$	$=$	$b + (a + c)$	*zadd_left_commute*

simplifier AC-mult

(1)	$b \cdot a$	$=$	$a \cdot b$	*zmult_commute*
(2)	$(a \cdot b) \cdot c$	$=$	$a \cdot (b \cdot c)$	*zmult_assoc*
(3)	$a \cdot (b \cdot c)$	$=$	$b \cdot (a \cdot c)$	*zmult_left_commute*

simplifier AC = AC-add $\cup$ *AC-mult*

These simplifiers are all complete rule sets. *Minus* and *AC-mult* together terminate in a normal form which should be produced as soon as possible after the appearance of a term in a calculational proof. *Minus*, *AC-mult* together with *AC-add* usually result in a major rearrangement of a term, which may call for a students special attention in order not to confuse him.

5.1.4 Conclusions and future work

The conclusion of the study is: Isabelle can be taught to calculate. The study showed how to implement numerical calculation on integer terms, how to adapt the termorder to the needs of calculation, and how to normalize terms for calculation.

The approach presented does not exactly follow the spirit of Isabelles logical rigor. We shall briefly explain 'how' (we do not follow) by comparison with the recent development of an arithmetic toolkit for integers [BNP98] and explain 'why' we do not follow Isabelle traditions.

The arithmetic toolkit mentioned implements integers in a binary representation, which can calculate operations as $+ - * \ ^\wedge$ *div mod* by rewriting within Isabelles logic. Simplification of terms with numerals, variables and other function symbols is done outside the logic by calling an oracle, as we have done. The oracle works by mapping the Isabelle terms t_I to appropriate SML terms t_{ML}. Then t_{ML} is rewritten to t'_{ML} by SML functions

implementing the rewrite rules. Finally t'_{ML} is mapped back to an Isabelle term t'_I, and the oracle returns the theorem $t_I = t'_I$.

Again, this kind of simplification does not meet our requirements for interaction with the user, who may want to know which theorem has been used for a proof step (and should it be $1 + 2 = 3$). This is the reason why we choose the design described in this study.

Calculation in the integers is not all in high-school mathematics, rational, real and complex numbers are needed as well.
The question about rational numbers is clear: Isabelle provides for rationals defined as quotient field over the integers, but this theoretic construction is not necessary for practical calculation, i.e. for computing of numeric values and for rewriting. Rather, division is introduced as partial function.

A major problem are the reals. Already syntax for integers suffers from the # heading the numbers, and it seems to be almost impossible to get rid of that. The problem with the reals is definitely worse, we still have no concrete idea how to solve it. It seems to be hard to make Isabelles syntax-analyser accept the usual format of reals.
Surprisingly, the availability of reals is not so much urgent as should be in priciple (w.r.t. real world problems as well as calculus which relies on complete domains): a scan over textbooks (see 5.3) shows that the majority of examples do not involve numbers with digits behind the comma. The author suspices, the reason is that major parts of example colletions date back to times before the advent of pocket calculators, and thus are designed for mental arithmetic.

Complex numbers do not introduces new problems, they follow the lines of integer and rational numbers.

5.2 Equations – a hierarchy of interdependent sub-problems

This is another early feasibility-study on two questions: (1) Is Isabelle suitable for implementing problem-types ? (2) Which interactive features are desirable in the modeling- and specification-phase ? These questions are being investigated from a technical point of view with respect to the logical framework Isabelle. For the investigation a 'problem specification module' (PSM) implemented in ML based on Isabelle is presented.
The prototype of the PSM provides a hierarchical classification of problem types, checks if an example meets the specification described by a problem type, and whether the pre- and postconditions of a method unify with the specification. On demand the PSM can refine a vague specified problem, and can propose methods for solving the specified example.
The examples are drawn from Diophantine equations.

5.2.1 Iterations of the specification and solving process

Equations frequently require the specification of their type intertwined with the solving process. This is because the two relations between problem types P_1 and P_2, namely 'P_1 refines P_2' and 'P_1 uses P_2 (as a subproblem of the solving method)', largely coincide with equations. Let us look at some equations:

$$\frac{x+1}{x-1} + x = \frac{x-1}{x+1} \tag{5.1}$$

$$\sqrt{4x+9} = \sqrt{x} + \sqrt{x+5} \tag{5.2}$$

$$\sin^2 x - 6\cos^2 x + \sin x \cos x = 0 \tag{5.3}$$

What type of equation shall we have, a linear, a quadratic, some high order polynomial, or what else ? We cannot decide on that question unless we actually have done some transformations, i.e. unless we have entered the problem solving process. After some steps following three different methods for solving the equations we get

$$x^3 + 3x = 0 \tag{5.4}$$

$$x - 4 = 0 \tag{5.5}$$

$$\tan^2 x + tanx - 6 = 0 \tag{5.6}$$

Now we see that 5.1-5.4 is a polynomial of degree 3, 5.2-5.5 is linear and 5.3-5.6, substituted $u = \tan x$ gives a quadratic equation. In high-school assignments one can find equations with even more nested subproblems.

Having students of elementary mathematics courses in mind, we really want to have the most appropriate — and the most simple — solving method

for a given example, and not some general decision procedure which aims at solving the largest class of problems possible. For instance an equation of type 5.1-5.4 from above can only specify the subproblem 'univariate integer equation' which, after multiplying out the fractions, can be refined to 'linear equation' for the above example (and eventually other problem types for other equations).

We see: In equation solving the specification of equation-types is particularly challenging because the specification process recursively comes up to new subproblems after some steps of the solving process. What we want to have is

1. a collection of problem-types which cover an partition all the different equations that can occur

2. a feature that checks whether a given equation belongs to a particular problem-type

3. for each problem type a list of methods which con solve all equations belonging to that problem type

4. and last not least a feature that finds the most appropriate problem type for an equation, which in particular refines the specification of subproblems.

5.2.2 An implementation based on Isabelle

We describe an implementation of a 'problem specification module (PSM)' which faces the challenges just mentioned. The PSM is based on the theorem prover Isabelle. The only number constants, which are contained in the present Isabelle distribution package, are integers [7] . Thus we decided to do our investigation on equations in the domain of integers, called Diophantine equations [8] in the literature.

We use Isabelle's theory HOL/Integ/Integ.thy, built upon high order logic (HOL) and enhance it by numerals as discussed in the feasibility study 5.1 on terms with numeric constants.

Declarations and signatures

Here we show declarations, which require special discussion within the specification phase.

[7] This statement refers to Isabelle94-8; from this version onward numerals have been developed consequently. Isabelle99 provides for generic (integer) numerals which can be instantiated to appropriate types.

[8] Diophant of Alexandria

The data in the mathematics knowledge base are described by the following ML declaration:

```
type domID = string;
type pblID = string list;
type metID = (domID * string);
type ndID = string;
type pblNd = ndID * (metID list) * ppc;
datatype pblTre = PTyp of (pblNd * (pblTre list));
datatype dom = Dom of ( domID * ( pblTre list ));
type mat3D = dom list;
```

Each node `pblNd` in the problem tree `pblTre` [9] not only contains an identifier `ndID:string` and the 'given', 'where', 'find', 'with' in the `ppc`, but also a list of methods `(metID list)`. A list facilitates several methods to be applied, some of which may produce a 'better' result (considered 'better' from some meta-level; example: a better approximation) than the others.

Some further design decisions contained in the ML code above will be discussed below under remarks 'on the design of the hierarchy'.

The mathematics-engine which performs the mathematical tasks in the specification process, is one of the main three modules, front-end, dialog-manager and 'math-engine', as we will call it shortly. The interfaces between the modules are kept simple. In particular the interface of the math-engine with the outside (i.e. the other two modules) is almost perfectly described by

```
signature MATHS_ENGINE =
  sig
    val init_expl   :  expl * spec -> result * next_steps
    val do_         :  int -> result * next_steps
    val check       :  step -> chk list
    val init_method :  metID -> result * next_steps
  end
```

The functions above basically work on two states during the specification process.

The function `init_expl` enters a state (1) **'production'** where `do_` takes the `i-th` element of `next_steps` just previously proposed by the system and 'produces' the next step of the proof (of the calculation) as side-effect on the internal proof state. And the value of `do_` immediately returns the `result` together with a proposal for `next_steps`. [10]

[9] An early version of Def.2.2.3.

[10] Thus there is little to do for the dialog-manager if there is no reason from the dialog-state or a request from the user.

State (2) **'discussion'** will be entered, if the user decides to input another step than proposed in `next_steps`, which can be done by `check` only. `Check` returns a checklist (`chk list`) of items which cause troubles, but it can return `[Ok]`, too. In this case a `do_` returns to state (1).

The two states involved in the specification process are left, when the specification of a domain, a problem type and a solving method which meets the given example (`init_method`).

The data handled at the interface between math-engine and dialog-manager are types step, next_steps, result and chk list.

```
datatype step =
    Domain of domID
  | Expl of string list
  | Probl of pblID
  | Method of metID
  | InitMethod of metID
  | ...
  | EndExample
  | CantDo; (*should not occur:'check' before 'do'*)
```

`Step` provides the information for each action promoting the solution of an example; i.e the `step`s are those which extend the proof-tree. [11] At the end of each step (propagated by `do_`) the maths-engine proposes the `next_steps` to continue the proof:

```
type next_steps = int * (bool * step) list;
```

The `int`eger points at the first choice in the list, each `step` in the `list` is decorated with a flag of type `bool` indicating, whether this step is considered to succeed or not.

Besides the `next_steps` we are interested in a result, of course:

```
datatype result =
    Spec of ((bool * string) list * (bool * string) ppc)
  | Form of string list;
```

The result expected during the solving process seems obvious: the formula resulting from the current proof step, which is `Form of string list`, the second part from above. The result from a specification step is not so obvious. We decided to present rather more information about the state than less: `(bool * string) list` displays the parts of the given example (type string), decorated with `bool` flags which are false, if an item could

[11] In the specification phase the *sequent* is not extended, because *one* problem object (definition **??**) contains all information concerning the specification

not have been parsed correctly. And `(bool * string) ppc` presents the whole problem type [12] under consideration. Again the `bool` flags indicate that something is wrong with an item. In the case of the 'given', 'where', 'find' and 'with' in a problem type, however, the errors that may occur are of various kinds. The possibilities of errors with problemtypes are captured by the

```
datatype chk =
    Ok
  | Syntax of (bool * string)
  | SyntaxExpl of (bool * string) list
  | SyntaxProbl of (bool * string) ppc
  | SyntaxMeth of (bool * string) ppc
  | MissProbl of (bool * string) list
  | MissMeth of (bool * string) list
  | FalseProbl of (bool * string) list
  | FalseMeth of (bool * string) list
  | ...
  | Refine of pblID;
```

The `list`s and `ppc` always contain the whole source list (and in the same order, which both considerably simplifies the interface with the front-end — but this is not our topic here).

The interface to Isabelle

The PSM's code could be added without changing Isabelles source. The additional code only needs to be evaluated after loading Isabelle with the theory `Integ.thy` (which in turn loads its parent theories up to HOL).

Some (data)types and functions of Isabelle are used by the PSM, for the PSM the interface is small. Most of the material is well documented in the reference manual contained in Isabelles distribution package. Some details we had to read out of Isabelles sources. [13]

Functions for parsing, matching and substitution are based on facilities of Isabelle.

Parsing is necessary, because the interface between the maths-engine and dialog-manager only handles strings. Our function *parse* is described in definition 2.1.1, it delivers the abstract Isabelle type `cterm`, which is 'certified' under the signature of a theory. The other direction, from `cterm`s to strings is completely covered by Isabelles pretty printing facility.

[12] `(bool * string) ppc` presents the pre- and postcondition of the method under consideration

[13] In that case 'isabelle-users@cl.cam.ac.uk' provided help very accurately

Furthermose in the PSM we need to compare objects of given examples with objects constituing problem types; this requires pattern matching on terms. the PSM's according functions *match* and *matching* are described in definitions 2.1.3 and 2.1.3.

We also need to check, if examples meet the characteristics of problem types described by predicates; this requires substitution and evaluation. Definition 2.1.1 describes the according function *subst* in the PSM.

Finally, the functions `meets` and `refines` [14] are based on the functions *match*, *subst* and *eval*.

Compose and decompose terms: Parsing, matching and substitution deals with 'certified terms', which Isabelle checks for well-formedness with respect to arities and types, and which Isabelle represents in pretty printing.

In some cases we have to go one level deeper into the system, for instance when composing the formula for 'find' from the objects 'given' in an example:

`example:`	$x^3 + x + 2 = 0,\ x$
`given:`	`?lhs=?rhs, ?bdVar`
`find:`	{ `bdVar. ?lhs=?rhs` }
`composed to:`	$\{x.\ x^3 + x + 2 = 0\}$

Similarly we have to decompose terms like $\{x.\ x^3 + x + 2 = 0\}$ into the parts $x^3 + x + 2 = 0$ and x for checking with respect to the 'given' of the subsequent problem. For that purpose we have to handle Isabelles terms directly, which reveal their constructors for reasons such as that.

On the design of problem types

We decided to have the objects 'given' in the problem types as elementary data-types as possible. We could have chosen to describe equations in 'given' as '$\{x.equ(x) \wedge P(x)\}$' from the beginning. But we aim at combining several problem types to more powerful methods. For instance, Newton's method can be seen as a combination of the subproblems (1) generalize the equation to a function (2) differentiate that function (3) linearize the function by use of the derivative and (4) iterate finding the zero-point of the linearized function. Simple data-structures of the objects will make the subproblems interact easier.

When the objects in the 'given' list become more complicated, simple pattern matching is no more sufficient. For instance, if we want to describe a polynomial or an equational system, we need 'enumerative' descriptions of terms, which Isabelle has not yet implemented: [15]

[14] Early versions of Def.2.2.5 and 2.2.6.

[15] Instead of this infinite axiom schema of first order we actually would use an axiom (formula) of second order ([Har97] p.73)

```
tuple_n (i = 0..n) a_i
sum_n (i = 0..n) a_i * x ^ i
and_n (i = 0..m) ((sum_n (j = 0..n) a_i_j * x_j) = b_i)
```

Finally we want to call attention to an ambiguity in our notation for equations: the identifier x as bound variable, *and* as the name of an element of the solution set. But we find that this comes closest to the traditional way of equational calculus.

On the design of the problems' hierarchy
mat3D contains the following knowledge for our examples of Diophantine equations:

```
"Integer"
  \--"equation"
  |     \--"univar"
  |     |     \--"linear"
  |     |     \--"lhs=rhs"
  |     |     \--"lhs=0"
  |     |           \--"homogen"
  |     |           \--"inhomogen"
  |     |                 \--"not-canceld"
  |     |                 \--"canceld"
  |     \--"multivar"
  |     |     \--"lhs=rhs"
  |     |     \--"lhs=0"
  |     |           \--"not-canceld"
  |     |           \--"canceld"
  |     |                 \--"linear"
  |     |                 \--"linear-in-1-var"
  |     |                 \--"Pells"
  |     |                 \--"Pythagoras"
  |     \--"system"
  |           \--"lhs=rhs"
  |           \--"lhs=0"
  |                 \--"linear"
  |                 |     \--"homogen"
  |                 |     \--"inhomogen"
  |                 |           \--"n*m"
  |                 |           \--"n*n"
  |                 \--"polynom"
  \--"equ-auxiliary"
  |     \--"cancel-equ"
  \--"divisor"
        \--"is-divisor"
        \--"divisors"
"Rational"
  \--"equation"
        \--"rational"
              \--"linear"
```


[16] This kind of presentation is used by Isabelle in HTML for surveys on the theories'

We immediately see in this hierarchy that a considerable portion is dedicated to serval kinds of 'normalizations'. At a closer look we notice that those normalizations are located in different ways.

On the location of normalizations. Normalized forms [17] simplify pattern recognition. A normal form for equations would be: rhs = 0, lhs expanded by the law of distributivity, and cancelled to gcd = 1, but the coefficient a_n of the element of highest degree not with $a_n = 1$ in order to remain within the integer domain. Then Pells equation for instance can easily be recognized within other multivariate equations.
We see several possibilities to do normalizations:

1. Place them explicitly into the hierarchy of problem types, and then:
   (a) at the bottom of the hierarchy, as done for the integer univariate equations in our example, or
   (b) On top, as done with the multivariate equations. This may not be in accordance with calculations done by hand; beginners could be irritated by this technically motivated procedure.
2. Encapsulate them into the search procedure and into every problem solving method, too. This would have the advantage, that for instance Pells equation could be written as $x^2 - Dy^2 = 1$ as usual in mathematics literature instead of $x^2 - Dy^2 - 1 = 0$.

5.2.3 User-guidance in the specification process

We show by examples, how the PSM meets the issues formulated for reactive user-guidance.

We show the action of the PSM at an internal interface: the interface between the 'mathematics-engine' and a 'dialog-manager'. The maths-engine contains the PSM; the 'dialog-manager' prepares the presentation of the problem solving process to the user. In particular the 'dialog-manager' is supposed to decide for various kinds of interaction (see the capter on tutoring) or to hide several steps to the user, for instance the selection of the `domain` in some exercise concentrating on other tasks. It even makes sense for some exercises to hide the specification process as a whole.

This internal interface is somewhat technical, but close to the PSM under consideration.

---

hierarchy. It would be nice, if we could give this presentation for our hierarchies of problem types in the same way.

[17] as opposed to *normal forms*

Issue (1): The PSM can specify a problem

Both, a tutor and a software tutor, should be able to solve a problem him or herself. The PSM provides all features to enable the dialog-manager to generate a dialogue, in which the system demonstrates how to do a specification step by step.

PSM knows the next step in prepared examples: In an educational setting there are usually prepared collections of exercises. If the exercises have prepared a special kind of 'formalization of the examples', and a specification with pointers at a knowledge domain, a problem type and a method, then the PSM itself knows each step. We pass over the information by `init_expl` at the beginning of the dialogue:

```
> init_expl ( [ ''#3 * x - #4 = #5'', ''x'' ],
              ( domain=''Integer'',
                probl=[''Integer'', ''equation'', ''univar'', ''linear''],
                method=(''Integer'', 'solve-linear'') ) ];                    (1)
Domain ''Integer''
> do;
ProblTyp [''Integer'', ''equation'', ''univar'', ''linear'']
> do;
Method (''Integer'', ''solve-linear'')
> do;
InitMethod (''Integer'', ''solve-linear'')
goal thy ''{x.  #3 * x - #4 = #5} = ?z'';
Level 0
{x.  #3 * x - #4 = #5} = ?z
 1.  {x.  #3 * x - #4 = #5} = ?z
val it = [] :  thm list
Level 1
{x.  #3 * x - #4 = #5} = {#3}
No subgoals!
val it = () :  unit
>
```

The > are the prompts of the test-interface, followed by the command `'do'` of the dialog-manager. The system ackowledges each `'do'` with a suggestion for the next step, propagating the specification by `'Domain'`, `'ProblTyp'`, `'Method'` and `'InitMethod'` which calls Isabelle's subgoal module for solving the problem.

PSM guides in specifying a new example: A random example, in our case a randomly input equation, cannot be surely solved in any case, of course. Nevertheless, the system can provide very concrete guidance in specifying the problem:

```
> init_expl ( [ ''#3 * x - #4 = #5)'', ''x'' ], noSpec )
Domain ''Pure''
> chk_domain ''Integer'';
[ Ok ]
> do;
ProblTyp [''None'']
```

```
> chk_problem [''Integer'', ''equation''];                                (2)
[ ProblTyp [''Integer'', ''equation'', ''univar'', ''linear''] ]
> chk_problem [''Integer'', ''equation'', ''univar'', ''linear''];
[ Ok ]
> do;
Method (''Integer'', ''solveLinear'')
> do;
InitMethod (''Integer'', ''solveLinear'')
...
```

PSM comes up with the defaults `Domain ''Pure''` and `ProblTyp [''None'']`, and needs to get some more information (which could come from some defaults in the dialog manager or from the user).

The commands `'chk_*'` and `'do'` alternate in the dialogue: `'chk_*'` requests a list of diagnostics from the math-engine until the list is `'[Ok]'`. Then the system is ready to `'do'` a step towards the solution, and it responds with a suggestion for a next step.

PSM refines a vague specification in the example above: In step `(2)` the command `chk_problem [''Integer'', ''equation'']` suffices to get the exact problem type at once.

Issue (2): The PSM can justify its specification

The second requirement we demand from a tutor is to be able to discuss each step in a solving process with the user.

If we watch the two examples above, we see: Each step is declared *before* it is executed. This kind of dialogue at the internal interface to the mathematics-engine enables the dialog-manager to discuss the step with the user, and eventually allows the step to be presented by the system or request the step as input from the user.

Particularly distinct from the behavior of present algebra systems is the following feature:

PSM decomposes a problem into subproblems

One fundamental idea in problem solving is to break down the problem into simpler parts until all of them can be solved one by one. There are several ways to structure the handling of subproblems: this can be done in sequence, in parallel, recursively etc. Several strategies guide the decision for these combinations: systematic enumeration, reduction of the search space, elimination of possibilities, approximation, bisection etc.

We show this feature in a simple sequence of 'normalization' steps:

```
> init_expl ( [ ''(#3 * x ^ #2 -- #9 * y ^ #2 = #3)'', ''( x, y)'' ],
                ( domain= ''Integer'', probl=[''Integer'', ''equation''],
                  method=(''None'',''None'' ) ) ];
Domain ''Integer''
> do;
ProblTyp [''Integer'', ''equation'', ''multivar'', ''lhs=rhs'']
```

```
> do;
InitMethod (''Integer'', ''lhs-rhs=0'')
goal thy ''{x.  #3 * x ^ #2 -- #9 * y ^ #2 = #3} = ?z'';
Level 0
{x.  #3 * x ^ #2 -- #9 * y ^ #2 = #3} = ?z
 1.  {x.  #3 * x ^ #2 -- #9 * y ^ #2 = #3} = ?z
val it = [] :  thm list
Level 1
{x.  #3 * x ^ #2 -- #9 * y ^ #2 = #3} = {x.  #3 * x ^ #2 -- #9 * y ^ #2 -- #3 = #0 }
No subgoals!
val it = () :  unit
ProblTyp [''Integer'', ''equation'', ''multivar'', ''lhs=0'', ''not-canceld'']
> do;
InitMethod (''Integer'', ''cancel-equ'')
goal thy ''{x.  #3 * x ^ #2 -- #9 * y ^ #2 -- #3 = #0 } = ?z'';
Level 0
{x.  #3 * x ^ #2 -- #9 * y ^ #2 -- #3 = #0 } = ?z
 1.  {x.  #3 * x ^ #2 -- #9 * y ^ #2 -- #3 = #0 } = ?z
val it = [] :  thm list
Level 1
{x.  #3 * x ^ #2 -- #9 * y ^ #2 -- #3 = #0 } = {x.  x ^ #2 -- #3 * y ^ #2 -- #1 = #0 }
No subgoals!
val it = () :  unit
ProblTyp [''Integer'', ''equation'', ''multivar'', ''lhs=0'', ''not-canceld'', ''Pells'']
> do;
InitMethod (''Integer'', ''solve-Pells'')
goal thy ''{x.  x ^ #2 -- #3 * y ^ #2 -- #1 = #0 } = ?z'';
Level 0
{x.  x ^ #2 -- #3 * y ^ #2 -- #1 = #0 } = ?z
 1.  {x.  x ^ #2 -- #3 * y ^ #2 -- #1 = #0 } = ?z
val it = [] :  thm list
Level 1
{x.  x ^ #2 -- #3 * y ^ #2 -- #1 = #0 } = {x.  x = ...}
No subgoals!
val it = () :  unit
```

The system can 'refine' the problem to subproblems by use of the hierarchy in *mat3D*. The result is the same as with an algebra system's 'solve'-function. The difference is that the 'specification module' allows the user to decide which method shall be used at which point.

We have already raised the question of whether this particular sequence of subproblems should be built into each method separately.

Issue (3): The PSM can check the user's steps in specification

A third requirement we demand from a tutor is, to be free to pass over the initiative to the student, and to comment on his steps in the solving process.

The problem specification module provides the following functionality:

The order of the specification steps is free as shown in the following example, where the domain is given at last, although it is necessary for parsing the input and starting proper specification:

```
> init_expl ( [ ''x ^ #3 + x + #2 = #0 '' ], noSpec )
Domain ''Pure''
```

```
> chk_probl [''Integer'',''equation'',''univar'',''lhs=0'',''homogen''];
[ Ok ]
> do;
Domain ''Pure''
> chk_method (''Integer'', ''factor-bdVar'');
[ Ok ]
> do;
Domain ''Pure''
> chk_domain ''Integer'';                                                    (3)
[ Missing [ ''bdVar = ?bdVar'' ] ]                                           (4)
> chk_expl [ ''x ^ #3 + x + #2 = #0'', ''x''];                               (5)
[ False [ ( a_0 (x ^ #3 + x + #2 ) x ) = #0 ] ]                              (6)
> chk_probl [''Integer'',''equation'',''univar'',''lhs=0'',''inhomogen''];
[ Problem [''Integer'',''equation'',''univar'',''lhs=0'',''inhomogen'',''canceld'']]
> chk_probl [''Integer'',''equation'',''univar'',''lhs=0'',''inhomogen'',''canceld'']];
[ Ok ]
> do;
Method (''Integer'', ''try-divisors'')
> do;
InitMethod (''Integer'', ''try-divisors'');
...
```

As soon as the PSM can parse in step `(3)`, it does the checks following these steps:

PSM checks for completeness of the given objects in step `(4)` noting the missing object `bdVar`, which is added in step `(5)`.

PSM checks the constraints on the objects given in the example. In the listing above the predicate `( a_0 (x ^ #3 + x + #2 ) x ) = #0` is false in `(6)`. The appropriate problem type has to be chosen, which in turn leads to the suggestion of the right method.

5.2.4 Conclusions and future work

We have found the following answers to the initial questions:

Appropriateness of Isabelle: Isabelle provides the most important prerequisites for implementing problem-types as well as Mathematica (which had been used for a preceding feasibility-study). Its knowledge is freely extensible to arbitrary topics, its language can flexibly be adapted and it is close to traditional mathematics notation, and the strong typing is an advantage over untyped systems.

Interactive features: We have shown: It is possible to construct a problem specification part for a tutoring system which imitates a human tutor in that it can (1) specify examples itself, (2) discuss the specification and (3) check random user input.

The support for the user is given by the following features: The system 'knows' the next successful specification step (if the problem is appropriately prepared) and can give specific advice for tackling new problems in a context, where the knowledge is prepared.

And we have shown: The class of problems predominant in applying mathematics can be substantially supported in the specification phase. The class is 'example constructing problems'. Its specification comprises the selection of a knowledge domain, of a problem type and of a method appropriate for solving the given problem.

Future work in continuing this prototype will be:
(1) Design hierarchies of some parts of high-school mathematics. A first candidate are equations, consequently extending the hierarchy presented here. The feasibility-study exhibits the difficulties connected with this task: find the arrangement of types of equations parallel on branches of the problem-tree, find appropriate predicates to distinguish them, and design the whole hierarchy as intuitive for the user as well as efficient for mechanical search.
(2) Develop an authoring system which allows experts to implement and test their own problem-hierarchies efficiently.

5.3 Rewriting – a survey on high-school math

This case study conveys a complete scan over a typical textbook used for mathematics education in Austrian technical high-schools on the secondary level, grade 9 [S$^+$98a] to grade 12 [S$^+$98b]. The scan collects examples where the description essentially equals the formalization, i.e. those without relevant text or drawings in their descriptions. For each more or less homogeneous group one representative is selected. The examples' notation in the textbook is purposefully maintained in the collection: this will give raise to annotations within the collection, and comments in the conclusion at the end.

The following collection of examples is divided into such involving canonical simplifiers and such without canonical simplifiers. This division concerns the crucial distinction between examples for which a result can be judged correct with certainty: this holds for final results, and is equally important for the interactive behavior of the tutor, i.e. the decision whether a formula input by the user can be rejected as wrong with certainty needs a canonical simplifier for solving the word problem (by testing two terms for equivalence by calculating their respective normal-form).

All examples are listed together with their result, which, of course is not given in the textbook.

5.3.1 Topics involving canonical simplification

Integer terms have two canonical simplifiers and two normal-forms of general importance, the polynomial form and the factorized normal-form. Testing formulae for equivalence will use the polynomial form, because it can be calculated by rewriting (as opposed to the factorized form) and is less expensive in computational power.

Evaluation of ground terms are the first examples when starting the scan with the first volume of the text book. Section 5.1 has shown how the calculation of operations on numeral constants can be integrated into the rewriting paradigm.

$$\begin{aligned} 8\ 9(7\ 11-6)5+72/6 &= \cdots = 25572 \\ ((-20)/(-4)-4)(7-11(-3)) &= \cdots = 40 \\ (+10)((-6)-8) &= \cdots = -480 \end{aligned}$$

where the two lines stem from [S$^+$98a] p.36 and p.37 respectively. The notation has already been adapted to the needs of mechanical treatment by using parenthesis only instead of { [(_)] } , helpful for beginners

in matching opening and closing elements of the pair. We also dropped the operator for multiplication for convenience (which does not follow the design decision on p.42).
The last example is copied literally; note the inconsistency in using the sign for positive integers which is maintained throughout this part of the examples. What kind of concrete syntax will be used when in a mechanical treatment ?

Expand terms with variables as described in section 5.1 is done (like simplification in all the examples of the collection) intertwined with numeric evaluation of sub-terms. Examples are

$$\begin{aligned} 10x + ((7y - 12x) - (5y + 12x - 13)) - 13 &= \cdots \\ \cdots &= -14x + 2y \\ (3x + 2)(5x + 7) - (-(-3(5x - 3)(7x + 5) - (2x + 8)(3x + 7))) &= \cdots \\ \cdots &= -96x^2 + (-19)x + 3 \end{aligned}$$

from [S$^+$98a] p.59 and p.63 respectively.

Factorize terms can only be done by 'reverse rewriting' as discussed on p.40. Factorization, and thus irreducibility, depend on the underlying domain. In elementary mathematics education this is either $\mathcal{I}$, $\mathcal{Q}$, $\mathcal{R}$ or $\mathcal{C}$. All of them are unique factorization domains with 1.

The respective part of the collection in [S$^+$98a] p.65 begins with examples giving a partial result – a kind of interaction featured by an appropriate dialog-atom.

$$\begin{aligned} 30a^2bc - 27a^2b = 3a^2b(\cdots - \cdots) &= \cdots = \\ 3x(x + y) - 2x^2(x + y) &= \cdots = (3x - 2x^2)(x + y) \\ (5 - a)2m - 3n(a - 5) &= \cdots = (5 - a)(2m + 3n) \\ (5x + 5y + zx + zy &= \cdots = (5 + z)(x + y) \\ y^3 + 6y^2 + 12y + 8 &= \cdots = (y + 2)^3 \\ (u + v)^2 - (w - x)^2 &= \cdots = (u + v + w - x)(u + v - w + x) \end{aligned}$$

Terms with exponents $\in \mathcal{N}$ are normal forms in mathematics notation, in the real practical sense. However, $a^n = a \cdot a \cdots a$ n times, is not only a notational convention: in factoring this exceeds rewriting: $aaa + aaaaa$ can be factored by the rewrite rule $ab+ac = a(b+c)$, but a^3+a^5 can not without dealing with the numerical constants in the exponent.

Simplification of ground terms are the initial examples, again, as found in [S$^+$98a] p.36 and p.37 respectively:

$$\begin{aligned}(3\ 2^2+24)3^3-8\ 4^2+52/4 &= \cdots = 857\\ (10(-1)^6+(7(-2)^4+2\ 3^2-5(-2)^2\ 2 &= \cdots = 230\end{aligned}$$

Expansion of terms to polynomial form concerns the follow-up exercises in [S$^+$98a] p.58, p.63 and p.65 respectively:

$$\begin{aligned}6a^2b+6ab^2-7a^2b+9ab^2-10a^2b+5ab^2 &= \cdots = -11a^2b+20ab^2\\ (3a-2b)(5a^2+7ab+b^2)-(5a^2-7ab+b^2)(3a+2b)\cdots &= 22a^2b+(-4)b^3\\ (3x+2)^3-3x(3x+2)^2-(3x-2)(3x+2)2 &= \cdots = 24x+16\end{aligned}$$

Factorization again can only be handled by 'reverse rewriting'; the exercises are from [S$^+$98a] p.65 and p.66, where the initial examples provide for help by partial results:

$$\begin{aligned}30a^2bc-27a^2b=3a^2b(\cdots\ \cdots) &=\\ 3x(x+y)-2x^2(x+y) &= \cdots = (3x-2x^2)(x+y)\\ (5-a)2m-3n(a-5) &= \cdots = (5+(-a))(2m+3n)\\ (5x+5y+zx+zy &= \cdots = (5+z)(x+y)\\ y^3+6y^2+12y+8 &= \cdots = (y+2)^3\\ (u+v)^2-(w-x)^2 &= \cdots = (u+v+w-x)(u+v-w+x)\end{aligned}$$

The example in the last line is a typical challenge for calculation by hand. There is, as with some former examples, an notational inconsistence in using the unary minus-sign. An abstract syntax, presumerably, will have $(+a)+(-a)$; and if there are pedagogical reasons important enough, a adaptive 'pretty printing' function provide for the respective concrete syntax either as $a+(-a)$ or as $a-a$.

Rational terms require some considerations about their notion. Question: $\frac{x^2-1}{x-1} = \frac{(x+1)(x-1)}{x-1} = x+1$? The two terms are not 'functionally' [18] equivalent: the left-hand term has no value at $x=1$, but the right-hand term has. If we define rational terms as elements in the quotient-field formed from $\mathcal{R}[x_1,\cdots,x_n]$, then the simplification problem consists in finding a canonical simplifier S for the equivalence relation $-$ defined on $\mathcal{R}[x_1,\cdots,x_n]\times(\mathcal{R}[x_1,\cdots,x_n]-\{0\})$ by [BL82]:

$$(f_1,g_1)-(f_2,g_2) \quad \text{iff} \quad f_1\cdot g_2 = f_2\cdot g_1$$

where we restrict the domain of the polynomials to $\mathcal{R}$, a field and a unique factorization domain, which avoids some complications.

[18] if we disregard the theory of *meromorphic* functions

Expansion of terms in this domain $\mathcal{R}$ is not mere rewriting by the law of distributivity any more. The question, what a normal form could be, is not trivial.
For $\mathcal{R}$ there exist three computable functions $G, /, l$ with the following properties:

$G(f,g)$ =a greatest common divisor (GCD) of f and g
(i.e. G satisfies: $G(f,g)|f$, $G(f,g)|g$ and
for all h : if $h|f$ and $h|g$, then $h|G(f,g)$),
$f|g \Rightarrow f \cdot (g/f) = g$ (division),
$l(f)$ is a unit and $f \equiv g \Rightarrow l(f) \cdot f = l(g) \cdot g$
(i.e. the function $s(f) := l(f) \cdot f$ is a canonical simplifier
for the relation $\equiv$ defined by: $f \equiv g$ iff f and g are
associated elements, i.e. $f = u \cdot g$ for some unit u.
A unit is an element for which there exists an inverse).

In this case the above simplification problem can be solved by the following canonical simplifier S [BL82]:

$$S((f,g)) := (c \cdot (f/G(f,g)), c \cdot (g/G(f,g))), \quad \text{where} \quad c = l(g/G(f,g)).$$

The canonical form expands to summands which either are polynomials or rational terms with the nominators degree less than the denominators degree. The computation of this canonical form is rather time consuming and needs techniques beyond rewriting (some generalization of the Euclidean algorithm etc.). But rational terms play such a fundamental rôle in high-school mathematics, that it is worth any effort to implement the calculation normal-forms in $\mathcal{R}$. Examples are from [S$^+$98a] p.39, p.40, p.40, p.68 and p.67 respectively:

$$\begin{aligned}
\frac{870}{900} &= \cdots = \frac{29}{30} \\
\frac{\frac{181}{14} - \frac{184}{21}}{\frac{5}{6} + \frac{10}{9}} &= \cdots = \frac{15}{7} \\
\left(\frac{55}{2} - \frac{50}{17}\left(5 + \frac{29}{14}\right)\right) / \frac{17}{4} + \frac{44}{3} / \left(4 + \frac{36}{13}\right) &= \cdots = \frac{45439}{12138} \\
\frac{x+2}{x-1} + \frac{x-e}{x-2} - \frac{x+1}{(x-1)(x-2)} &= \cdots = \frac{2x^2 - 5x - 2}{(x-1)(x-2)} \\
\frac{3(x+2)}{6} - \frac{4(x-2)}{3} + \frac{2(x-1)}{36} - \frac{x+1}{4} &= \cdots = \frac{-37x + 121}{36}
\end{aligned}$$

Fractions raise another notational question (as can be seen in the examples above): when use them, and when use the slash / for division ?

Rational terms with exponents $\in \mathcal{N}$ *and* $\in \mathcal{I}$ create the next group of examples. As this group is considered very important in high-school mathematics, these examples are exercised at several locations in the book: the following lines in sequence [S$^+$98a] p.60, p.61, p.61, p.66, p.68, p.69, p.70, [S$^+$98c] p.36, p.62.

$$\begin{aligned}
\frac{(a-b)^3(x+y)^4}{(x+y)^2(a-b)^5} &= \cdots = \left(\frac{x+y}{a-b}\right) \\
((\frac{4x^4}{3y^2})^2(\frac{2x^3}{5y})^3)/(-(\frac{8x^4}{15y^2})^2) &= \cdots = -\frac{2x^9}{5y^3} \\
\frac{3x^4}{6x^{-3}} &= \cdots = \frac{x^7}{2} \\
\frac{2x-50x^3}{25x^2-10x+1} &= \cdots = \frac{2x+10x^2}{1-5x} \\
\frac{a^2}{a-b} - \frac{4ab^3}{(a^2-b^2)(a+b)} - \frac{b^2(a-b)}{(a+b)^2} &= \cdots = a+b \\
(4x^2+4x+1)\frac{x^2-2x^3}{4x^2+2x} &= \cdots = \frac{x(1-4x^2)}{2} \\
\left(\frac{27a}{9a^2-4} - \frac{6a+13}{6a+4} + 1\right)\left(\frac{3a-7}{3} - \frac{a-4}{2}\right) &= \cdots = \frac{3}{4} \\
\left(\frac{\frac{1}{a+b}+\frac{1}{a-b}}{\frac{a}{b}-\frac{b}{a}}\right)/\frac{a^2b}{a^2-b^2} &= \cdots = \frac{2}{a^2-b^2} \\
\left(\frac{3x^2}{5y}\right)^{-2}\left(\frac{7y^4}{x^{-2}}\right)\left(\left(\frac{5}{(xy)^5}\right)^2\frac{3^{-2}}{7^3}\right) &= \cdots = 1 \\
\left(\frac{3x^2}{5y}\right)^{-2}\left(\frac{7y^4}{x^{-2}}\right)^{-3}/\left(\left(\frac{5}{(xy)^5}\right)^2\frac{3^{-2}}{7^3}\right) &= \cdots = 1
\end{aligned}$$

Complex expressions do not introduce any new notions or difficulties, except that there is the special symbol i which is distinct from other variables. Examples, from [S$^+$98c] p.181:

$$\begin{aligned}
(2+3i)-(3+2i) &= \cdots = -1+i \\
(6+i)(1-i) &= \cdots = 7-5i \\
\frac{(8-7i)^2}{8+7i^49} &= \cdots = -\frac{664}{113} + -\frac{1001}{113}i
\end{aligned}$$

Radical terms require, similar to rational terms, considerations on the terms themselves: If we have the rule $(s \cdot t)^r = s^r \cdot t^r$, do we allow for $\sqrt{(-x)} \cdot \sqrt{(-x)} = -x$? The least confusing convention for high-school students might be, to restrict radices to arguments $a \geq 0$. That requires to formulate all rules as conditional rewrite rules. Then each rule application

adds an assumption to the environment. These assumptions, however, normally are not shown to the student, and treated in a 'second line reasoning' when checking the postcondition of the problem.

The domain of radical terms has a canonical form, but it is badly intelligible. The term is better conceivable than its normal form [BL82]:

$$\left(\frac{2x-2}{x^3-1}\right)^{-7/3} + \left(\frac{2/(x+1)^{1/2}}{24x+24}\right)^{1/4} =$$

$$= \frac{x^4+2x^3+3x^2+2x+1}{48x+48} \cdot 2^{\frac{11}{22}} \cdot e^{\frac{3}{4}} \cdot (x+1)^{\frac{3}{4}} \cdot (x^2+x+1)^{\frac{1}{3}} + \cdots$$

$$\cdots + \frac{1}{6x+6} \cdot 2^{\frac{9}{12}} \cdot 3^{\frac{3}{4}} \cdot (x+1)^{\frac{1}{4}}$$

where the latter is the normal form. The calculation of this canonical form requires considerable computational power. Nevertheless, this is necessary for checking user input terms.

Roots , i.e. the respective examples, are found in [S+98c] p.53 to p.56.

$$4\sqrt[3]{3} + 5\sqrt[3]{24} - 2\sqrt[3]{81} = \cdots = 8\sqrt[3]{3}$$

$$\sqrt{x^3-1}\sqrt[3]{\frac{x^3+1}{x^6-1}} = \cdots = \sqrt[6]{x^3-1}$$

$$\frac{\sqrt[3]{x^2}\sqrt[5]{x^3}\,x^2}{x\sqrt[15]{x}\sqrt[4]{x}} = \cdots = x\sqrt[20]{x^{19}}$$

$$\frac{5\sqrt{7}+7\sqrt{5}}{\sqrt{7}+\sqrt{5}} = \cdots = \sqrt{35}$$

$$\left(\sqrt[5]{\frac{1}{x^3}-\frac{1}{y^3}}\right)^2 \sqrt[5]{\left(\frac{1}{x^3}+\frac{1}{y^3}\right)^2}\sqrt[5]{\left(\frac{xy}{\sqrt[6]{x^6-y^6}}\right)^{12}} = \cdots = 1$$

5.3.2 Non-canonical simplification

arises in topics which are important for high-schools; in particular technical high-schools prepare their students with knowledge about exponential functions, logarithms, trigonometric and hyperbolic functions, together with their respective inverse functions. The domains of these functions all are not decidable in general, i.e. they do not have a canonical simplifier. For the tutor, however, the drawbacks might not be too bad.

Expansion and contraction of logarithmic terms is a good example, where many examples are still decidable, as found in [S+98c] p.152.

$$\log\left(a^4-b^4\right)^3 = \cdots$$

$$\begin{aligned} \cdots &= \log(a+b) + 3\log(a-b) + 3\log(a^2+b^2) \\ \log\left(\frac{a^3\sqrt[6]{a^9(x-y)^5}}{b^3\sqrt[7]{b^7(x-y)^6}}\left(ab^2\right)^4\right) &= \cdots \\ \cdots &= \frac{17}{2}\log a + 4\log b - \frac{1}{42}\log(x-y) \end{aligned}$$

$$\begin{aligned} 2\log 3 + 3\log 4 - 5\log 2 &= \cdots = \log 18 \\ \frac{1}{5}\left(\log x - \log(x+y) + 3\log y - \frac{4}{7}\log xy\right) &= \cdots \\ \cdots &= \log\left(\sqrt[5]{\frac{y^2\sqrt[7]{x^3y^3}}{x+y}}\right) \end{aligned}$$

Trigonometric functions surely are the hardest topic for rewriting. Algebra systems employ various switches and modes in order to approach the results envisaged by the user. The following examples may not even be representatives for the groups found at [S+98c] p.112 (the first two examples) and p.114:

$$\begin{aligned} \frac{\cos\alpha\tan\alpha}{\sin\alpha} &= \cdots = 1 \\ \frac{(\sin\alpha)^4 - (\cos\alpha)^4}{(\sin\alpha)^2 - (\cos\alpha)^2} &= \cdots = 1 \\ \sin(2x+y) - 2\sin x\cos(x+y) &= \cdots = \sin y \\ 1 + \sin x = 2\left(\sin(\frac{x}{2} + \frac{\pi}{4})\right) &\cdots \textit{true} \end{aligned}$$

Differentiation is based on a set of rules which have been proven to be complete by [KB70] already. Difficulties, however, come up with the terms differentiation is applied to. These terms introduce just the difficulties of their respective domains as discussed above.

Differentiation on term over $\mathcal{R}$ with potences $\in \mathcal{N}$ without the bound variable being a nominator in a fraction or in a transcendental function. Examples are at [S+98d] p.61, p.62 and p.63 respectively:

$$\begin{aligned} \frac{d}{dx}\left(4x^7 + 2x + 13\right) &= \cdots = 28x^6 + 2 \\ \frac{d}{dx}\left((x+3)(x^2-3x+9)\right) &= \cdots = 3x^2 \\ \frac{d}{dl}\left((l-2x)(l-8x)x\right) &= \cdots = 2x(l-5x) \end{aligned}$$

Differentiation on rational terms as found at [S$^+$98d] p.63, p.82 and p.83:

$$\begin{aligned}\frac{d}{dx}\left(\frac{6-5x}{5-6x}-\frac{3+4x}{4-3x}+\frac{2x^2+3x+4}{4x^2-3x+2}\right) &= \cdots\\ \cdots &= \frac{11}{(5-6x)^2}-\frac{25}{(4-3x)^2}+\frac{-18x^2-24x+18}{(4x^2-3x+2)^2}\end{aligned}$$

$$\begin{aligned}\frac{d}{dx}\left((x^5-\frac{1}{x^2})^4\right) &= \cdots = 4(x^5-\frac{1}{x^2})^3(5x+\frac{2}{x^3})\\ \frac{d}{df}\left(\frac{3f-2g}{4g-7f}\right) &= \cdots = = -\frac{2g}{(4g-7f)^2}\end{aligned}$$

Differentiation on radical terms as found at [S$^+$98d] p.60, p.61, p.82, p.83 and p.61:

$$\begin{aligned}\frac{d}{dx}\left(\sqrt[7]{x^3}\,\sqrt[3]{x^7}\right) &= \cdots = \frac{58x\,\sqrt[21]{x^16}}{21}\\ \frac{d}{dx}\left(\sqrt{x\,\sqrt[3]{x\,\sqrt[4]{x}}}\right) &= \cdots = \frac{17}{24\,\sqrt[24]{6^7}}\\ \frac{d}{dx}\left(\sqrt[5]{(3x^2-x+1)^2}\right) &= \cdots = \frac{2(6x-1)}{5\,\sqrt[5]{(3x^2-x+1)^3}}\\ \frac{d}{dA}\left(\sqrt{\frac{s^2+t-A}{A^2+t^2-s}}\right) &= \cdots = \frac{A^2-t^2+s_sAs^2-2At}{2\sqrt{(A^2+t^2-s)(s^2+t-A)}}\end{aligned}$$

$$\begin{aligned}\frac{d}{dx}\left(\sqrt[5]{4x^2}-3\,\sqrt[9]{2x^8}-7x^{-\frac{1}{4}}+3x^{-\frac{1}{8}}+\sqrt{5}\right) &= \cdots\\ \cdots &= \frac{2}{5}\,\sqrt[5]{\frac{4}{x^3}}-\frac{8}{3}\,\sqrt[9]{\frac{2}{x}}+\frac{7}{4x}\,\sqrt[4]{\frac{1}{x}}-\frac{3}{8x}\,\sqrt[8]{\frac{1}{x}}\end{aligned}$$

Differentiation on transcendental terms as found at [S$^+$98d] p.137 (first four examples), p.142 (three examples),p.143 and p.144:

$$\begin{aligned}\frac{d}{dx}\left((\sin x^2)^3\right) &= \cdots = 6x\cos x^2(\sin x^2)^2\\ \frac{d}{dx}\left(\frac{11\cos 4}{10\cos 8x}\right) &= \cdots = \frac{22(2\sin 8x\cos 4x-\sin 4x\cos 8x)}{5(\cos 8x)^2}\\ \frac{d}{dx}\left(\tan\left(\sqrt[6]{x^5}-1\right)\right) &= \cdots = \frac{5}{6\,\sqrt[6]{x}(\cos\left(\sqrt[6]{x^5}-1\right))^2}\\ \frac{d}{dx}\left(\frac{\tan x}{\sqrt{\tan 2x}}\right) &= \cdots = \frac{1}{(\cos x)^2\sqrt{\tan 2x}}\end{aligned}$$

$$\begin{aligned}
\frac{d}{dx}\left(e^{-x}x^2\right) &= \cdots = \\
\frac{d}{dx}\left(\frac{e^x-1}{e^x+1}\right) &= \cdots = \\
\frac{d}{dx}\left(\frac{1}{2}ln(4x^2-7x)^{\frac{7}{6}}\right) &= \cdots = \\
\frac{d}{dx}\left(ln\sqrt{\frac{2+3x}{2-3x}}\right) &= \cdots = \\
\frac{d}{dx}\left(x^2 ln(\sin x)^2\right) &= \cdots =
\end{aligned}$$

5.3.3 Combining simplifiers: equation solving

Equation solving is modeled by rewriting in a slightly more complicated way compared to the simplification using *one* rule set as discussed so far. Combining simplifiers is tricky because termination and confluence get lost in general; this has been discussed on p.39.
An equation is solved by applying several different simplifiers in a sequence very special for a type of equation. This is briefly demonstrated by the following example.

An example may be the following rational equation in m_2

$$E = \frac{m_1 m_2 v_1^2}{2(m_1+m_2)} + \frac{m_1 m_2 v_2^2}{2(m_1+m_2)}$$

The bound variable m_2 is scattered over the whole term on the right-hand side, to the nominators as well as to the denominators. How difficult is it, to make m_2 explicit ?

The solution of this problem is surprisingly simple applying several rule sets in a way, which is very similar to the calculations one would do by hand. Moreover this method is rather general and applies for a large class of rational (but linear) equations.

The main idea one should follow in solving equations like this is, to expand the equation completely (such that no product or fraction is left). In order to do so, we first factorize all terms as far as possible:

$$E = \frac{m_1 m_2 (v_1^2 - v_2^2)}{2(m_1+m_2)}$$

Now it is easy to eliminate the fractions by multiplying with the nominators

$$2E(m_1+m_2) = m_1 m_2 (v_1^2 - v_2^2)$$

and we get the fully expanded representation:

$$2Em_1 + 2Em_2 = m_1m_2v_1^2 - m_1m_2v_2^2$$

This is the first half of the solution following a strategy which is appropriate for all rational equations. The second half depends on the type of equation we have got at this stage. In our case, luckily, it is an equation linear in m_2. And this type of equation can be solved by the following three steps. First we try to get all terms containing the bound variable to the left-hand side of the equation [19]

$$2Em_1 + 2Em_2 - m_1m_2v_1^2 - m_1m_2v_2^2 = 0$$

factor out the bound variable m_2 as far as possible [20]

$$2Em_1 + m_2(2E - m_1v_1^2 - m_1v_2^2) = 0$$

and isolate the bound variable

$$-\frac{2Em_1}{2E - m_1v_1^2 - m_1v_2^2}$$

and have the result. Finally one could try to make the term prettier, for instance factorize the nominator as far as possible.

Each of these six steps can be easily modeled by a simplifier, which can be found on p.201

Equations on integer and rational terms

Linear equations as found at [S+98a] p.90 (the first two examples) p.91 (three examples), p.92, p.93 and p.107. It is typical for such example collections, that some of the equations do not seem to be linear, and need to be transformed first. This has been discussed in 5.2. The second and third example are tricky, particularly, and challenges the mechanical check of the postcondition.

$$\begin{aligned}
3(5x-2) - 2(4x+1) &= 2(3x-5) + 5 && L = \{3\} \\
(2x+1)^3 + (x+1)^3 &= (2x+1)^2 2x + (x+2)3 + x^2 && L = \{\} \\
\frac{x-1}{(x+1)^2} &= \frac{1}{x-1} - \frac{2}{x^2-1} && L = \{\} \\
\frac{17x-51}{9} - \left(-\frac{13x-3}{6} + 11 - \frac{9x-7}{4}\right) &= 0 && L = \{3\}
\end{aligned}$$

[19] ... and all other terms to the other side: this would be done by hand. We have some difficulties to identify the terms *not* containing a certain variable(the bound variable m_2), and thus proceed by another way

[20] Applying the law of distributivity not to *all* sub-terms, but *only* to the sub-terms containing the bound variable, is a technique exceeding mere rewriting.

$$\left(\frac{1}{2}+\frac{5x}{2}\right)^2-\left(\frac{13x}{2}-\frac{5}{2}\right)^2 = -(6x)^2+29 \qquad L = \{1\}$$

$$\left(\frac{\frac{x-1}{x+1}+1}{\frac{x-1}{x+1}-\frac{x+1}{x-1}}\right) = 2 \qquad L = \{-3\}$$

$$E = \frac{m_1 m_2 v_1^2}{2(m_1+m_2)} + \frac{m_1 m_2 v_2^2}{2(m_1+m_2)} \qquad \text{solve for } m_2$$

$$m_2 = -\frac{2Em_1}{2E - m_1 v_1^2 - m_1 v_2^2}$$

$$\sigma_y = -\frac{P}{4\pi h}\left((1-\mu)\frac{x}{x^2+y^2} - \frac{(}{1}+\mu)\frac{2xy^2}{(x^2+y^2)^2}\right) \qquad \text{solve for } \mu$$

$$\mu = \frac{4\pi h\sigma_y(x^2+y^2)^2 + Px(x^2-y^2)}{Px(x^2+ey^2)}$$

Root equations are found at [S$^+$98a] p.56 (the first two examples) and [S$^+$98c] p.66:

$$\{x \in \mathcal{R}.\ \sqrt{4x+15} - \sqrt{x+3} = \sqrt{x-2}\} \qquad L = \{6\}$$

$$\{x \in \mathcal{R}.\ \sqrt{x+12} + \sqrt{x-3} = \sqrt{x+32} - \sqrt{5+x} \qquad L = \{\}$$

$$\{x \in \mathcal{R}. \sqrt{29 - \sqrt{x^2-9}} = 5\ \} \qquad L = \{5, -5\}$$

Other equations are found at [S$^+$98c] p.66 (the first two), p.68, p.185 and [S$^+$98d] p.88:

$$\{x \in \mathcal{R}.\ x^2 = 64\} \qquad L = \{8, -80\}$$

$$\{x \in \mathcal{R}.\ 36x^2 - 25 = 0\} \qquad L = \{\frac{5}{6}, -\frac{5}{6}\}$$

$$\{x \in \mathcal{R}.\ \frac{1}{x-a+b} = \frac{1}{x} - \frac{1}{a} + \frac{1}{b}\} \qquad L = \{a, -b\}$$

$$\{z \in \mathcal{C}.\ z^2 + \frac{8+2i}{1-i}z + \frac{40+20i}{1-i} = 0\} \qquad L = \{-4+2i,\ 1-7i\}$$

$$\{x \in \mathcal{R}.\ |x^3+6x^2+9x+4| < 0.001\}, x_0 = 0 \qquad L = \{-4, -1\}$$

Inequalities and equation systems are not the typical candidates to be treated by rewriting; but transforming an equational system to a normal-form, say to $\{(x,y).\ ax+by = e \wedge cx+dy = f\}$, would be preparatory work

for matching with the respective problem-type. Then a rewriting approach could be feasible. Examples of linear inequalities are found at [S$^+$98a] p.97:

$$\begin{aligned} \{x \in \mathcal{N}.\ 19x - 19 < 7(x-1)\} \quad & L = \{0\} \\ \{x \in \mathit{Prim}.\ \frac{x+3}{4} - 2 < \frac{2x+1}{7} - \frac{x-5}{8}\} \quad & L = \{2,3,5,7,11,13,17,19,23\} \end{aligned}$$

Transcendental equations are the final challenge. As already mentioned, they are particularly relevant for many topics of applied mathematics.

Goniometric equations are very hard to solve, in particular it is almost impossible to get all solutions automatically (see [YS80]). This is not nice, because there are attractive 'real-world' problems which have as a subproblem very simple and very special goniometric equations. 'Pure' exercises, i.e. without textual descriptions, are found at [S$^+$98c] p.128:

$$\begin{aligned} &\{x \in [0, 360^o[.\ 2(\sin x)^2 + 4(\cos x)^2 = 3.\ \} && \text{[S}^+\text{98c] p.128} \\ &\qquad L = \{45^o, 135^o, 225^o, 315^o\} \\ &\{x \in [0, 360^o[.\ (\tan x)^2 + 2.5 \tan x = -1\} && \text{p.128} \\ &\qquad L = \{116.6^o, 153.4^o, 296.6^o, 333.4^o\} \\ &\{x \in [0, 360^o[.\ \cos(\tfrac{\pi}{2} + x)\cos(\tfrac{\pi}{2} - x) = -\cos 2x\} && \text{p.128} \\ &\qquad L = \{35.6^o, 144.74^o, 215.26^o, 324.74^o\} \\ &\{x \in [0, 360^o[.\ \cos x + \cos 2x + cos4x + cos5x = 0\} && \text{p.128} \\ &\qquad L = \{30^o, 60^o, 90^o, 150^o, 180^o, 210^o, 270^o, 300^o, 330^o\} \end{aligned}$$

Exponential and logarithmic equations are found at [S$^+$98c] p.157 and p.167 (the last one):

$$\begin{aligned} \{x \in \mathcal{R}.\ 3^{2x-3} - 3^{x-1} = 3^{2x-5} - 3^{x-3}\} \quad & L = \{2\} \\ \{x \in \mathcal{R}.\ 7^{4x+1} + 4\ 5^{3x} = 5^{3x+3} + 7^{4x-1}\} \quad & L = \{0.3787\} \\ \{x \in \mathcal{R}.\ \log x^5 - \log x^2 = \log 8\} \quad & L = \{2\} \\ \{x \in \mathcal{R}.\ (\log(x+1)) = \log(x+1)\} \quad & L = \{0, 9\} \\ \{\beta \in \mathcal{R}.\ A = \frac{a^2}{4b}\left(e^{2\beta b} - e^{2\alpha b}\right)\} \quad & \beta = \frac{1}{2b}\ {}^e\log\left(\frac{4Ab}{a^2} + e^{2\alpha b}\right) \end{aligned}$$

5.3.4 Conclusions and future work

The scan over all examples in a typical textbook for high-school mathematics can be quantified as follows: About 30% of the topics have been considered as not suited for rewriting: vectorspaces, trigonometry, statistics and probability theory. The other 70% are considered suitable. From these 70% about 40% of the examples are 'pure' exercises, where the description is (almost) identical with the formalization. These examples, covering the basic

skills, are the subproblems of the other 60%, i.e. because of the combination of subproblems the modeling and specification-phase become prevalent. Of course, the basic subproblems are also used in trigonometry and other topics 'not suitable for rewriting' – this leads to the estimation: *rewriting is the basic mechanism for solving more than 70% of high-school mathematics.*

It needs to be noted, that problems not yet discussed will be encountered, e.g. the problem how to handle partial terms like $\frac{x^2-1}{\sqrt{x-1}}$; an interesting approach for this problem is to be found in [Bee95].

Many fundamental design-decisions are still to do, apart from the remarks on notation already given above. For instance, should differentiation $\frac{d}{dx}$ be (1) a high-order function mapping functions to functions or (2) a function with signature $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$?
(1) would allow to express the chain-rule of differentiation separately; on the other hand this would introduce notation of the λ-calculus: differentiation of the identity function would look like $\frac{d}{dx}\lambda x.\, x$, which is not like traditional mathematics notation.
(2) would allow to maintain traditional notation $\frac{d}{dx}\, x$; on the other hand this would enforce to include the chain rule into the derivative of each function, e.g. $\frac{d}{dx} \sin u = \cos u \cdot \frac{d}{dx} u$.

Mechanization of mathematics also puts didactical considerations into a new light: If there is a rather homogeneous group of examples, say linear equations, and suddenly there are a few examples involving the notion of absolute value in-between the others, this raises the question w.r.t. mechanization of the group: should the method solving the whole group be overloaded with the knowledge necessary for the few additional examples, or should these examples put into another problem-type solved by another method (which thus may be simpler) ? This technical question has a didactic equivalent: Do the examples out of line pose a challenge, or do they cause bewilderment ?
The author is convinced, that there is a bulk of similar cases, just not yet realized, and waiting for discovery upon mechanization of the respective group of examples.

Facing the survey on mathematics topics, a key question for future development of the tutor may be raised: *What is the effort for 're-engineering CAS', and is this effort necessary and worthwhile ?*

The effort comprises several functions already established in algebra systems (CAS):
(1) the design of several, special purpose simplifiers, which is indispensable for meeting the requirement of stepwise calculation.
(2) the implementation of factorization (and eventually others) as 'reverse rewriting', delivering rules like a simplifier; this is necessary for stepwise calculation, too.

(3) the implementation of canonical simplifiers for each domain, which is not concerned with stepwise mode, but is needed to check formulae, input by the user, for correctness (i.e. for equivalence between the input and the formula computed by the system).

Thus, (1) and (2) really require a re-engineering of CAS functions; with (3) it may be questioned, whether to newly implement that functions, or provide for an interface calling a CAS.

5.4 Examples: proof-trees, work-sheets, scripts, etc.

This section collects examples of different kind. One group concerns examples which drove the specification of user requirements during the years; some of the examples were pushed out of the scope of the thesis, some are hard at the limit (and leave issues for future development), and some became memebers of the demo-version of the prototype.
The presentation of the examples is also taken as an occasion to note design issues for authors in order to acchieve didactic aims stated in the user requirements.
Many of the examples have been mentioned in this thesis already; this section tries to complete their respective presentation.

5.4.1 Examples on induction

are not the typical 'example construction problems' (p.46) this thesis is dedicated to; actually one has to be carful, not to mix up the language levels (e.g. what would the *find* in the problem-type be: a 'true' ? And then, what the post-condition relating *given* and *find* ?)

However, induction on the naturals $\mathcal{N}$ is one of the basic proof techniques (if any), a high-school student is taught. Thus it is desirable to have the tutor capable of handling simple proofs like induction. In fact, there is no problem to map the proof to the proof-tree as defined in Def.2.3.1; the tow branch-types (Def.2.3.2) *Transitive* and *And* are used:

form $= \forall n \in \mathcal{N}.\ \sum_{i=1}^{n} i = n(n+1)/2$
rule = *Method "natural-induction"*
branch= Transitive
  form $= \forall n \in \mathcal{N}.\ \sum_{i=1}^{n} i = n(n+1)/2$
  rule = *rewrite* $((P(0) \wedge \forall n.P(n) \Rightarrow P(n+1)) \Rightarrow P(n))$
  branch= None
  result $= (\sum_{i=1}^{1} i = 1(1+1)/2 \ \wedge$
    $\forall n.\ \sum_{i=1}^{n} i = n(n+1)/2 \ \Rightarrow \sum_{i=1}^{(n+1)} i = (n+1)((n+1)+1)/2)$
    $\Rightarrow \sum_{i=1}^{n} i = n(n+1)/2$
  ,
  form $= (\sum_{i=1}^{1} i = 1(1+1)/2 \ \wedge$
    $\forall n.\ \sum_{i=1}^{n} i = n(n+1)/2 \ \Rightarrow \sum_{i=1}^{(n+1)} i = (n+1)((n+1)+1)/2)$
    $\Rightarrow \sum_{i=1}^{n} i = n(n+1)/2$
  rule = *Split_And*
  branch= And [*"base-case", "induction-step"*]
    form $= \sum_{i=1}^{1} i = 1(1+1)/2$
    rule = *Rewrite_Set simplify_nat*
    branch= Transitive
      form $= \sum_{i=1}^{1} i = 1(1+1)/2$
      rule = *Rewrite*$(\sum_{i=1}^{1} i = 1)$

branch= None
result = $1 = 1(1+1)/2$
,
form = $1 = 1(1+1)/2$
rule = *Rewrite*$(1a = a)$
branch= None
⋮
result = *true*
result = *true*
, (* 2nd branch of And *)
form = $\sum_{i=1}^{(n+1)} i = (n+1)((n+1)+1)/2)$
rule = *Rewrite*(*sum_def_indstep*)
branch= None
⋮
result = *true*
result = *true*
result = *true*

The examples usually solved within this problem-type are

$\sum(\lambda i.\ i^2)\ n = n(n+1)(2n+1)/6$
$\sum(\lambda i.\ i^3)\ n = n^2(n+1)^2/4$
$\sum(\lambda i.\ 2i-1)\ n = n^2$
...

where $\sum$ may be defined (in Isabelle notation) as

$\sum$: [nat=>nat, nat] => nat
$\sum f\ 0 = 0$ *sum_def_base*
$\sum f(n+1) = f(n+1) + \sum\ f\ n$ *sum_def_indstep*

Another group of simple proofs is induction on lists, as taught in introductory courses on functional programming. This is an example for such a proof as presented on the work-sheet, if the student lets the tutor be active, and only asks for the rules applied in the induction-step:

rev (xs @ ys) = (rev ys) @ (rev xs)
1. *Assume (rev (xs @ ys) = (rev ys) @ (rev xs))*
2. *base-case ∧ induction-step*
2.1. *rev ([] @ ys) = (rev ys) @ (rev [])*
2.1.1. *rev ys = (rev ys) @ (rev [])*
2.1.2. *rev ys = (rev ys) @ []*
2.1.3. *rev ys = (rev ys)*
2.1.4. *true*
2#. *And*
2.2. *rev (x::xs @ ys) = (rev ys) @ (rev x::xs)* *Rewrite rev_ind*
2.2.1. *rev (x::xs @ ys) = (rev ys) @ ((rev xs) @ [x])* *Rewrite app_assoc*
2.2.2. *rev (x::xs @ ys) = ((rev ys) @ (rev xs)) @ [x]* *Apply_Asumption*

2.2.3. *rev (x::xs @ ys) = (rev (xs @ ys)) @ [x]* *Rewrite app_ind*
2.2.4. *rev (x::(xs @ ys)) = (rev (xs @ ys)) @ [x]* *Rewrite rev_ind*
2.2.5. *(rev (xs @ ys)) @ [x] = (rev (xs @ ys)) @ [x]*
2.2.6. *true*
2#. *true*
true

The theorems and the rule-set used in this proof are

list_induction $= (P([]) \wedge (P(xs) \Rightarrow P(x :: xs))) \Rightarrow P(xs)$
ruleset equ_logic
equ_true *= ((a=a) = true)*
...
rules in ruleset list_thms
rev_base *= rev [] = []*
rev_ind *= rev x::xs = (rev xs) @ [x]*
app_base *= [] @ ys = ys*
app_ind *= x::xs @ ys = x::(xs @ ys)*
app_right_id *= xs @ [] = xs*
app_assoc *= (xs @ ys) @ zs = xs @ (ys @ zs)*

In general, an induction step requires creative proof steps which cannot be automated. In order to get a script solving a major class of examples, a construct 'hint' could be introduced which covers the minimum of information unique for a particular example, added to the hidden information on each example.

5.4.2 Reasoning in calculations

Typical high-school calculations try to challenge the students by problems to be solved by break them down into sub-problems. There are few problem-types, all of which can be solved in a straight forward manner, and which are exercised thourougly. In the following example which has already been mentioned on p.2.3.1, these subproblems are

$f_x = \frac{d}{dx}(x^3 - y^3 - 3x + 12y + 10) = \cdots = 3x^2 - 3$
$f_y = \frac{d}{dy}(x^3 - y^3 - 3x + 12y + 10) = \cdots = -3y^2 + 12$
$solve\_equ((3x^2 - 3 = 0, -3y^2 + 12 = 0), (x, y)) = \cdots$
$\cdots = \{(-1, -2), (-1, 2), (1, -2), (1, 2)\}$
$f_{xx} = \frac{d}{dx}(3x^2 - 3) = \cdots = 6x$
$f_{yy} = \frac{d}{dy}(-3y^2 + 12) = \cdots = -6y$
$f_{xy} = \frac{d}{dy}6x = \cdots = 0$
$Substitute\ [(f_{xx}, 6x), (f_{yy}, -6y), (f_{xy}, 0)]\ (f_{xx}(a, b) * f_{yy}(a, b) - (f_{xy}(a, b))^2 > 0)$

where the latter substitution has to be done for all (a, b) solving the equational system, i.e. for $\{(-1, -2), (-1, 2), (1, -2), (1, 2)\}$, to evaluate the inequality for all substitutions, and to take all solution evaluating to true as the solutions of the root-problem.
The issue for designing the solving process for this kind of example is: Incorporate the 'theory'-knowledge the student is taught to solve the example,

such that 'reflection' can work: That means, if somewhere in the calculation the student asks *HowCome* or *WhatFor*, then an automated generation of explanations should be possible. The following extract of a work-sheet suggests such a design:

problem: 'max-on-surface'
given : [$x^3 - y^3 - 3x + 12y + 10,\ (x, y)$]
where : [$(\lambda(x, y).\ x^3 - y^3 - 3x + 12y + 10)$ is-diffable-on R^2]
find : [maxs $\subset R^2$]
with : [maxs $= \{(a, b).\ (a, b)$ is-local-max-of $(\lambda(x, y).\ x^3 - y^3 - 3x + 12y + 10)\ \}$]
1. we use ruleset analysis_thms
1.1. $\{(a, b).\ (a, b)$ is-local-max-of-in f(x,y) $\} =$
1.2. $= \{(a, b).\ (a, b)$ is-local-extremum-of-in $f(x, y)\ \wedge\ f_{xx}(a, b) < 0\} =$
1.3. $= \{(a, b).\ f_x(a, b) = 0 \wedge f_y(a, b) = 0 \wedge f_{xx}(a, b) * f_{yy}(a, b) - (f_{xy}(a, b))^2 > 0$
$\wedge f_{xx}(a, b) < 0\} =$
2. we group $\wedge$ by $[2, 1, 1]$
$= \{(a, b).\ (f_x(a, b) = 0 \wedge f_y(a, b) = 0)\ \wedge\ (f_{xx}(a, b) * f_{yy}(a, b) - (f_{xy}(a, b))^2 > 0)$
$\wedge (f_{xx}(a, b) < 0)\} =$
3. we use theorem *thm*$\wedge$*to*$\cap$
$= \{(a, b).\ (f_x(a, b) = 0 \wedge f_y(a, b) = 0\}$
$\cap \{(a, b).\ f_{xx}(a, b) * f_{yy}(a, b) - (f_{xy}(a, b))^2 > 0)\}$
$\cap \{(a, b).\ f_{xx}(a, b) < 0)\}$
4. we calculate the sets from left to right
4.1. $\text{set}_1 = \{(a, b).\ (f_x(a, b) = 0 \wedge f_y(a, b) = 0\}$
4.1.1. $f_x = \frac{d}{dx}(x^3 - y^3 - 3x + 12y + 10) =$
4.1.1.1. $= \frac{d}{dx}x^3 - \frac{d}{dx}(y^3 - 3x + 12y + 10) =$
4.1.1... $= \ldots$
4.1.1. $= 3x^2 - 3$
4.1.2. $f_y = \frac{d}{dy}(x^3 - y^3 - 3x + 12y + 10) =$
4.1.2... $\ldots$
4.1.2. $= -3y^2 + 12$
4.1.3. $L = solve\_equ((3x^2 - 3 = 0, -3y^2 + 12 = 0), (x, y))$
4.1.3... $\ldots$
4.1.3. $= \{(-1, -2), (-1, 2), (1, -2), (1, 2)\}\}$
4.1. $\text{set}_1 = \{(-1, -2), (-1, 2), (1, -2), (1, 2)\}\}$
4.2. $\text{set}_2 = set_1\ \cap\ \{(a, b).\ f_{xx}(a, b) * f_{yy}(a, b) - (f_{xy}(a, b))^2 > 0)\}$
4.2.1. we solve some subproblems
4.2.1.1. $f_{xx} = \frac{d}{dx}(3x^2 - 3)$
4.2.1.1... $\ldots$
4.2.1.1. $= 6x$
4.2.1.2. $f_{xy} = \frac{d}{dy}6x$
4.2.1.2... $\ldots$
4.2.1.2. $= 0$
4.2.1.3. $f_{yy} = \frac{d}{dy}(-3y^2 + 12)$
4.2.1.3... $\ldots$
4.2.1.3. $= -6y$
4.2.1. and substitute from the results
$\text{set}_2 = set_1\ \cap\ \{(a, b).\ 6a * 6b - 0^2 > 0)\} =$
4.2.2. $= set_1\ \cap\ \{(a, b).\ 36ab > 0)\} =$

4.2.3. we calculate the intersection $\text{set}_2 = set_1 \cap \{(a,b).\ 36ab > 0)\}$
by testing for each element:
4.2.3.1. $(-1,-2) \in \{(a,b).\ 36ab > 0)\} =$
4.2.3.1.1. $= 36 * (-1) * (-2) > 0 =$
4.2.3.1.1. $=$ true
4.2.3.2. $(-1,2) \in \{(a,b).\ 36ab > 0)\} =$
4.2.3.2.... ...
4.2.3.2.1. =false
4.2.3.3. $(1,-2) \in \{(a,b).\ 36ab > 0)\} =$
4.2.3.3.1. =false
4.2.3.4. $(1,2) \in \{(a,b).\ 36ab > 0)\} =$
4.2.3.4.1. =true
4.2. $\text{set}_2 = \{(1,2),(-1,-2)\}$
4.3. $\text{set}_3 = set_2 \cap \{(a,b).\ f_{xx}(a,b) < 0\}$
4.3.1. we substitute from the results
4.3.1.1. $\{(a,b).\ f_{xx}(a,b) < 0\} =$
4.3.1.1. $= \{(a,b).\ 6a < 0\}$
4.3.2. we calculate the intersection $\text{set}_3 = set_2 \cap \{(a,b).\ 6a < 0\}$
by testing for each element:
4.3.2.1. $(1,2) \in \{(a,b).\ 6a < 0\} =$
4.3.2.1.0. $= (6 * 1 < 0) =$
4.3.2.1.1. =false
4.3.2.2. $(-1,-2) \in \{(a,b).\ 6a < 0\} =$
4.3.2.2.1. =true
4. $\text{set}_1 \cap \text{set}_2 \cap \text{set}_3 \cap = \{(-1,-2)\}$
found : maxs $= \{(-1,-2)\}$

If the student asks, for instance, *WhatFor* at 4.1.3., the system can generate even a chain of explanations on repeated *WhatFor*-requests:

$L = \text{set}_1 \quad \cdots$
$\text{set}_2 = set_1 \cap \{(a,b).\ f_{xx}(a,b) * f_{yy}(a,b) - (f_{xy}(a,b))^2 > 0)\} \quad \cdots$
$\text{set}_3 = set_2 \cap \{(a,b).\ f_{xx}(a,b) < 0\} \quad \cdots$
maxs=$\text{set}_1 \cap \text{set}_2 \cap \text{set}_3$

And on the students repeated *HowCome*-requests the sequence of automatically generated replys could be:

we calculate the sets from left to right
$\text{set}_1 = \{(a,b).\ (f_x(a,b) = 0 \wedge f_y(a,b) = 0\} \quad \cdots$

we use theorem $thm \wedge to \cap$
$\{(a,b).\ (f_x(a,b) = 0 \wedge f_y(a,b) = 0\}$
$\cap \{(a,b).\ f_{xx}(a,b) * f_{yy}(a,b) - (f_{xy}(a,b))^2 > 0)\}$
$\cap \{(a,b).\ f_{xx}(a,b) < 0)\} \quad \cdots$

we use ruleset analysis_thms
$\{(a,b).\ (a,b)$ is-local-max-of-in f(x,y) $\} =$
$= \{(a,b).\ (a,b)$ is-local-extremum-of-in $f(x,y) \ \wedge \ f_{xx}(a,b) < 0\} =$
$= \{(a,b).\ f_x(a,b) = 0 \wedge f_y(a,b) = 0 \wedge f_{xx}(a,b) * f_{yy}(a,b) - (f_{xy}(a,b))^2 > 0$

The next example sheds light on the same issue by another question: how to explain formulae on a students request just at the moment the formula is being used ? Let us assume, the student is going to find one solution for a transcendental equation, and the system presents a method iterating the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The student could use it as a black-box, but he decides to ask *HowCome* ? Of course, the student could look up somewhere else for 'Newtons method'. But the tutor should explain the method w.r.t. the *actual example*, and in the detail the students actually wants. This issue can be accomplished by opening the black-box [Buc92] in the following way:

problem 'equation-approx.'
given: $t(x) = x^2 - \sin x,\ x_0 = 1,\ \epsilon = 0.0005$
find: $\alpha \in \mathcal{R}$
with: $|t(\alpha)| < \epsilon$
1. we have problem 'derivative'

1. given: $f \in \mathcal{R}^{\mathcal{R}}$
1. where:$f(x)$ is-diffable
1. find: $f' \in \mathcal{R}^{\mathcal{R}}$
1. with: f' is-derivative-of f
1.1.1. and check the assumption: $(t(x) = x^2 - \sin x) \wedge t(x)$ is-diffable
1.1.1.*. ⋮
1.1.1.n. true
1.1.1. true
1.1. $t'(x) = \frac{d}{d.x}t(x)$
1.2. $t' = \frac{d}{d.x}(x^2 - \sin x)$
1.*. ⋮
1. found $\{\ t'(x) = 2x - \cos x\}$
2. we have problem 'linearize'
2. given: $f \in \mathcal{R}^{\mathcal{R}}, p \in \mathcal{R}$
2. where:$f(x)$ is-diffable
2. find: $g \in \mathcal{R}^{\mathcal{R}}$
2. with: $g(x)$ is-linear $\wedge g(p) = f(p) \wedge' g(p) = f'(p)$
2.1.1. we invent: $g(x) = t(x_0) + t'(x_0) * (x - x_0)$
2.1.1. and show the 'with'-part:
2.1.2.1. g(x) is-linar
2.1.2.1.*. ⋮
2.1.2.1. found: true
2.1.2.2. $g(x_0) = t(x_0)$
2.1.2.2.*. ⋮
2.1.2.2. found: true
2.1.2.3. $'g(x_0) = t'(x_0)$
2.1.2.3.*. ⋮

2.1.2.3. found: true
2.1.2. found: true
2.1. found: $g(x) = t(x_0) + t'(x_0) * (x - x_0)$
2. found: $\{\ g(x) = t(x_0) + t'(x_0) * (x - x_0)\}$
3. we have problem 'linear-equation'
3. given: $\mathcal{D}, \sqcup \in \text{Term}[\mathcal{D}]$
3. given: $\mathcal{D}$ is-field $\wedge$ $t(x)$ is-linear
3. find: $\alpha \in \mathcal{D}$
3. with: $t(\alpha) = 0$
3.1. we use rule-set 'equivalence-transformations'
3.1.1. $t(x_0) + t'(x_0) * (x - x_0) = 0$
3.1.2. $\vdots$
3.1.n. $x = x_0 - \frac{t(x_0)}{t'(x_0)}$
3.1. found: $x = x_0 - \frac{t(x_0)}{t'(x_0)}$
3. found: $\{\ x = x_0 - \frac{t(x_0)}{t'(x_0)}\}$
4. we start iteration: $x \to x_1, x_1 = x_0 - \frac{t(x_0)}{t'(x_0)}$
4. and iterate: $i \to i + 1, x_i \to x_{i+1}, x_{i+1} \to \alpha$
4.1. iteration with i = 0
4.1.1. $x_1 = x_0 - \frac{t(x_0)}{t'(x_0)}$
4.1.*. $\vdots$
4.1.n. $x_1 = 0.891395$
4.1. $\alpha = 0.891395, |t(\alpha)| = 0.0166372 \not< \epsilon$
4.2. iteration with i = 1
4.2.1. $x_2 = x_1 - \frac{t(x_1)}{t'(x_1)}$
4.2.*. $\vdots$
4.2.n. $x_2 = 0.876984$
4.2. $\alpha = 0.876984, |t(\alpha)| = 0.0002881 < \epsilon$
4. found: $\alpha = 0.877$
found: $\alpha = 0.877$

This example is out of the tutors scope because numerals in $\mathcal{R}$ with digits are not available in Isabelle yet.

These two examples showed how to provide a 'theory'-framework for calculations, in order to automatically generate explanations. It is the authors strong belief, that the issue of generating explanations automatically, coincides with the issue to *proof the correctness of a script w.r.t. the post-condition* and the pre-condition given in the guard. This issue is considered very important, but out of the scope of this thesis.

5.4.3 A collection of scripts

During the development of the mathematics-engine and the dialog-guide there was little space for scripts. Thus some typical examples are collected here. The first kind of scripts is concerned with guiding the dataflow to and from subproblems. The following examples do that for the problem-types already presented for the 'maximum-example' on p.53:

```
Script Maximum_value (fix_::bool list) (m_::real) (rs_::bool list)
            (v_::real) (itv_::real set) (err_::bool) =
   (let
      e_ = (hd o (filter (Testvar m_))) rs_;
      t_ = (if #1 < Length rs_
         then (make_fun (Reals,[make,function],no_met) m_ v_ rs_)
         else (hd rs_));
      mx_ = max_on_interval (Reals,[on_interval,max_of,function],
            maximum_on_interval) t_ v_ itv_
   in (find_vals (Reals,[find_values,tool],find_values)
            mx_ (Rhs t_) v_ m_ (dropWhile (ident e_) rs_)))
```

The script calls the subproblem `make_fun` only if there are more than two relations `rs_` given, then calculates the maximum `mx_` solving the problem `[on_interval,max_of,function]` by use of the method `maximum_on_interval`, and finally determines the requested results by calling `find_vals`.

One source of the systems flexibility is the problem-tree featuring search for the most appropriate problem-type; if the problem-type is found, the method can be chosen from an associated list. In the case above the call `make_fun` has as the first argument the specification `(Reals,[make,function], no_met)`, i.e. there is no method specified. This is the signal for the system to *refine* (Def.2.2.6) the specification: if the user decides for variant II (p.12) of the maximum-example

variant II
$[\,[R = ArbFix],\ \ A,\ \ [A = 2ab - a^2,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = R^2],$
$\quad b,\ \ \{x.\ 0.0 \leq x \leq \frac{R}{2}\},\ \ (err = \#0)\,]$

then the *refine*d problem-type will lead to the associated method, called as `Make_fun_by_explicit` $A\ \ a\ \ [A = 2ab - a^2,\ (\frac{a}{2})^2 + (\frac{b}{2})^2 = R^2]$, where `Make_fun_by_explicit` is defined by the script

```
Script Make_fun_by_explicit (f_::real) (v_::real) (eqs_::bool list) =
   (let
      h_ = (hd o (filter (Testvar m_))) eqs_;
      e1_ = hd (dropWhile (ident h_) eqs_);
      vs_ = dropWhile (ident f_) (Var h_);
      v1_ = hd (dropWhile (ident v_) vs_);
      s1 = (solve_univar (Reals, [univar,equation], no_met) e1_ v1_)
   in Substitute [(v_1, (Rhs o hd) s_1)] h_)
```

If, however, the user had decided for variant III,

variant III
$[\,[R = ArbFix],\ \ A,\ \ [A = 2ab - a^2,\ \frac{a}{2} = R\sin\alpha, \frac{b}{2} = R\cos\alpha],$
$\quad \alpha,\ \ \{x.\ 0.0 \leq x \leq \frac{\pi}{2}\},\ \ (err = \#0)\,]$

and thus called

`Maximum_value` $[R = ArbFix]\ \ A\ \ [A = 2ab - a^2,\ \frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha]$
$\quad \alpha\ \ \{x.\ 0.0 \leq x \leq \frac{\pi}{2}\}\ \ (err = \#0)$

which refined problem-type `[make,function]` and thus calls the according method by

`Make_fun_by_new_variable` $A\ \alpha\ [A = 2ab - a^2,\ \frac{a}{2} = R\sin\alpha,\ \frac{b}{2} = R\cos\alpha]$

with the script

```
Script Make_fun_by_new_variable (f_::real) (v_::real)
                                (eqs_::bool list) =
   (let
      h_ = (hd o (filter (Testvar m_))) eqs_;
      es_ = dropWhile (ident h_) eqs_;
      vs_ = dropWhile (ident f_) (Var h_);
      v1_ = Nth #1 vs_;
      v2_ = Nth #2 vs_;
      e1_ = (hd o (filter (Testvar v1_))) es_;
      e2_ = (hd o (filter (Testvar v2_))) es_;
      s_1 = (solve_univar (Reals, [univar,equation], no_met) e1_ v1_);
      s_2 = (solve_univar (Reals, [univar,equation], no_met) e2_ v2_)
   in Substitute [(v_1, (Rhs o hd) s_1),(v_2, (Rhs o hd) s_2)] h_)
```

The latter script shows the case where two tactics can be done in parallel (but must be done): `solve_univar` has two disjunct argument lists.

Other kinds of scripts describe parallel execution of tactics, where *some* of the tactics *can* be applied; an example is rewriting by a complete rule-set given already on p.**??**.
Here a last example is given which shows the combination of parallel execution in `Rewrite_Set` and execution in a mandatory sequence established by two nested `let`:

```
Script square_equation (eq_::bool) (v_::real) (err_::real) =
   (let e_ =
      (while (not o is_root_free) do
         %e_.  (let
            e_ = try (Rewrite_Set simplify False) eq_;
            e_ = try (repeat (Rewrite assoc_plus_inv False)) e_;
            e_ = try (repeat (Rewrite assoc_mult_inv False)) e_;
            e_ = try (Rewrite_Set isolate_root False) e_;
            in ((Rewrite square_equation_left True) or
               (Rewrite square_equation_right True)) e_)
         eq_);
      e_ = try (Rewrite_Set_Inst [(bdv,v_)] norm_equation False) e_;
      L_ = solve_univar (Reals, [equation,univariate], no_met) e_ v_ err_
   in Check_elementwise L_ Assumptions)
```

This example, shows a rather complete collection of script-expressions; actually it was the primary example in prototyping.

5.4.4 Rewriting in Mathematica

Mathematica employs very elegant techniques to make symbolic computations easy to handle. In particular, it rewrites modulo associativity, and it simplifies each term into a normal form immediately after input [21] . This is extremely useful, because a normal form is the best prerequisite for any mechanical manipulation. Actually, this mechanism leads to very short rule-sets, as shown in the sequel. On the other hand, some kinds of examples cannot be done, for instance those involving numeral constants as listed on p.177.

Looking back to the above example of root-equations solved by Isabelles rewriter, we note two tactics which are not necessary with Mathematica:
(1) `(Rewrite_Set real_simplify_p False e_)` produces polynomial form; this includes ordering w.r.t. a suitable term-order, and the related AC-operators necessarily arrange terms associated like $(a + (b + \cdots (y + z)))$. Because + associates to the right, normally, these parentheses are displayed, which may confuse a beginner. Thus for removing the ( ) another rule-set is employed, too:
`(Rewrite_Set rearrange_assoc False e_)` produces the arrangement $(\cdots (a+ b) + \cdots y) + z$, which can be displayed without the parentheses.

The following examples stem from an early prototype implemented in Mathematica. Mathematicas automated simplification is annotated by 'auto.simp.'.

1. $L = \{x \in \mathcal{R}.\ \sqrt{9+4x} = \sqrt{x} + \sqrt{5+x}\} =$
2. we repeat until *not square-rooted*

ruleset *real-simplify* not applicable
ruleset *isolate-root* not applicable
$(a = b) = a^2 = b^2$ *square-equ*

2.1. $= \{x \in \mathcal{R}.\ (\sqrt{9+4x})^2 = (\sqrt{x} + \sqrt{5+x})^2\} =$

auto.simp.

2.1. $= \{x \in \mathcal{R}.\ 9 + 4x = (\sqrt{x} + \sqrt{5+x})^2\} =$
2.2. we apply *ruleset real-simplify*

$(a + b)^2 = a^2 + 2ab + b^2$

2.2.1. $= \{x \in \mathcal{R}.\ 9 + 4x = (\sqrt{x})^2 + 2\sqrt{x}\sqrt{5+x} + (\sqrt{5+x})^2\} =$

auto.simp.

2.2.1. $= \{x \in \mathcal{R}.\ 9 + 4x = 5 + 2x + 2\sqrt{x}\sqrt{5+x}\} =$

$\sqrt{a}\sqrt{b} = \sqrt{ab}$

2.2.2. $= \{x \in \mathcal{R}.\ 9 + 4x = 5 + 2x + 2\sqrt{x(5+x)}\} =$

$a(b + c) = ab + ac$

2.2.3. $= \{x \in \mathcal{R}.\ 9 + 4x = 5 + 2x + 2\sqrt{x5 + xx}\} =$

auto.simp.

2.2.3. $= \{x \in \mathcal{R}.\ 9 + 4x = 5 + 2x + 2\sqrt{5x + x^2}\} =$
2.3. we apply *ruleset isolate-root*

$(a = b + c\sqrt{d}) = (\sqrt{d} = \frac{a-b}{c})$

2.3.2. $= \{x \in \mathcal{R}.\ \sqrt{5x + x^2}\} = \frac{(9+4x)-(5+2x)}{2} =$

auto.simp.

[21] The automated simplification can be circumvented by an alteration of the system-calls at input and output, the rewriting modulo associativity can be circumvented by an appropriate setting of the 'flatable'- attribute of the respective operation

2.4.5. $= \{x \in \mathcal{R}.\ \sqrt{5x + x^2}\} = \frac{1}{2}(4 + 2x) =$

$(a = b) = a^2 = b^2$ *square-equ*

2.5. $= \{x \in \mathcal{R}.\ (\sqrt{5x + x^2})^2\} = (\frac{1}{2}(4 + 2x))^2 =$

auto.simp.

2.5. $= \{x \in \mathcal{R}.\ 5x + x^2 = \frac{1}{4}(4 + 2x)^2 =$

ruleset *real-simplify* applicable !!!

$(a + b)^2 + a^2 + 2ab + b^2$

2.6.1. $= \{x \in \mathcal{R}.\ 5x + x^2 = \frac{1}{4}(4 * 4 + 2 * 4 * 2x + (2x)^2)$

auto.simp.

2.6.1. $= \{x \in \mathcal{R}.\ 5x + x^2 = \frac{1}{4}(16 + 16x + 4x^2)$

$a(b + c) = ab + ac$

2.6.2. $= \{x \in \mathcal{R}.\ 5x + x^2 = \frac{1}{4}16 + \frac{1}{4}(16x + 4x^2)$

auto.simp.

2.6.2. $= \{x \in \mathcal{R}.\ 5x + x^2 = 4 + \frac{1}{4}(16x + 4x^2)$

$a(b + c) = ab + ac$

2.6.3. $= \{x \in \mathcal{R}.\ 5x + x^2 = 4 + \frac{1}{4}16x + \frac{1}{4}4x^2$

auto.simp.

2.6.3. $= \{x \in \mathcal{R}.\ 5x + x^2 = 4 + 4x + x^2$

ruleset *isolate-root* not applicable

square-equ not applicable

3. we apply *ruleset normalize-equation*

$\{x.\ a = bx + c\} = \{x.\ a - bx = c\}$ applies !!!

3.1. $= \{x \in \mathcal{R}.\ (5x + x^2) - 4x = 4 + x^2\} =$

auto.simp.

3.1. $= \{x \in \mathcal{R}.\ x + x^2 = 4 + x^2\} =$

$\{x.\ a = x^n + c\} = \{x.\ a - x^n = c\}$

3.2. $= \{x \in \mathcal{R}.\ (x + x^2) - x^2 = 4\} =$

auto.simp.

3.2. $= \{x \in \mathcal{R}.\ x = 4\} =$

$\{x.\ a = b\} = \{x.\ a - b = 0\}$

3.3. $= \{x \in \mathcal{R}.\ x - 4 = 0\} =$

4. we solve subproblem *linear-equation*

4.1. $= \{x \in \mathcal{R}.\ x = 4\}$

found: $L = \{4\}$

Rewriting in this example is done by the rule-sets

real-simplify

$(a + b)^2 = a^2 + 2ab + b^2$
$\sqrt{a}\sqrt{b} = \sqrt{ab}$
$a(b + c) = ab + ac$
$(a + b)(c + d) = ac + bc + ad + bd$

isolate-root

precondition: $\neg(matching(equ, \sqrt{\text{--}} = \text{--}) \ \vee \ matching(equ, \sqrt{\text{--}} = \text{--}))$
$(\sqrt{a} + b = c) = (\sqrt{a} = c - b)$
$(a\sqrt{b} = c) = (\sqrt{b} = c/a)$
$(a = b + c\sqrt{d}) = (\sqrt{d} = (a - b)/c$

normalize-equation

$a(b+c) = ab + ac$
$(a+b)^2 = a^2 + 2ab + b^2$
$\{x.\ a = x + c\} = \{x.\ a - x = c\}$
$\{x.\ a = bx + c\} = \{x.\ a - bx = c\}$
$\{x.\ a = x^n + c\} = \{x.\ a - x^n = c\}$
$\{x.\ a = bx^n + c\} = \{x.\ a - bx^n = c\}$
$\{x.\ a = b\} = \{x.\ a - b = 0\}$

The other example has already be mentioned on p.39; here are more details, presented as a work-sheet, again without user-interaction:

$L = \{m_2 \in \mathcal{R}.\ E = \frac{m_1 m_2 v_1^2}{2(m_1+m_2)} + \frac{m_1 m_2 v_2^2}{2(m_1+m_2)}\} \quad =$

Rewrite_Set *factorize*

1. we apply *ruleset factorize*

Rewrite $a + \frac{b}{c} = \frac{b+ac}{c}$

1.1. $m_2 \in \mathcal{R}.\ E = \frac{\frac{1}{2}m_1 m_2 v_1^2}{2(m_1+m_2)} + \frac{\frac{1}{2}m_1 m_2 v_2^2}{m_1+m_2}\} \quad =$

Rewrite $a + b\frac{c}{d} = \frac{b+acd}{d}$

1.2. $= \{m_2 \in \mathcal{R}.\ E = \frac{m_1 m_2 (v_1^2 - v_2^2)}{2(m_1+m_2)}\} \quad =$

Rewrite_Set *elim-denominators*

2. we apply *ruleset elim-denominators*

Rewrite $(a = \frac{c}{d}) \ = \ (ad = c)$

2.1. $= \{m_2 \in \mathcal{R}.\ E(m_1+m_2) = \frac{1}{2}m_1 m_2 (v_1^2 - v_2^2)\} \quad =$

Rewrite $(a = b\frac{c}{d}) \ = \ (ad = bc)$

2.2. $= \{m_2 \in \mathcal{R}.\ 2E(m_1+m_2) = m_1 m_2 (v_1^2 - v_2^2)\} \quad =$

Rewrite_Set *expand*

3. we apply *ruleset expand*

Rewrite $a(b+c) = ab + ac$

3.1. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + 2Em_2 = m_1 m_2 (v_1^2 - v_2^2)\} \quad =$

Rewrite $a(b+c) = ab + ac$

3.2. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + 2Em_2 = m_1 m_2 v_1^2 - m_1 m_2 v_2^2\} \quad =$

Rewrite_Set *var-to-left*

4. we apply *ruleset var-to-left*

Rewrite $(a = bx + c) \ = \ (a - x = c)$

4.1. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + 2Em_2 - m_1 m_2 v_1^2 = \sim m_1 m_2 v_2^2\} \quad =$

Rewrite $(a = bx) \ = \ (a - bx = 0)$

4.2. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + 2Em_2 - m_1 m_2 v_1^2 - m_1 m_2 v_2^2 = 0\} \quad =$

Rewrite_Set *factor-var*

5. we apply *ruleset factor-var*

Rewrite $ax + bx = (a+b)x$

5.1. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + m_2(2E - m_1 v_1^2) - m_1 m_2 v_2^2 = 0\} \quad =$

Rewrite $ax + bx = (a+b)x$

5.2. $= \{m_2 \in \mathcal{R}.\ 2Em_1 + m_2(2E - m_1 v_1^2 - m_1 v_2^2) = 0\} \quad =$

Rewrite_Set *isolate-var*

6. we apply *ruleset isolate-var*

Rewrite $(a + cx = b) \ = \ cx = -a + b$

6.1. $= \{m_2 \in \mathcal{R}.\ m_2(2E - m_1 v_1^2 - m_1 v_2^2) = \sim 2Em_1\} \quad =$

Rewrite $(ax = b) \;=\; x = \frac{b}{a}$

6.2. $= \{m_2 \in \mathcal{R}.\ m_2 = \sim \frac{2Em_1}{2E - m_1 v_1^2 - m_1 v_2^2}\}$
found: $L = \{\sim \frac{2Em_1}{2E - m_1 v_1^2 - m_1 v_2^2}\}$

The rule-sets used by *Rewrite_Set* above are

factorize

$$\begin{array}{lcl} ac + bc & = & (a + b)c \\ a + b/c & = & (ac + b)/c \\ a + \frac{c}{d}b & = & (acd + b)/d \end{array}$$

elim-denominators

$$\begin{array}{lcl} (a/b = c/d) & = & (ad = cb) \\ (a = c/d) & = & (ad = c) \\ (a = \frac{c}{d}b) & = & (ad = cb) \\ (a = bc^{-n}) & = & (ac^n = b) \end{array}$$

expand

$$a(b + c) \;=\; ab + ac$$

var-to-left

$$\begin{array}{lcl} \{x.\ a = x + c\} & = & \{x.\ a - x = c\} \\ \{x.\ a = bx + c\} & = & \{x.\ a - bx = c\} \\ \{x.\ a = x\} & = & \{x.\ a - x = 0\} \\ \{x.\ a = bx\} & = & \{x.\ a - bx = 0\} \end{array}$$

factor-var

$$\begin{array}{l} ax + bx = (a + b)x \\ ax + x = (a + 1)x \end{array}$$

isolate-var

$$\begin{array}{lcl} \{x.\ a + cx = b\} & = & \{x.\ cx = b - a\} \\ \{x.\ a + x = b\} & = & \{x.\ x = b - a\} \\ \{x.\ ax = b\} & = & \{x.\ x = b/a\} \end{array}$$

The proof script is a simple, linear sequence:

```
Script make_explicit e_ =
   let
      e_ = try Rewrite_Set False factorize e_;
      e_ = try Rewrite_Set False elim-denominators e_;
      e_ = try Rewrite_Set False expand e_;
      e_ = try Rewrite_Set False var-to-left e_;
      e_ = try Rewrite_Set False factor-var e_;
   in try Rewrite_Set False isolate-var e_
```

Appendix A

ABBREVATIONS

ACDCA Austrian Center for Didactics of Computer Algebra

AMMU Arbeitskreis moderner Mathematik-Unterricht

BNF Backus normalform

CTP computer theorem proving, or computer theorem prover

CAD Computer-aided Design

CAS computer algebra system

CSCW computer supported cooperative work

DG dialog guide

HCI human computer interaction

KBS knowledge based system

ME mathematics engine

MSCM man-system cooperation model

PSM problem specification module

T^3 Teacher Training with Technology

Appendix B

ISABELLE SYNTAX AND SEMANTICS FOR SCRIPTS

Here these parts of Isabelles syntax and semantics are presented, which are used by the scripts. The prototype designed within this thesis is based on high-order logic, its implementation uses the release Isabelle99.

B.1 Propositions

The scripts BNF refers to *form* briefly explained as 'constructed with the usual logical operators'. These are defined in Isabelles basic theory on high-order logic, *HOL.thy*. This theory also contains the definition of `if then else` used by scripts, and the definition of *arbitrary* necessary for functions on lists.
The relevant parts of this theory are the following:

```
(*  Title:    HOL/HOL.thy
    ID:       $Id: HOL.thy,v 1.47 1999/09/01 19:24:50 wenzelm Exp $
    Author:   Tobias Nipkow
    Copyright 1993  University of Cambridge

Higher-Order Logic.
*)

typedecl bool

arities
  bool :: "term"

consts
  Not       :: "bool => bool"           ("~ _" [40] 40)
  True      :: bool
  False     :: bool

  "="       :: "['a, 'a] => bool"       (infixl 50)
  &         :: "[bool, bool] => bool"   (infixr 35)
  "|"       :: "[bool, bool] => bool"   (infixr 30)
  -->       :: "[bool, bool] => bool"   (infixr 25)
```

```
  Eps       :: "('a => bool) => 'a"
  If        :: "[bool, 'a, 'a] => 'a"   ("(if (_)/ then (_)/ else (_))" 10)
  arbitrary :: 'a

syntax
  ~=        :: "['a, 'a] => bool"       (infixl 50)
  "ALL "    :: "[idts, bool] => bool"   ("(3! _./ _)" [0, 10] 10)
  "_Eps"    :: "[pttrn, bool] => 'a"    ("(3SOME _./ _)" [0, 10] 10)

defs

  True_def: "True      == ((%x::bool. x) = (%x. x))"
  False_def:"False     == (!P. P)"
  not_def:  "~ P       == P-->False"
  and_def:  "P & Q     == !R. (P-->Q-->R) --> R"
  or_def:   "P | Q     == !R. (P-->R) --> (Q-->R) --> R"

  (*arbitrary is completely unspecified, but is made to appear as a
    definition syntactically*)
  arbitrary_def:  "False ==> arbitrary == (@x. False)"
```

B.2 List-expressions

The scripts BNF refers to *listexpr* comprising expressions built by functions on lists. During execution of a script, these expressions are being evaluated by the prototypes rewrite engine. The rewrite rules are those defined by `primrec` below.

```
(*  Title:    HOL/List.thy
    ID:       $Id: List.thy,v 1.48 1999/08/16 20:07:12 wenzelm Exp $
    Author:   Tobias Nipkow
    Copyright 1994 TU Muenchen

The datatype of finite lists.
*)

datatype 'a list = Nil ("[]") | Cons 'a ('a list) (infixr "#" 65)

consts
  "@"         :: ['a list, 'a list] => 'a list      (infixr 65)
  filter      :: ['a => bool, 'a list] => 'a list
  concat      :: 'a list list => 'a list
  foldl       :: [['b,'a] => 'b, 'b, 'a list] => 'b
  hd, last    :: 'a list => 'a
  set         :: 'a list => 'a set
  list_all    :: ('a => bool) => ('a list => bool)
```

```
  map         :: ('a=>'b) => ('a list => 'b list)
  mem         :: ['a, 'a list] => bool              (infixl 55)
  nth         :: ['a list, nat] => 'a               (infixl "!" 100)
  list_update :: 'a list => nat => 'a => 'a list
  take, drop  :: [nat, 'a list] => 'a list
  takeWhile,
  dropWhile   :: ('a => bool) => 'a list => 'a list
  tl, butlast :: 'a list => 'a list
  rev         :: 'a list => 'a list
  zip         :: "'a list => 'b list => ('a * 'b) list"
  upt         :: nat => nat => nat list             ("(1[_../_'(])")
  remdups     :: 'a list => 'a list
  nodups      :: "'a list => bool"
  replicate   :: nat => 'a => 'a list

syntax
  (* Special syntax for filter *)
  "@filter"   :: [pttrn, 'a list, bool] => 'a list  ("(1[_:_ ./ _])")

primrec
  "hd([]) = arbitrary"
  "hd(x#xs) = x"
primrec
  "tl([]) = []"
  "tl(x#xs) = xs"
primrec
  "last [] = arbitrary"
  "last(x#xs) = (if xs=[] then x else last xs)"
primrec
  "butlast [] = []"
  "butlast(x#xs) = (if xs=[] then [] else x#butlast xs)"
primrec
  "x mem [] = False"
  "x mem (y#ys) = (if y=x then True else x mem ys)"
primrec
  "set [] = {}"
  "set (x#xs) = insert x (set xs)"
primrec
  list_all_Nil  "list_all P [] = True"
  list_all_Cons "list_all P (x#xs) = (P(x) & list_all P xs)"
primrec
  "map f [] = []"
  "map f (x#xs) = f(x)#map f xs"
primrec
  append_Nil  "[]    @ys = ys"
  append_Cons "(x#xs)@ys = x#(xs@ys)"
primrec
  "rev([]) = []"
  "rev(x#xs) = rev(xs) @ [x]"
```

```
primrec
  "filter P [] = []"
  "filter P (x#xs) = (if P x then x#filter P xs else filter P xs)"
primrec
  foldl_Nil  "foldl f a [] = a"
  foldl_Cons "foldl f a (x#xs) = foldl f (f a x) xs"
primrec
  "concat([]) = []"
  "concat(x#xs) = x @ concat(xs)"
primrec
  drop_Nil  "drop n [] = []"
  drop_Cons "drop n (x#xs) = (case n of 0 => x#xs | Suc(m) => drop m xs)"
  (* Warning: simpset does not contain this definition but separate theorems
     for n=0 / n=Suc k*)
primrec
  take_Nil  "take n [] = []"
  take_Cons "take n (x#xs) = (case n of 0 => [] | Suc(m) => x # take m xs)"
  (* Warning: simpset does not contain this definition but separate theorems
     for n=0 / n=Suc k*)
primrec
  nth_Cons  "(x#xs)!n = (case n of 0 => x | (Suc k) => xs!k)"
  (* Warning: simpset does not contain this definition but separate theorems
     for n=0 / n=Suc k*)
primrec
 "    [][i:=v] = []"
 "(x#xs)[i:=v] = (case i of 0     => v # xs
                          | Suc j => x # xs[j:=v])"
primrec
  "takeWhile P [] = []"
  "takeWhile P (x#xs) = (if P x then x#takeWhile P xs else [])"
primrec
  "dropWhile P [] = []"
  "dropWhile P (x#xs) = (if P x then dropWhile P xs else x#xs)"
primrec
  "zip xs []     = []"
  "zip xs (y#ys) = (case xs of [] => [] | z#zs => (z,y)#zip zs ys)"
  (* Warning: simpset does not contain this definition but separate theorems
     for xs=[] / xs=z#zs *)
primrec
  "[i..0() = []"
  "[i..(Suc j)() = (if i <= j then [i..j() @ [j] else [])"
primrec
  "nodups []     = True"
  "nodups (x#xs) = (x ~: set xs & nodups xs)"
primrec
  "remdups [] = []"
  "remdups (x#xs) = (if x : set xs then remdups xs else x # remdups xs)"
primrec
  replicate_0   "replicate  0      x = []"
```

```
replicate_Suc "replicate (Suc n) x = x # replicate n x"
```

BIBLIOGRAPHY

[Aag92] Aagaard, M. and Leeser, M. A Methodology for Reusable Hardware Proofs. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 177–196, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[ABY85] J.R. Anderson, C.F. Boyle, and G Yost. The geometry tutor. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1–7, Los Angeles, 1985.

[AF96] K. Aspetsberger and K.J. Fuchs. The Austrian projekt. *Int. Journal of Computer Algebra in Mathematics Education*, 3(1), 1996.

[AFHK94] K. Aspetsberger, K.J. Fuchs, H. Heugl, and W. Klinger. The Austrian DERIVE project – final report. Technical report, Austrian federal ministry for education, 1994.

[AFS96] K. Aspetsberger, K. Fuchs, and F. Schweiger. Fundamental ideas and symbolic algebra. In Kent Bromley, editor, *The state of the art*, pages 45–51. Chartwell-Bratt, 1996.

[AG93] A. Armando and E. Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, (8(3-4)):475–502, 1993.

[Age92] A. Tryg Ager. Naturalizing computer algebra for mathematical reasoning. Mathematical Association of America, MAA Notes Series, 1992.

[AL97] Michele Artigue and Jean-Baptiste Lagrange. Pupils learning algebra with DERIVE – a didactic perspective. *Zentralblatt Didaktik der Mathematik*, (4):105–112, 1997.

[Ano94] Anonymous. The qed manifesto. In A. Bundy, editor, *12th International Conference on Automated Deduction*, volume 828 of *Lecture Notes in Computer Science*, pages 238–251, Nancy, France, 1994. Springer-Verlag.

[Asp00] David Aspinall. Proof general: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in LNCS. TACAS, 2000.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Aus00] Andreas Ausserhofer. *Improvements in Web-based Educational Systems for Teaching Computing*. PhD thesis, Institute for Software Technology, TU-Graz, Austria, 2000.

[B+92] Peter M. Bruun et al. *RAISE Tools, Reference Manual*. CRI, 1992.

[BDUS99a] Bürger, Dinauer, Unfried, and Schatzl. *Mathematik 4*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[BDUS99b] Bürger, Dinauer, Unfried, and Schatzl. *Mathematik 8*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[Bee84a] Michael J. Beeson. Foundations of constructive mathematics: metamathematical studies. In *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1984. Volume 3.

[Bee84b] Michael J. Beeson. Mathpert: Computer support for learning algebra, trig, and calculus. In A. Voronkov, editor, *Logic programming and automated reasoning: international conference LPAR '92*, pages 454–456. Springer-Verlag, 1984. Volume 624 of Lecture Notes in Computer Science.

[Bee88] Michael J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60(3):297–340, December 1988.

[Bee95] Michael J. Beeson. Using nonstandard analysis to ensure the correctness of symbolic computations. *IJFCS: International Journal of Foundations of Computer Science*, 6, 1995.

[BGH+97] Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup, and Søren Prehn. A railway system. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. Coordination'97, Case Study Workshop Example.

[BGP95] Dines Bjørner, Chris George, and Søren Prehn. Scheduling and rescheduling of trains. Research Report 52, UNU/IIST, P.O.Box 3058, Macau, December 1995. Published in Industrial-Strength Formal Methods in Practice.

[BJ98] Bruno Buchberger and Tudor Jebelean, editors. *Proceedings of the Second International Theorema Workshop*, RISC-Reports series No. 98-10. 29-30 June, RISC-Hagenberg, Austria, 1998.

[BKSS97] Yves Bertot, Thomas Kleymann-Schreiber, and Dilip Sequeira. Implementing proof by pointing without a structure editor. Technical report, LFCS, 1997.

[BL82] Bruno Buchberger and Rüdiger Loos. Algebraic simplification. In Bruno Buchberger, George E. Collins, Rüdiger Loos, and Rudolf Albrecht, editors, *Computer Algebra. Symbolic and Algebraic Computation*, pages 11–43. Springer Verlag, 1982.

[BL+97] J. Baumert, R. Lehmann, et al. TIMSS Mathematisch-naturwissenschaftlicher Unterricht im internationalen Vergleich. Berlin, 1997.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[BNP98] Marco Benini, Dirk Nowotka, and Carl Pulley. Computer arithmet5ic: Logic, calculation, and rewriting. In *Proceedings of FroCoS'98*, Applied Logic Series. Kluwer Academic Publishers, 1998. to appear.

[Bow93] Jonathan Bowen. Formal methods in safety-critical standards. In *Software Engineering Standards Symposium (SESS'93)*, pages 168–177. IEEE Computer Society Press, 30. August - 3. sep 1993.

[Bow97] David Bowers. Opportunities for the use of CAS in middle secondary mathematics in england and wales. *Zentralblatt Didaktik der Mathematik*, (4):113–117, 1997.

[BT98] Yves Bertot and Laurent Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, February 1998.

[Buc65] Bruno Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal (German)*. PhD thesis, Math. Inst., Univ. of Innsbruck, Austria, 1965.

[Buc84] Bruno Buchberger. Mathematik für Informatiker II (Problemlösestrategien und Algorithmentypen. lecture notes CAMP-Publ.-No. 84-4.0, RISC-Linz, Johannes Kepler University, A-4040 Linz, SS 1984.

[Buc92] Bruno Buchberger. The white box / black box principle. Technical report, RISC-Linz, Johannes Kepler University, A-4040 Linz, July 1992.

[Buc93] Bruno Buchberger. Mathematica: a system for doing mathematics by computer ? Invited Talk DISCO 93, September 1993.

[Buc94] Bruno Buchberger. Thinking speaking writing. Lecture notes, RISC-Linz, Johannes Kepler University, A-4040 Linz, 1994.

[Buc96a] Bruno Buchberger. A Prover for Propositional Logic in Natural (Deduction) Style: Implementation in Mathematica 3.0. Technical Report Theorema-96-9-2, RISC-Linz, Austria, September 1996.

[Buc96b] Bruno Buchberger. Mathematica as a rewrite language. Invited paper at "The Second Fuji International Workshop on-Functional and Logic Programming", Shonan Village, Japan, November 1996.

[Buc97] Bruno Buchberger. Theorema: Natural Language Proofs and Nested Cells Representation of Proofs. In *First International Theorema Workshop, Hagenberg, Austria, June 9–10, 1997.* RISC Report 97-20, 1997.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Acadmic Press, 1983.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, London, Toronto, Sydney, Tokyo, 1988.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CGG+92] B.W. Char, K.O Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V Language Reference Manual.* Springer Verlag, 1992.

[CS93] A. A. Clarke and M. G. G. Smyth. A co-operative computer based on the principles of human co-operation. *International Journal of Man-Machine Studies*, 38(1):3–22, 1993.

[CZ93] Edmund Clarce and Xudong Zhao. Analytica: a theorem prover for Mathematica. *The Mathematica Journal*, 3:56–65, 1993.

[Dav92] J. H. Davenport. The AXIOM system. AXIOM Technical Report TR5/92 (ATR/3) (NP2492), December 1992.

[DFR93] Francoise Darses, Pierre Falzon, and J. M. Robert. Cooperating partners: Investigating natural assistance iv. help and learning. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 303–308, 1993.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[DM94] Babak Dehbonei and Fernando Mejia. Formal methods in the railways signalling industry. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94:Industrial Benefit of Formal Methods*, pages 26–34. Springer-Verlag, October 1994.

[Dör91] Willibald Dörfler. Der Computer als kognitives Werkzeug und kognitives Medium. In *Schriftenreihe Didaktik der Mathematik*, volume 21, pages 51–75. hpt, B.G.Teubner, Wien, Stuttgart, 1991.

[DS96] Andreas Dolzmann and Thomas Sturm. Redlog user manual. Technical report, FMI, Universität Passau, D-94030 Passau, Germany, October 1996.

[FGT90] W.M. Farmer, J.D. Guttman, and F.J. Thayer. IMPS: an interactive mathematical proof system. Technical report, The MITRE Corporation, 1990.

[Fin00a] Thomas Maximilian Fink. Java-notebook. software-requirements-document V 1.4. Institute for Softwaretechnology, TU Graz, Jan.24. 2000.

[Fin00b] Thomas Maximilian Fink. A notebook for calculational proofs. Master's thesis, Institute for Software Technology, TU-Graz, Austria, 2000.

[Fuc98] Karl Josef Fuchs. Computeralgebra – neue Perspektiven im Mathematikunterricht. Habilitation, University of Salzburg, Austria, 1998.

[G+96] J. Goguen et al. Software component search. *Journal of Systems Integration*, 6(1/2):93–134, 1996.

[Geo95] Chris George. A theory of distributing train rescheduling. Research Report 51, UNU/IIST, P.O.Box 3058, Macau, December 1995. Published in: Marie-Claude Gaudel and James Woodcock (eds.), FME'96: Industrial Benefit and Advances in Formal Methods.

[GHHK77] W. Gellert, Kuestner. H, M Hellwich, and H. Kaestner. *The VRN concise encyclopedia of mathematics*. Van Nostrand Reinhold Company, 1977.

[GHJM94] Erich Gamma, Richard Helm, Ralph Johnson, and John M.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gri81] David Gries. *The science of programming*. Texts and monographs in computer science. Springer-Verlag, 1981.

[Gro95] Günther Grogger. Der Einsatz von DERIVE im Mathematikunterricht an AHS. Technical Report ZSE Report No.6, University of Klagenfurt, Zentrum für Schulentwicklung, 1995.

[GS00] David Garlan and Bridget Spitznagel. Toward compositional construction of complex connectors. In *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, November 2000.

[H+93] Anthony C. Hearn et al. *REDUCE 3.5 User's Manual*. Codemist Ltd, Bath, England, November 1993.

[Han94] Kirsten Mark Hansen. Validation of a railway interlocking model. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 582–601. Springer-Verlag, October 1994.

[Har96] John Harrison. A mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[Har97] John R. Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge, Computer Laboratory, November 1997.

[Hea69] A. C. Hearn. Standard LISP. *SIGSAM Bulletin*, 13:28–49, 1969.

[Hey96] Hans Werner Heymann. Mathematikunterricht in der gymnasialen Oberstufe. *MU - Mathematik Unterricht*, 4(5):107–120, 1996.

[HKPM94] G. Huet, G. Kahn, , and C. Paulin-Mohring. *The Coq Proof Assistant*. CNRS-ENS Lyon, 1994.

[HMJR93] M. Hunter, P. Marshall, Monaghan J., and T. Roper. Using a computer algebra system with year 10 students. In B. Jaworski, editor, *Technology in Mathematics Teaching*, pages 281–288. Birmingham University, 1993.

[Hor88] C. Horn. *The Oyster Proof Development System*, 1988.

[HS93] Scott E. Hudson and John T. Stasko. Animation Support in a User Interface Toolkit: Flexible, Robust and Reusable Abstractions. Technical report, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, April 1993.

[Hut89] E Hutchins. Metaphors for interface design. In Neel F. Taylor, M.M. and D.G. Bouwhuis, editors, *The Structure of Multimodal Dialogue*. Elsevier Sci Pub, 1989.

[KB70] Donald E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kir00] C. Kirchner, H.; Ringeissen, editor. *Proceedings of the 3rd International Workshop on Frontiers of Combining Systems, FroCoS'2000*, volume 1794, Nancy (France), March 2000. Spinger-Verlag.

[KJ84] Milos Konopasek and Sundaresan Jayaraman. *The TK!Solver Book. A guide to problem-solving in science, engineering, business and education*. Osborne / McGraw-Hill, 1984.

[KSF93] A. Hadj Kacem, J.-L. Soubie, and J. Frontin. A software architecture for cooperative knowledge based systems ii. software tools. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 2, pages 303–308, 1993.

[Kut97] Bernhard Kutzler. With the TI-92 towards computer age maths teaching. *Int. Journ. of Computer Algebra in Mathematics Education*, 4(1), 1997.

[Len96] Helge Lenne. *Analyse der Mathematikdidaktik in Deutschland*. Ernst Klett Verlag, 1996.

[Lov96] Donald W. Loveland, editor. *Automated Deduction - Some Achievements and Future Directions*, Chicago, April 20-21 1996. Report of a Workshop on the Future Directions of Automated Deduction.

[Maz91] C. Mazza. ESA software engineering standards. Technical Report ESA PPSS-05-0 Issue 2, Board for software standardisation and control, Paris, France, Jan 30 1991.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. Technical Report CS-TR-72-288, Stanford University, Department of Computer Science, May 1972.

[Mil73] Robin Milner. Models of LCF. Technical Report CS-TR-73-332, Stanford University, Department of Computer Science, January 1973.

[Mon97a] John Monaghan. Teaching and learning in a computer algebra environment: Some issues relevant to sixth-form teachers in the 1990s. *Int. Journ. of Computer Algebra in Mathematics Education*, 4(3):207–220, 1997.

[Mon97b] John Monaghan. What are they doing ?! *Int. Journ. of Computer Algebra in Mathematics Education*, 4(3):117–127, 1997.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer Verlag, 1992.

[MST94] J. Monaghan, S. Sun, and D.O. Tall. Construction fo the limit concept with a computer algebra system. In 18^{th} *International Conference on the psychology of Mathematical Education*, 1994.

[MT94] J. Monaghan and D. Tall. Hand-mind interaction: Reflections on processes used in learning mathematics using computers. Preprint available from CSSME, University of Leeds, 1994.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990.

[MW92] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. Technical report, Stanford, 1992.

[Mye98] Brad A. Myers. A brief history of human computer interaction technology. *ACM interactions*, 5(2):44–54, March 1998.

[N+99a] Novak et al. *Mathematik Oberstufe*, volume Band 1. Reniets Verlag, Wien, 1999.

[N+99b] Novak et al. *Mathematik Oberstufe*, volume Band 4. Reniets Verlag, Wien, 1999.

[NB98] Tobias Nipkow and Franz Baader. *Term rewriting and all that*. Cambridge University Press, 1998.

[Neu98] Erich Neuwirth. *Information and Communications Technology in School Mathematics*, chapter Spreadsheets: just smart calculators or a new paradigm for thinking about mathematics structure? In [TJ98], 1998.

[Neu99a] Walther A. Neuper. Mathematics tutoring I: Problem types for mechanized problem solving. technical report IST-TEC-99-07, IICM - Inst. f. Software Technology, Technical University, A-8010 Graz, February 1999.

[Neu99b] Walther A. Neuper. Mathematics tutoring II: A mathematics-engine for guided interaction. technical report IST-TEC-99-15, IICM - Inst. f. Software Technology, Technical University, A-8010 Graz, August 1999.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74, 1993.

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, (10):171–186, 1998.

[Noc96] Robert Nocker. Der Einfluß von Computeralgebrasystemen auf die Unterrichtsmethoden und die Schüleraktivitäten. *Beiträge zum Mathematikunterricht*, 1996.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theoriy. An Introduction.* Oxford University Press, 1990.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification*, pages 411–414. CAV'96, 1996.

[Pap72] S. Papert. Teaching children to be mathematicians versus teaching about mathematics. *Int.J.Educ.Sci.Technol.*, 1972.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Topias Nipkow.

[Pau97a] Lawrence C. Paulson. *The Isabelle reference manual.* University of Cambridge, Computer Laboratory, July 1997.

[Pau97b] Lawrence C. Paulson. *Isabelle's object-logics.* University of Cambridge, Computer Laboratory, July 1997.

[Pfa85] G. E. Pfaff, editor. *User Interface Management Systems: Proceedings of the Seeheim Workshop*, Berlin, 1985. Springer Verlag.

[Pre71] Larry Press. Toward balanced man-machine systems. *International Journal of Man-Machine Studies*, 3(1):61–73, 1971.

[R+92] Hans-Christian Reichel et al. *Lehrbuch der Mathematik*, volume 5-8. Hölder-Pichler-Tempsky, 1992.

[Rec98] Tomas Recio. Didactic relevance of meaningless mathematics. *Int. Journ. of Computer Algebra in Mathematics Education*, 5(1):15–26, 1998.

[RMLH99a] Hans-Christian Reichel, Robert Müller, Josef Laub, and Günther Hanisch. *Lehrbuch der Mathematik, 5. Band für die 5.Klasse AHS*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[RMLH99b] Hans-Christian Reichel, Robert Müller, Josef Laub, and Günther Hanisch. *Lehrbuch der Mathematik, 8. Band für die 8.Klasse AHS*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[Rud92] P. Rudnicki. An Overview of the MIZAR Project. Available at `ftp://menaik.cs.ualberta.ca/pub/Mizar/`, 1992.

[S+94] Eduard Szirucsec et al. *Mathematik*, volume 5-8. Hölder-Pichler-Tempsky, 1994.

[S+98a] Heinz-Christian Schalk et al. *Mathematik für Höhere Technische Lehranstalten*, volume 1. Reniets Verlag, Wien, 1998.

[S+98b] Heinz-Christian Schalk et al. *Mathematik für Höhere Technische Lehranstalten*, volume 4. Reniets Verlag, Wien, 1998.

[S+98c] Heinz-Christian Schalk et al. *Mathematik für Höhere Technische Lehranstalten*, volume 2. Reniets Verlag, Wien, 1998.

[S+98d] Heinz-Christian Schalk et al. *Mathematik für Höhere Technische Lehranstalten*, volume 3. Reniets Verlag, Wien, 1998.

[Sch88] David A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[Sch91] Peter Schüller. *Der Mathematikunterricht an der Höheren Technischen Lehranstalt*. PhD thesis, University of Vienna, Inst.of Mathematics, Juli 1991.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Sho79] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26(2):351–360, 1979.

[Smi91] Douglas R. Smith. Kids: A knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[Sof94] Soft Warehouse, Inc., Honolulu, Hawaii. *Derive Handbuch. Der Mathematik-Assistent für Ihren Personal Computer*, 1994.

[Sta92] John Thomas Stasko. Animating Algorithms with XTANGO. *SIGACT News*, 23(2):67–71, Spring 1992.

[Sta97] Kaye Stacey. Mathematics – what should we tell the children ? *Int. Journ. of Computer Algebra in Mathematics Education*, 4(4):387–390, 1997.

[SUSD99a] Eduard Szirucsek, Hubert Unfried, Herwig Schatzl, and Gerhard Dinauer. *Mathematik 4*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[SUSD99b] Eduard Szirucsek, Hubert Unfried, Herwig Schatzl, and Gerhard Dinauer. *Mathematik 8*. Verlag Hölder Pichler Tempsky, Wien, 1999.

[Sve95] Erich Svecnik. Der Einsatz von DERIVE im Mathematikunterricht an AHS. Technical Report ZSE Report No.12, University of Klagenfurt, Zentrum für Schulentwicklung, 1995.

[TB85] A. Trybulec and H. Blair. Computer Aided Reasoning with Mizar. In *Proc. of 9th IJCAI*. Springer Verlag, 1985. LNCS 193.

[TJ98] D. Tinsley and D. Johnson. *Information and Communications Technology in School Mathematics*. Chapman and Hall, 1998.

[TKW97] U.P. Tietze, M. Klika, and H. Wolpers. *Didaktik des Mathematikunterrichts in der Sekundarstufe II*, volume 1. Vieweg, Braunschweig, Wiesbaden, 1997.

[TMPE86] Enn H. Tyugu, Mikhail B. Matskin, Jaan E. Penjam, and Peep V. Eomois. NUT - an object-oriented language. *Computers and Artificial Intelligence*, 1986.

[VBRLP98] Jeffrey Van Baalen, Peter Robinson, Michael Lowry, and Thomas Pressburger. Explaining synthesized software. In *13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, October 13-16 1998.

[vEB97] P. van Emde Boas. Resistance is futile; formal linguistic observations on design patterns. Technical Report CT-1997-03, The Institute For Logic, Language, and Computation (ILLC), University of Amsterdam, 1997.

[vHIN+89] F. van Hermelen, A. Ireland, S. Negrete, A. Smaill, , and A. Stevens. *The Clam proof planner*, 1989.

[Wei97] Hans-Georg Weigand. 'Mangelhaft' für den deutschen MU? – Eine Stellungnahme zur TIMSS-Studie. *TI-Nachrichten*, (2):23, 1997.

[Wit81] Erich Wittmann. *Grundfragen des Mathematikunterrichts*. Vieweg, Braunschweig, Wiesbaden, 1981.

[Wol96] Stephen Wolfram. *The Mathematica Book*. Wolfram Media and Cambridge University Press, 1996.

[Wur96] Otto Wurnig. From the first use of the computer up to the integration of DERIVE in the teaching of mathematics. *Int. Journal of Computer Algebra in Mathematics Education*, 3(1):11–24, 1996.

[WW91] Hans-Georg Weigand and Thomas Weth. Das Lösen von Abituraufgaben mit Hilfe von DERIVE. *Mathematisch Naturwissenschaftlicher Unterricht*, 44(3):177–182, 1991.

[YS80] D.Y.Y. Yun and R.D. Stoutemyer. Symbolic mathematical computation. In J. Belzer, A.G. Holzman, and A. Kent, editors, *Encyclopedia of computer science and technology*, volume 15, pages 235–310. Marcel Dekker, New York-Basel, 1980.