An Interface for Interactive Proving with the Mathematical Software System Theorema

Felix Kossak

RISC Research Institute for Symbolic Computation
A−4232 Hagenberg, Austria

Felix.Kossak@risc.uni−linz.ac.at

Abstract

The *Theorema* project develops an integrated mathematical software system with special focus on a natural−style language with clear semantics and support for proving. Based on the existing automated proving power, we developed an interface for interactive proving.

We describe the combination of automated and interactive proving in detail, including the changes we had to make to the existing automated system. The only difference in the proof function call is the setting of an additional option 'Interaction' with three possible values for interactive, semi−interactive and fully−automated mode.

Two main windows give a survey over the current proof situation, the initial proof situation, and the past proof steps. Special input windows are created if required. The user has control over the knowledge base and is able to determine each proof step by selecting formulae and available inference rules. Additional input may be given, for instance, in the form of terms for substituting variables.

In addition to the technical details, we demonstrate the use of the interactive proving environment with examples.

Preface

This thesis was written in *Mathematica* 3.0©. This has the advantage that, in addition to its printed form, it is also available as an interactive electronic document (a *Mathematica*© notebook). Readers of the electronic form can go through the given examples (in section 2.6) by simply executing the given commands, so far as they have a respective version of *Theorema* at hand.

On the other hand, *Mathematica* 3.0© does not yet offer all the features of a modern text processing system. Therefore the layout is not in all respects what the reader might be used to from other environments.

Most parts of this thesis describe the features of the interactive proving interface and details of its implementation in textual form. Those readers who are interested in actually using the system are referred to the tutorial examples of section 2.6. However, before starting with those examples unpreparedly, it will be useful to read the following sections: 1.1, Context; 1.2, Motivation; 1.3, Problem; 2.1, Definitions and Basic Notions; and 2.3, Specification. For a proper use of the system, basic knowledge about *Theorema* is assumed, although this is not required for going through the given examples only.

Acknowledgements

I would not have been able to perform this work without the help of others, neither in general nor in detail. I want to thank especially my supervisors who determined the subject of this thesis, namely Bruno Buchberger, the leader of the *Theorema* project, and Franz Lichtenberger, who wants to use the implemented system for the mathematical education of Software Engineering students.

Many members of the *Theorema* group have helped me to understand the internals of proving in Theorema and to solve technical problems with *Mathematica*©; I want to mention in particular Daniela Vasaru−Dupre, Franz Kriftner, Tudor Jebelean and Wolfgang Windsteiger (in alphabetical order), whom I very often had to bother with questions. The style file for this work was provided by Martin Kirchner, which saved me a lot of time.

1. Introduction

1.1 Context

Theorema is a mathematical software system which is being developed at the RISC Research Institute for Symbolic Computation in Hagenberg, Austria, under the leadership of Bruno Buchberger.

Theorema is designed as a comprehensive working environment for mathematicians and mathematically−oriented scientists, supporting all key aspects of numerical and symbolic computation as well as the presentation of the work performed within this environment.

According to Bruno Buchberger, the ’Creativity Sprial’ of Mathematics comprises four key steps:

- Inventing,
- Proving,
- Programming, and
- Computing.

Mathematical software systems which are currently available on the market, such as *Mathematica*© or Maple©, can provide good support for the programming and computing phases, but virtually no support for proving. On the other hand, there is tremendous progress in the field of automated proving, but there is no integration into an intregrated environment so far. The ’inventing’ phase can be considered to be essentially ’human’, but the better the support for the other phases, the more likely is ’invention’.

Theorema tries to close the gap by adding automated proving to *Mathematica*©. Moreover, it introduces an even more powerful user language with a considerably more ’natural’ notation close to the handwriting of mathematicians, offering alternative notations and even allowing the personal extension of the notation [Buchberger, 1998].

Together with *Mathematica*©, Theorema provides an ideal environment not only for the development of mathematical papers, but also for publishing the results using the same documents.

1.2 Motivation

In the present version, Theorema offers provers for fully automated theorem proving. However, interaction with the user to increase the proving power as well as for the use of the system in mathematical education was an essential design feature of Theorema from the outset.

Apart from Bruno Buchberger's motivation, the Department of Software Engineering of the Polytechnical University of Upper Austria came up with a requirement for software support for formal specification and proof training within the first two terms in the course of Algorithmic Mathematics for software engineers.

This proposal was brought forward by Franz Lichtenberger, who is in charge of the mathematical education in the department. It includes the support of programme verification, the prerequisites for which were developped by Martin Kirchner [Kirchner, 1999].

Franz Lichtenberger argues that the trade of software engineering requires a different mathematics than other fields of engineering, namely in particular discrete mathematics rather than analysis. More specifically, formal methods are becoming more and more important in the field of software development, including formal specification and verification. If these methods are taught accordingly, there will be a more widespread use, leading to enhanced quality and efficiency [Lichtenberger, 1998a] (see also [Lucas, 1999] and section 1.4, History and State of the Art, p 3).

The system is also proposed to be used in introductory courses of the Institutes of Mathematics and Computer Science at the Kepler University in Linz. For the contents of the respective courses at the polytechnic and at the university, see [Lichtenberger, 1998] and [Buchberger and Lichtenberger, 1981].

Interactive proving is not something new. Peter Hirt implemented an 'Assistant for Interactive Proof Construction' (AIPC) at the Kepler University in Linz in 1995 [Hirt, 1995]. Other well–known examples are LCF, HOL, or Isabelle (for an overview, see [Harrison, 1999] or [Nipkow and Reif, 1998]; [Hirt, 1995] also gives a description of two other systems). Within *Theorema*, Tudor Jebelean implemented a prototype in 1998 which allows for stepwise proving and the removal of assumptions from the knowledge base at any time during the proof. This prototype also allowed the user to insert a new subgoal (as a kind of 'lemma' as explained in section 2.3, 'Specification') [Buchberger, Jebelean, and Vasaru, 1998].

However, the challenge was to integrate a facility for interactive proving into an existing, powerful environment for automated proving and to work further towards the combination of computer power for routine parts and human power for the creative parts of proving.

1.3 Problem

Bruno Buchberger outlines a first specification for interactive proving in an internal report [Buchberger, 1999]:

Ways of interaction are:

- providing solving terms for the proof of existentially quantified formulae,
- proposing suitable instances of universally quantified formulae in the knowledge base,
- proposing inference rules to be applied or a direction of how the proof should proceed in a certain proof situation.

For this to be possible, we must provide means such that

- a prover can stop at certain critical proof situations, propose a way to continue and ask for confirmation by the user that it should continue in this way,
- a prover can stop at certain critical proof situations and ask for advice about the way to continue,
- the user can set some stopping points before the prover starts to work,
- the user can cause the prover to back up in a proof and start again, in a different direction, in an earlier stage of the proof without having to start again at the very beginning.

The first part of this work was to make experiments with an existing prover in order to establish the detailed requirements for interaction in this sense. The system has to serve the needs of the mathematician who does not get on with the proving power of the automated provers at a certain point as well as to support proof training for students.

1.4 History and State of the Art

In the past, many looked at automated proving as sort of unserious 'playing', as the provers continued to fail with most interesting proof problems for a long time. Nowadays, many experts believe that computer support for deduction – or proving – is becoming indispensable.

The reason lies in the new challenges for engineers with respect to complexity and quality which requires them to increasingly adopt the logical process of formal deduction in their process of development. This is evident in the field of software engineering, where security–critical systems already have to undergo verification, but it also holds true for other areas [Bibel and Schmitt, 1998].

The problem of past and current automated proving systems does not so much lie in the problem of incompleteness. It was proved by Church and Turing in 1936 that there is no general algorithm which could check the validity of every proposition in first−order logic in finite time (this problem is 'semi−decidable'). But we have complete algorithms for proving theorems if they are indeed valid (else they may not stop) [Chang and Lee, 1973].

Many deduction systems are complete for first−order logic in this sense. But the real problem is to find efficient algorithms and realise efficient systems that can solve non−trivial problems on standard computers within an acceptable period of time [Bibel and Schmitt, 1998].

The development of machine support for mathematical proving started as early as 1960 when Gilmore implemented a mechanical method for proving theorems proposed by Herbrand in 1930 [Chang and Lee, 1973]. The first 'mechanical provers' were so−called black−box provers which simply state whether the theorem in question could be proved or not. The best−known provers of this kind are based on J. A. Robinson's *resolution principle* published in 1965 [Chang and Lee, 1973].

For a long time, automated proving concentrated on black−box proving using either saturation calculi (like resolution) or tableau− and connection−calculi, and most research is still centred upon this approach [Bibel and Schmitt, 1998]. With the *Theorema* system, Bruno Buchberger introduced another approach which enables 'natural−style' language output to represent proofs obtained by a 'natural deduction' calculus developed from G. Gentzen's calculus (see eg [Buchberger, 1998]).

This natural deduction calculus proves also very suitable for interactive proving (see eg [Hirt, 1995]), therefore automated provers employing the Gentzen calculus are especially suitable for being combined with interactive proving. Other systems use sequent calculi instead [Nipkow and Reif, 1998].

A short history of interactive theorem proving is given in [Nipkow and Reif, 1998]. Nipkow and Reif highlight the great influence of the Stanford LCF on most present systems. John Harrison lists well−known interactive systems like HOL or Nuprl in the 'LCF family' [Harrison, 1999].

The reason lies in LCF's generic architecture which is based merely on functional programming and data abstraction. Theorems are modelled as abstract types, and the constructors for theorems correspond exactly to the inference rules of the logic used.

LCF−type systems usually support both forward proving (by constructing the theorem from the given theory, ie the knowledge base) and backward proving by so−called *tactics*. A tactic is basically a function which transforms the proof goal to (simpler) sub−goals.

Both forward inference rules and tactics are user−definable [Nipkow and Reif, 1998]. However, the user is required to programme these rules or tactics; for instance, in Isabelle, the user is supposed to be familiar with the programming language ML [Paulson, 1995].

A very important field of application is programme verification. The present state of the art is that given enough time and manpower, any program or hardware design can be verified by an interactive theorem prover [Bibel and Schmitt, 1998]. Therefore also in interactive proving, the focus of research is on efficiency, but also on flexibility.

Most systems support quite specific logics, because more customised systems tend to be more effective. However, in generic systems like Isabelle, one can present one's own logical system in a meta−logic. Moreover, it should be possible to integrate different reasoning styles such as top−down, bottom−up, or middle−out [Nipkow and Reif, 1998]. (For a thorough description of Isabelle, see [Paulson, 1995].)

In general, interactive theorem provers support structured theory formation. However, it is also expected that they make use of the given structure in the proofs. Many interactive provers give the users access to different kinds of proof engineering capabilities such as

- organisational support for keeping track of proof obligations and the knowledge base,
- proof manipulation support for inspecting proofs, withdrawing proof steps, etc and
- library support for reusing frequently required theories and lemmas.

Some provers offer more advanced proof engineering techniques, like proof reuse and −adaption, proof generalisation, proof transformation, learning from failure, or proof by analogy. Many systems treat proofs as concrete data objects which can be displayed, manipulated and recorded [Nipkow and Reif, 1998].

Interactive proof tools are expected to automatically fill in routine steps − like simplifying goals − and to discharge automatically as many proof obligations as possible. Therefore various groups are now working to combine fully automated and interactive modes into one system. Automation may also be attained by appropriate proof strategies within the interactive system or by other external reasoners such as model checkers or computer algebra systems [Bibel and Schmitt, 1998][Nipkow and Reif, 1998].

For instance, in the example of programme verification, a combination of automated proving and interactive proving is in particular aimed at the ability to handle hundreds of axioms, most of which use equality. Whereas equality is not handled very well by most automated provers, an interactive user will have problems with such large theories.

Such an integrated deduction system incorporates the proof engineering capabilities of an interactive system and, at the same time, eliminates user interaction for those goals that can be solved by the efficient combinatorial proof search embodied in an automated prover. Ahrendt et al. have implemented a prototype of such a system [Ahrendt et al, 1998].

According to Nipkow and Reif, very few interactive theorem provers can be considered to be completely sound. For reasons of efficiency, they take shortcuts – decision procedures which are not based on formal proofs [Nipkow and Reif, 1998].

1.5 Solution

Analysing the major problems we encountered with fully automated proving lead to a specification for an interactive interface which should support three basic modes of proving:

- full automation (no interaction at all),
- automation with interaction at certain points, and
- fully interactive, stepwise proving.

The basic feature of stepwise proving is that the user is able to determine every proof step by applying a particular inference rule on a particular formula. Some rules will require additional input such as a second formula or a term to substitute a quantified variable. The latter makes it possible to cope with the two proof situations which are probably those that require the most creativity: to find a suitable examplary term to prove an existentially quantified formula and to find a useful term for using a universally quantified formula from the knowledge base.

Another important feature enables the user to control the knowledge base during a proof. Assumptions can be cleared from the knowledge base and new subgoals can be added. Once a subgoal has been proved, it is automatically added to the knowledge base.

Other features include the ability to let the automated prover which runs in the background do a certain step or finish the proof from a certain moment on.

In our implementation, we concentrated on stepwise proving and those features which are required for the system's use in proof training for students.

We approached a solution from two sides: we had to design a suitable internal control structure and develop a user interface with the respective functionality.

The original control structure designed for automated proof construction had to be broken up to make the process dependent on the current value of the new prover option 'Interaction'. In case of fully interactive or semi–automated proving, the visual proof output must also be created step by step rather than in one step at the end of the process. This forced us to re–locate the handling of the 'proof window' where the 'pretty–print' output is displayed.

To enable user input, we suspend the proof process and create a child process which ends when the input for the next step is complete.

The two major windows which are visible all the time in the interactive mode are the ’proof window’, where the proof gradually evolves, and the ’interactive proof situation’, which is the command center for user interaction. The latter depicts the current proof goal and the assumptions currently available in the knowledge base.

Selecting one of these formulae is the starting point for constructing the next proof step. It causes a new window to open where an inference rule to be applied on the selected formula can be chosen. Other custom windows pop up when additional input is required. Several options are available in the interactive proof situation window and in the rule selection window.

Unfortunately, it was not possible to use an existing automated prover as the basis for interactive proving without making considerable supplements. The necessary steps – which will have to be repeated for other provers to be fit for interaction – are described in detail in the section of ’Implementation’.

2. Developing the Interactive Interface

2.1 Definitions and Basic Notions

2.1.1 Notions Commonly Used in Theorema

In *Theorema*, there are two basic kinds of provers: the basic provers and the user provers. Each *basic prover* implements a number of *inference rules*, rules which determine a certain step in a proof under certain conditions, ie in a certain proof situation.

The user can only work with the *user provers*. These will invoke one or more basic provers, but they also care for the required initialisations and some transformations of the given formulae and then call up the mechanism which controls the proof process and handles the output.

In the context of this thesis, the notion of a prover will refer to a basic prover if not otherwise specified.

Most basic provers provided by *Theorema* apply the *Natural Deduction* technique as introduced by Gerald Gentzen. The point is that the inference rules of natural deduction are 'close to the intuitive notion of reasoning' [Hirt, 1995].

Theorema provers basically follow the style as it is taught by Bruno Buchberger, eg in [Buchberger, 1991]. An example for an *inference rule* in that style would be as follows:

In order to prove 'A⇒B', it suffices to prove 'B' under the (additional) assumption 'A'.

Given a proof situation with a goal of the form 'A⇒B', a typical text in the generated proof will read like this:

We assume

(1) A

and show

(2) B.

As *Theorema* generally supports Higher Order Predicate Logic, also our interactive system supports it. The practical effect is that one can use function variables and predicate variables. In general, Theorema supports classical logic, where one can assume that a proposition A is true if (¬A) is false.

The proposition to be proved under the given assumptions will be called the *proof goal* (or main goal). The goal of a node other than the root can be different from the main goal; we will call such a goal a *subgoal*. For instance, when we apply the deduction rule on the main goal 'A⇒B', the subgoal in the succeeding AND node will be 'B' (see above).

The assumptions explicitly given in a proof call make up the explicit knowledge base for the proof. When we use the notion of *knowledge base* in this work, we will refer to these explicitly given assumptions. However, the actual knowledge base is far greater: a great part of it lies hard−coded in the inference rules implemented in the basic prover or provers used. For a more detailed discussion on the relation between (explicit) knowledge bases and inference rules see [Buchberger, 1999].

The example text from a prover output given above represents a *proof step* ('We assume ... and show ...'). A proof step is one application of one inference rule to one (or two) formulae. The result of a proof step is a new *proof situation*.

A *proof situation* consists of the current goal (the main goal or a subgoal) and the available knowledge base in a certain point of the proof. The knowledge base will usually contain all the assumptions given by the proof call, but also further assumptions derived during the previous part of the proof process. Consequently, the initial proof situation only comprises the main goal and the initial assumptions which were explicitly given.

In *Theorema*, a proof is represented by a *proof tree* (or *deduction tree*). A given proof situation is reduced to several new, 'simpler' proof situations. If all resulting proof situations are proved, then also the initial situation is proved [Tomuta et al, 1998].

One such proof situation forms a node of the proof tree. Possible nodes are AND nodes, OR nodes, IF nodes and a terminal node. A *branch* of the proof tree consists of a node and all its succeeding nodes, ie all nodes 'below' this node.

A terminal node represents a proof situation with either of the *proof values* 'proved', 'disproved' or 'failed', where 'disproved' means that a contradiction was arrived at. Non−terminal nodes have the proof value 'pending'. At an AND node, all the successive branches have to be proved in order to obtain the proof value 'proved' for the whole branch. Most nodes of a typical proof tree are AND nodes.

An OR node represents proof alternatives, corresponding to a disjunction. The proof of one branch suffices to obtain the proof value 'proved'. IF nodes were introduced mainly in order to handle case distinctions or definitions of partial operators [Tomuta et al, 1998].

A *proof object* is a formal representation of a (partial) proof with a special data structure. It consists of the initial proof situation, a proof history (which allows to reconstruct the proof starting with the initial proof situation) and a proof value. If the proof value is 'pending', it also contains the current proof situation with which the proof should be continued.

2.1.2 Special Notions for Interaction

In proving, we encounter many routine situations which can be handled in an automised way. The inference rules implemented by *Theorema* provers tackle such situations. However, some situations require creativity, notably proof situations where the goal is an existentially quantified formula and situations where one needs to transform a universally quantified formula from the knowledge base. In both cases, the difficult task is to find a suitable term for substituting the quantified variables [Buchberger and Lichtenberger, 1981][Buchberger, 1991]. On a more abstract level, the development of a proof idea must be considered as the most important creative act.

Creative acts must be considered a task for humans. However, the machine can support the human by taking off the routine parts of the work so that the mind is set free for ideas (see [Buchberger, 1998]).

For 'semi−interactive' proving, we have defined the notion of a *critical point*. A critical point is a proof situation that calls for a creative act. It is a proof situation where the prover should stop and ask for user input.

The user interface for interactive proving which is introduced in this thesis uses different windows. A window corresponds to a *Mathematica*© notebook. The two most important windows are the *proof window* and the *interactive proof situation* window.

The *proof window* can be the window (notebook) from which the proof was invoked (the proof call window). However, by default, a new window will be created for this. In the proof window, a pretty−print trace of the proof will be created. In fully automated proving, this will be done in one step as an output of the 'Proof' function (in addition to a proof object which can be used by the user for further steps). In interactive proving, the contents of this notebook are created step by step.

The *interactive proof situation* window depicts the current proof situation and some options. The user can select a formula from the proof situation for the next proof step.

Selecting a formula in the proof situation window will open another window, urging the user to select an induction rule (or an option). Other windows pop up when more user input is required, depending on the selected rule.

2.2 Analysis

The most important requirements for an interactive proving environment were laid down by Bruno Buchberger in an internal report of the *Theorema* project [Buchberger, 1999]. Some features were adopted from the AIPC Assistant for Interactive Proof Construction by Peter Hirt [Hirt, 1995]. Other ideas were contributed by experienced mathematicians, in particular members of the *Theorema* team.

The first part of this work was to conduct experiments with the existing Predicate Logic Prover of Theorema in order to establish the detailed requirements for interaction from the user's point of view. Thereby we had to have in mind the needs of the user who does not get on with the proving power of the automated provers at a certain point as well as didactic aspects for proof training.

For the selection of proof examples, we partially followed Bruno Buchberger's method of the Complete Exploration Cycle for building up theories [Buchberger, 1999; Buchberger, 1999a]. In addition, we used standard examples from the field of analysis.

2.2.1 Lessons from Experiments

One of the most important things for doing long proofs by hand is to have, at any time, a handy list of the proof goal or the current subgoal and all the formulae in the knowledge base that is currently available. Such a list will be called the proof situation. It is the basis of proceeding with the proof by choosing a formula to use it in the next step.

The first approach to add interaction to automated theorem proving is to have control over the knowledge base. *Theorema* already allows for complete control over the knowledge base for each call of a prover or evaluator. Tudor Jebelean went one step further in his interactive proving prototype by enabling the user to remove formulae from the knowledge base at each proof step, as well as enabling the insertion of a new subgoal.

Automised provers often find it difficult to select the appropriate formula for the next proof step. Experiments with the Predicate Logic Prover show that the prover often iterates with one rule over one formula of the knowledge base until the 'search depth' (iteration limit) is reached. Consequently, reducing the knowledge base can not only make proofs shorter but often leads to a success where the prover otherwise fails.

However, deleting a formula always affects a whole branch of the proof tree, although we could need it in a later step. This leads to the idea that one can explicitly select a formula to be used in the next step.

It often turns out that more than one rule is applicable to a particular formula in the knowlege base. Moreover, when training students in proving techniques, the student should

be urged to think about the next step in detail. Thus it should be possible to explicitly choose the inference rule to be applied on the selected formula.

A prover can implement a great number of inference rules which can lead to a long and hardly surveyable list to select from, whereas a researcher will only want to see the rules which are, in principle, applicable to the chosen formula. On the other hand, it might enhance the training effect if the student has to select from all the rules. So the applicability of a rule should be checked either before listing (such that only applicable rules are listed) or after the selection of the rule by the user, depending on the purpose.

Often, in a way similar to modular programming, one will prove a lemma that can then be assumed (ie added to the knowledge base) for proving the main goal. In automated, but interactive proving, this can be achieved by introducing a new subgoal during the proof. This subgoal will be proved before going on with the main goal or the previous subgoal, respectively.

When a proof attempt is about to fail, one might not want to start again from the beginning because up to a certain point the attempt might look promising. It would be convenient to just go back several steps and try again from there, and by this get rid of obviously void steps and newly created, redundant formulae.

The experience gathered in the way described above finally led to the following specification.

2.3 Specification

2.3.1 Modes of Interaction

When automated provers are employed, there are three possible modes of interaction which the system should support:

- no interaction at all,
- interaction at critical points, and
- fully interactive, stepwise proving.

When the user chooses to have no interaction, a call of the ’Proof’ function will simpy yield a ’pretty−print’ output with the proof result and a trace in natural deduction style.

In full interaction, on the other hand, one has to determine each proof step oneself. At each step, the user can choose an inference rule and the formula or formulae on which the rule should be applied.

Both approaches can be combined by letting the prover run automatically until a certain point and then stop and ask for input. Such a point in the proof process will be called a critical point. This is, in particular, a point where the prover does not know how to go on, ie

it would fail without guidance or additional knowledge. However, also the user should be able to define critical points within the proof call. For instance, the user could tell the prover to stop whenever a suitable instance for proving an existentially quantified formula is to be constructed.

The following details of the specification refer in particular to stepwise proving, although they will also be applicable to semi−automatic proving (when stopping at critical points).

2.3.2 Controlling the Knowledge Base

At each proof step, the user should be able to remove an assumption from the knowledge base. (Following a discussion within the *Theorema* team, we decided that it should not be possible to add an assumption because this could lead to a misleading reading of the proof output, which clearly states at the beginning under which assumptions the proof was obtained. This issue will have to be discussed further in the future.)

Alternatively, it should be possible to insert a new subgoal. After the subgoal is proved, it should be added to the knowledge base. Removing or adding formulae will be effective for all the successive branches of the current node of the proof tree.

2.3.3 How to Determine Proof Steps

The user must be able to select a formula − either the current goal or an assumption. Then the user should be shown a list of available inference rules which he or she can select.

For students to be trained in mathematical proving, a list of all the rules provided by the currently used prover should be shown. In this case, it is essential that the applicability of the rule to the selected formula is checked; if the rule is not applicable, the user must be notified.

For the comfort of other users, it should be possible to show only the rules which are applicable to the chosen formula.

Some rules require more input. For substituting an existentially quantified variable in the goal or a universally quantified variable in an assumption, a suitable term must be found. The user must be urged to input such a term.

There are rules which require more than just one formula. Modus Ponens, for instance, needs a formula of the form 'A⇒B' as well as another formula of the form 'A'. The student, after selecting the formula with the implication and the rule 'Modus Ponens', should be urged to choose an appropriate second formula from the knowledge base.

2.3.4 Other Facilities

The user should be able to go back in the proof tree, ie start again at an arbitrary node of the proof tree, and try all the successive branches again. This means that the proof situation must be recoverable at any point of the tree.

Other features for the user's comfort should enable her or him to

- insert text in the proof,
- let the prover do the next step automatically,
- let the prover finish the proof automatically, and
- abort the proof.

2.4 Solution

2.4.1 Design

We commenced the design from two sides, more or less independently: on the one hand, enabling interaction in the internal control structure for proving, and, on the other hand, designing the user interface. Figure 1 in section 2.4.3 below gives the result of the coarse design of the new control structure.

As for one part of the user interface, the proof window was retained in principle. The difference is that it should be able to build up the contents step by step during the proof, whereas normally it is only created and shown at the end of the proof with its contents built up more or less in one step.

The idea for the interactive interface was to start with a representation of the proof situation where the user is able to select a formula which is to play the key role in the next step. The proof situation was designed as a separate window.

For the selected formula, the user should be able to choose a particular inference rule. For this purpose, another window should pop up. If the selected rule requires additional information, yet another window (on a third layer) should be created to enable further input.

Later, options were added to both the interactive proof situation window and the rule selection window. The design allows the programmer (maintainer) to add more options at any time.

Only the proof window and the current proof situation should be visible all the time, whereas the other windows should only be created when needed and closed immediately after the input is completed. The proof situation window will be destroyed after each step and re−created with the new proof situation. (Experience shows that, in *Mathematica*©, this does

not take conceivably longer than the rather complicated process of deleting and inserting cells in an existing notebook if the number of cells is not so great.)

Due to the limited possibilities that *Mathematica*© offers to configure the user interface, selection was designed to be enabled by buttons. More specifically, there should be a button for each formula of the proof situation and for each available rule, as well as for each option. The button of a formula should bear its label rather than the whole formula, but the formula itself should be depicted beside it.

For giving mathematical expressions (eg terms) or text as input, we first employed a simple device offered by *Mathematica*© , called 'Input'. This is a small window (also known as 'dialog' in other environments) with an entry field and an 'OK' button. The programmer can add explanatory text (to prompt the user for a particular input).

However, tests after implementation showed that the Input window only accepts pure ASCII input. All *Mathematica*© expressions – and therefore also all *Theorema* expressions – can be written down using ASCII characters only, but this means that the user has to know the correct 'Input Form'. Therefore we re–implemented this sort of input using an ordinary window from which the contents of a special input cell are read after an 'OK' button was pressed.

2.4.2 New Prover Options

The syntax of the 'Proof' function call allows for setting options which the individual provers may handle or ignore. Some options are already handled by common functions. The current values of these options can be obtained by calling a respective 'Get' function which is globally available.

This enabled us to simply introduce a new option name of 'Interaction'. Supported values are "stepwise", "critical-points" and "none", which is the default. The intended semantics is that in the case of "stepwise", the user should be able to determine each proof step, whereas with "critical-points", the prover should stop when the proof situation matches the pattern of a critical point to ask for user input, and with "none", the proof should be performed fully automatically.

2.4.3 A New Control Structure

The goal of this thesis was to enable interaction for proving with *Theorema* using the existing provers and proof infrastructure, including data structures and proof control structures and functions.

The first step towards interaction was a basic change in the control structure for proving. This control structure is basically divided into two functions within two different packages. The functions are called 'ProofSearchAndShow' and 'SearchProof'.

’ProofSearchAndShow’ basically handles the interaction between the actual proving process and ’pretty−print’ output. At the end of a proof, it creates the proof window notebook (if a new one was to be created) and displays it.

For interactive proving, the proof window must be created in the very beginning. Its contents must be added to after each proof step. However, ’ProofSearchAndShow’ calls the ’SearchProof’ function only once. We therefore decided to move the handling of the proof window to ’SearchProof’ in case the Interaction option is set to "stepwise" or "critical-points", so that we did not have to change the overall structure of ’ProofSearchAndShow’ (as the structure of ’SearchProof’ had to be changed anyway).

In the function ’SearchProof’, a radical change of the control structure was required. The original structure was, to put it in a simplified way, a ’while’ loop which terminates when a final proof step is reached or when the current proof situation cannot be dealt with by the prover. Within this loop, the prover is called and the proof tree is constructed step by step.

The new structure of ’SearchProof’ is sketched in figure 1. The while loop is retained, but there are major changes within. The flowchart shows that first it is checked whether the interaction mode is set to "stepwise" (full interaction) or not. If not, it must be checked whether the basic prover ’needs help’. In the ’No’ branch, the original code is executed.

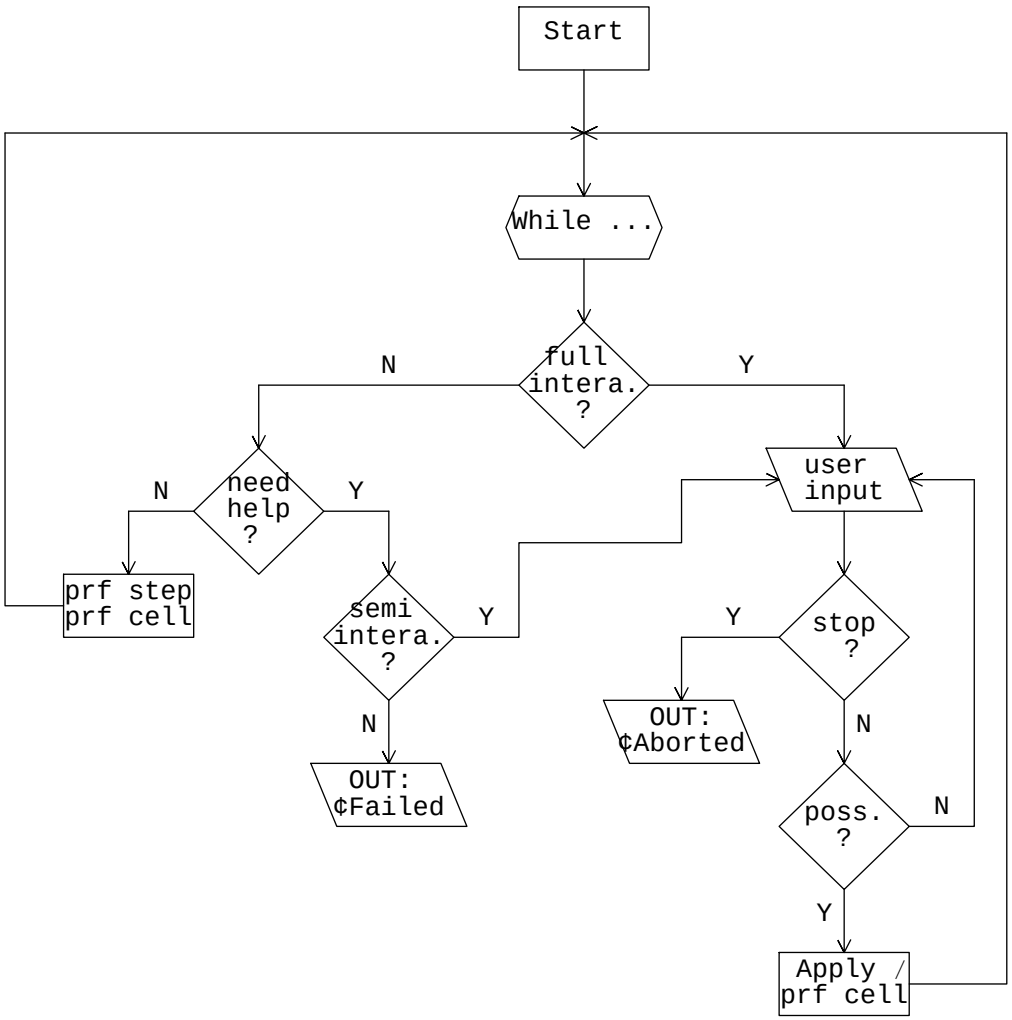

Figure 1 : A simplified flowchart for the modified function ' SearchProof'

What are the criteria for the prover to 'ask for help'? Although it has been taken into account in the new control structure, this has not been implemented yet. However, the idea is that the prover 'needs help' if

- it cannot transform the proof situation (ie there is no rule for this situation) or
- the user has declared that it should ask for input at certain proof situations such as a goal that is an existentially quantified formula or a particular assumption that is a universally quantified formula.

In case of the 'Interaction' option set to "stepwise", a new function 'InteractiveProofSituation' is called. In this function, the proof process is suspended (speaking from the software engineering point of view) to allow user input. This function could also be used when "critical-points" is implemented.

After the function returns, the function 'OneProofStep' is invoked. This is the same function as existed already before, although we had to modify it slightly. Next, the proof notebook is

updated. (In case of no interaction, all output into the proof notebook is still performed in the 'ProofSearchAndShow' function, which is logically one level higher – or, to put it another way, it is on a lower level of the stack.)

There was only one change that had to be made in the function 'OneProofStep'. In the case of stepwise proving with user interaction, additional information has to be handed over to the basic prover, namely the user input, consisting of the selected formula, the selected rule, and additional input where required. However, if the interactive user chooses to have the next step done automatically by the prover, the prover call is done as in fully automated proving.

2.4.4 The User Interface

The new function 'InteractiveProofSituation' creates a new notebook (which is displayed as a new window) and suspends the process in order to allow user input.

The notebook displays the proof goal or subgoal and the assumptions currently available. Both the current goal and the current knowledge base are passed as arguments to the function. The label of each formula is displayed in the form of a button, with the formula itself written beside it. This design makes the whole formula better readable and gives a better survey. The label on the button also reflects the philosophy to identify a formula by its label rather than by the actual expression. However, in *Theorema*, the user has to make sure that this label is unique.

But the first thing besides depicting the current proof situation was to enable stepwise proving in principle. To this end, we started with the realisation of the option 'Perform next step automatically' which was added to the proof situation window (along with other options).

Apart from going through a proof step by step, interaction by fully determining each proof step (within the power of the prover used) required the following steps:

- All the required information had to be prompted from the user and consolidated.
- It had to be made possible to explicitly use a particular rule implemented by the basic prover.
- The available inference rules had to be listed for the user to select.
- The applicability of the selected rule in the given situation, respectively with the given additional input, had to be checked.
- The rules implemented in the basic prover had to be adapted to take extra information (in the form of arguments) in addition to the proof situation.

For each category of input, an extra window was designed. The most important of these is the window which lists all the available inference rules. (In a later version, it should also be

possible to list only those rules which are, in principle, applicable to the selected formula.) Each rule is again designed as a button for choosing.

An alternative button deletes the selected formula from the knowledge base. Although it is listed under the header 'Options' (as on the logical level it cannot be seen as an inference rule), it is actually implemented as an extra rule of the basic prover. The option 'Abort' simply closes the window, enabling the user to select another formula.

For selecting a second formula from the knowledge base as it is required for several rules, an extra window lists the assumptions in the same way as it is done in the Interactive Proof Situation window. For giving a term to substitue a variable or for inserting text in the proof window (an option in the Proof Situation window), a window with a prompt and an 'OK' button is created. The content of the cell designed for user input is read into a global variable after the 'OK' button was pressed.

Inserting a new subgoal requires the input of a formula and a label. Also this input is handled with an input window as described above.

A critical point is the explicit selection of an inference rule on the level of the basic prover. So far, the standard structure of those rules does not provide for such an explicit use. The same holds true for the possibility to give additional arguments, namely the selected formula as well as a second formula or a substitution term.

A final solution to this problem is still to be discussed within the *Theorema* team. The solution also influences the listing of these rules. Our provisional solution is discussed in the section on implementation below.

In our design, we considered two different points where the selected rule should be checked for applicability to the selected formula. It certainly must be checked before actual application. By default, a *Theorema* prover exits with the proof value 'Failed' when no appropriate rule of a basic prover could be found – which is, on the programming level, exactly what happens if the selected formula does not match the selected rule. This programme termination must be prevented.

What should happen is that an error message is given to the user, and the unchanged proof situation enables the user to choose a different step.

For professional users, where the educational point of view is not relevant, the applicability of the selected rule should already be checked at an earlier stage: the rule selection window should only list those rules which apply to the selected formula. However, at least when additional input is given after selecting the rule, the applicability must be checked again. Therefore, and because we stressed the educational functions, we first implemented the check *after* the rule selection and the gathering of further input.

2.5 Implementation

2.5.1 General Problems

Although interactive proving was planned to be included in *Theorema* from the outset, previous work had concentrated on fully automated proving. Consequently, adaptions had to be made on several different points, in different modules (*Mathematica*© packages): in control structures for the proof process and output handling as well as in the used basic prover.

Apart from the design and implementation of the user interface, this required us to thoroughly study the design of the general proof mechanism as well as several implementation details, but also to deal with the implemented inference rules and the underlying logic.

Concerning the key feature of the programming aspect – the suspension of the running proof process to enable user input – we had to cope with a very relevant bug in the Unix and Linux versions of *Mathematica* 3.0©.

Mathematica offers two possibilities for enabling user input – the essence of interaction – within a running process. One respective function is called 'Input'. It opens a small notebook, corresponding to what is called a 'dialog' in other environments, with one entry field and an 'OK' button. The only parameter that this funtion allows is a string which is displayed above the entry field. (As there are no types in common *Mathematica*© programming, any expression will be accepted as input.)

However, the limits of this function were absolutely unsatisfactory for our purposes, as it does not allow for selecting items and the input is restricted to pure ASCII text. On the other hand, the function 'Dialog' offers unlimited scope for interaction. It suspends the running process and opens a new one, theoretically with all the possibilities of a normal *Mathematica*© Kernel process. After this child process is terminated using the function 'Return', the initial process continues just after the code line where 'Dialog' was called.

Unfortunately, our first trials with this function failed to really terminate the child process. Later we were confirmed by an experienced *Mathematica*© programmer that this is a bug in the Unix and Linux versions.

For this reason, we built a first rudimentary prototype with the beta version of *Mathematica* 4.0©. To our astonishment it turned out that this prototype was also running under *Mathematica* 3.0©. Our assumption is that 'Dialog / Return' works if the child process is 'sufficiently complex'. Hence we continued the development under version 3.0.

Mathematica is not a fully capable environment for developing a user interface – it offers some, but not all possibilities that the user interface programmer is used to. For instance, neither is there a provision for creating a context–sensitive menu, nor is it possible to edit the menu bar of the window. This limited the ergonomics of the user interface already at the design level.

2.5.2 Stepwise Proving

The control structure of *Theorema*'s function 'SearchProof', which controls the flow of the proof steps, was altered by the author and enables now three different modes for going through the proof process, according to the possible values of the new prover option 'Interaction': "none", "critical-points", and "stepwise".

When the option is set to "none", which is the default, proofs work exactly as before the changes. Not a single test has shown a different behaviour since the control structure was modified. The option value "critical-points" is provided for in the control structure, but it has not been further implemented so far.

The option value "stepwise" triggers the creation of the proof window before the first proof step is done. This is implemented in the function 'SearchProof' instead of 'SearchProofAndShow', where this happens in the case of no interaction.

After gathering user input by calling the new function 'InteractiveProofSituation', the existing function 'OneProofStep' is called as in the case of no interaction, but then the proof window is updated. For this we were able to employ the same auxiliary functions that are used in the original implementation in 'SearchProofAndShow'.

2.5.3 The User Interface: the Interactive Proof Situation

The suspension of the initial process by means of the *Mathematica*© function 'Dialog' is affected after the proof situation notebook is created and displayed. It turned out that the 'Return' function to quit the child process and continue with the initial process cannot be directly called inside a button function (ie a function that is invoked when the corresponding button is pressed).

Instead, we had to call several functions to write the expression 'Return[ ]' into a notebook, select the respective cell, and evaluate it. In one case we had to add dummy code to make it work. However, this operation also works in child windows, ie notebooks created on top of the Interactive Proof Situation. (The 'Dialog' call is made in the 'InteractiveProofSituation' function.)

In order to be able to explicitly select an inference rule, we had to introduce keywords which identify the available rules of the basic prover uniquely. To obtain a list of all the available rules, we first tried to extract them from the so–called 'down values' of the prover.

The down values are a list of transformation rules that *Mathematica*© internally creates from the respective function definitions. These rules can also be handled explicitly [Wolfram, 1996]. However, it appears that pattern matching is restricted for these down values, therefore we decided to let the basic prover explicitly offer a list of available rules.

This list is provided by an extra function. Although this has the advantage of lucidity, it requires the programmer or maintainer to explicitly add an entry for each rule provided. However, it also enabled us to transfer additional information to the user interface functions, namely about what additional input the respective rule requires. Moreover, it also allows the programmer to influence the order in which the rules appear in an arbitrary manner.

2.5.4 Handling the Information From User Input

Because of problems with return values from button functions within a 'Dialog', we decided to store the results of user input in global variables. These variables are then checked in the 'SearchProof' and 'OneProofStep' functions, outside the 'Dialog' subprocess.

In principle, it would be possible to get the information from user input via the global variables also within the basic prover. However, for the sake of lucidity, we pass this information from the 'OneProofStep' function to the basic prover via an additional argument which has the form of a list, containing the unique keyword for the selected rule, the selected formula, and additional information if needed.

The global variables are handled in such a way that their value is only changed in the functions which collect the user input, and all other access is by 'reading' only. However, those global variables still pose a certain threat to maintenance.

2.5.5 Entering Expressions in *Theorema* Syntax

An interesting detail of implementation is the window for the input of arbitrary *Theorema* expressions in the comfortable, 'natural−style' syntax. Again, we were required to make 'work−arounds' in order to achieve the desired result.

The window is designed to contain three cells:

- A text cell that prompts the user to enter the respective input,
- An input cell, and
- A cell with an 'OK' button.

The input cell should be of type 'Input' in order to allow for the usual, two−dimensional syntax and look of mathematical (*Theorema*) expressions. When the window is opened, this cell should contain a placeholder ("□") which is selected (and then looks like "■").

One should expect to get such a cell with the expression

```
Cell["□", "Input"]
```

where the second argument determines the type of the cell. (Selection has to be done separately, see below.) Unfortunately, the result is a cell which is neither an ordinary input cell nor a text cell. It does not allow for two−dimensional input (ie over−, super−, sub− and underscripts and special constructs like a fraction, etc).

A solution was obtained by first creating an input cell and replacing the (numerical) dummy contents by the placeholder in a second step:

```
NotebookWrite[nb, Cell["0", "Input"], All];
NotebookWrite[nb, "□"];
```

The third argument in the first call of 'NotebookWrite' makes the contents of the new cell, ie the zero, the selection right after it is created. Therefore "0" is replaced by "□" with the second expression.

Moving the selection within a notebook does not work exactly as we would have expected following the description of the respective functions [Wolfram, 1996]. It took a little experimenting to work with it. The purpose was to select the placeholder after the notebook is fully built up, ie after the button is inserted.

The next problem was to make *Theorema* expressions out of the cell contents, ie to translate the *Mathematica*© 'boxes' (the data structure for two−dimensional expressions) and their contents into the internal data structure of *Theorema*. This involves *Mathematica*'s© 'MakeExpression' function as well as *Theorema*−specific pre− and post−processing.

We think that this translation should already be done in the function which collects the input. The basic provers of *Theorema* already expect all expressions to be in this form. Performing the translation somewhere else, for instance in the 'SearchProof' function, would require to re−set the value of the global variables in breach of our self−imposed restrictions for handling global variables (see above).

Unfortunately, the post−processing did not work within this input collection function. So we tried to make the whole translation outside, namely in the basic prover (because this is the place where it is used, and we did not want to involve a third place which later would also have to be taken into account for maintenance). This did not work either.

However, it works when pre−processing and 'MakeExpression' are done in the collection function and post−processing is done in the basic prover. This division is hardly intuitive and will pose another challenge for maintenance.

Yet another challenge was to enable the use of 'arbitrary but fixed' constants in the input. These constants have a special data structure in *Theorema*, comprising the name of the variable which they substituted and a sort of 'index' which makes the identifier unique.

When displayed in the proof output, the ’index’ is written as a subscript to the name – which makes copy–and–paste impossible, as the internal data structure is not retained. And as subscripts can also have another semantics in *Theorema,* namely real indexing, we could not simply translate all subscripts in the input to an ’arbitrary but fixed’ constant. And, of course, it must also be checked whether such a constant actually already exists.

We considered it impractical to urge the user to input the full data structure of an ’arbitrary but fixed’ constant. Instead, we relied on the assumption that if the user would input an identifier with a subscript which corresponds to such an existing constant – and then actually looks the same as the latter when displayed in a notebook – then he or she would actually mean this ’arbitrary but fixed’ constant and did not want to introduce a new identifier which accidently looks the same.

Consequently, we scan each respective user input for subscripts and check whether it matches an existing ’arbitrary but fixed’ constant. If a match is found, all occurrences of it are transformed to the corresponding ’arbitrary but fixed’ identifier with the internal representation as described above. This is done in the basic prover, after the post–processing of the input.

For entering a complete formula including a label, we basically use the same input window, but supplemented with an additional input cell for the label, where the placeholder is already placed between double quotes to render the input a string.

First we wanted to have both input fields in one cell, to support the usual Theorema style which looks like this:

```
< expression > "<label>"
```

Unfortunately, we were not able to handle the ’SelectionMove’ function such that both fields could be read one after the other rather than together, which would have made the parsing and transforming of the input more difficult. So we made two different input cells, with a prompt for each of them.

2.5.6 Changes in the Basic Prover

One of the original requirements for the interactive proving interface was that it should work with any *Theorema* prover. However, it turned out that selecting inference rules could not be achieved without either altering the general structure of the rules of the basic provers or, as we did, adding rules adapted for interaction.

As mentioned above, these rules have an additional argument in the form of a list, containing a keyword for the rule and all the user input. In most cases we were able to retain the original algorithm of the rule, so that we just had to copy those rules and modify them. The most significant modification was to use formulae from the user input rather than from the knowledge base.

For each rule that was made available for explicit selection, we had to add yet another rule, which is a default rule with the same keyword. If the arguments given, which correspond to the user input, do not fit, 'False' is returned, signifying that the rule is not applicable with the given input. This is then handled in the function 'SearchProof', and an error message is written into the proof window. This message is not permanent and disappears with the next (successful) proof step.

The Predicate Logic Prover, which we used for the implementation of interaction as it is the most basic of the natural deduction provers, often shortens the proof by wrapping two or more basic proof steps into one. However, the neophyte in logic should only use basic steps in the beginning. Therefore we added such basic steps where we did not find them implemented in the Predicate Logic Prover.

In addition to that, we had to add special rules which are only required for interaction, such as deleting an assumption from the knowledge base. Deleting assumptions could be achieved by directly accessing the proof object, but it turned out that implementation as an extra rule is not only much more straight−forward, but also much more lucid and therefore better maintainable.

The insertion of a new subgoal is a little more complicated. It has to create three new branches in the proof tree: One is the proof of the subgoal. If this succeeds, the new proof situation has the original goal again, but the knowledge base is supplemented with the subgoal as a new assumption. If the subgoal cannot be proved, the proof goes on with exactly the same proof situation as we had immediately before inserting the new subgoal.

One possible node of a proof tree in *Theorema* is an IF node. However, it turned out that this has not exactly the required semantics, as the 'else' branch is only taken if the 'if' branch is *disproved*. If the proof of the 'if' branch fails, the whole node fails.

Therefore we implemented the rule for a new subgoal using an OR and an AND node in the following way:

```
BasicProver[{"New subgoal", newgoal}, goal, assumptions] :=
 OR[
  AND[
   ProofStep[newgoal, assumptions],
   ProofStep[goal, Append[assumptions, newgoal]]
  ],
  ProofStep[goal, assumptions]
 ]
```

(The rule is considerably simplified and not given in *Mathematica*© syntax. For the sake of intelligibility, also the function names have been changed.)

The above structure is reflected in the output in the proof window, which shows the three new branches of the proof tree:

- Try to prove the new subgoal.
- If the subgoal was proved, continue with the proof of the initial goal with the subgoal as an additional assumption.
- If the subgoal could not be proved, continue with the proof of the initial goal under the initial assumptions.

(For details, see section 2.6.2, 'Tutorial Examples': 'Manipulating the knowledge base'.)

Also aborting the proof was implemented as an extra rule, as was the insertion of additional text given by the user. In the latter case, this was necessary to render the added text permanent. Things are different in the case of the error message which is written to the proof notebook if the selected rule cannot be applied to the selected formula or other input does not match. We do not want such a message to be permanent, and therefore we did not make a prover rule for this (this is actually handled in 'SearchProof').

2.5.7 Resumé

Summing up, we have been able to implement the primary goal, which is an interface for interactive proving in *Theorema,* although the actual solution may be hard to maintain due to 'work–arounds' to cope with insufficient data structures and technical difficulties with *Mathematica* 3.0©.

For reasons given, it is not possible to use the interactive interface with an arbitrary *Theorema* prover without supplementing the basic provers involved.

The basic feature of the interface is working in a satisfactory way, which is the possibility to apply a particular inference rule of the basic prover to a particular formula, and to give an arbitrary term for substituting a variable.

So far, we have implemented in particular those features which are required for the system's employment in student training. The basic feature for professional users, the possibility to halt automatic proving at 'critical points', is still missing, as is the check of the applicability of inference rules before listing them.

Going back in the proof tree to start anew from a certain point has not been realised. This would require to either build up a second proof tree to memorise each proof situation in detail or to re–calculate the proof starting at the initial proof situation. Presently, the proof tree (ie the proof object) is held as slim as possible.

However, it is possible to remove any assumption from the knowledge base at any time during the proof, and to add a new subgoal to serve as a lemma. Other features that were implemented enable the user to

- insert text in the proof,
- let the prover do the next step automatically,
- let the prover finish the proof automatically, and
- abort the proof.

2.6 Tutorial Examples

In the following proof examples we want to give an impression of how an interactive proof develops and what the user interface looks like. In the electronic version of this thesis – as a *Mathematica*© notebook – the input cells can be evaluated to repeat the examples and experiment with them.

The first lines are needed to start *Theorema* and load the required environment:

```
Needs["Theorema‘"]

Needs["Theorema‘Provers‘UserProvers‘PredicateProver‘"]
```

2.6.1 A Simple One to Get Started

We start with a very simple proof to get accustomed to the system. We consider a few theorems which can be derived from the first four Peano axioms for the Natural Numbers:

```
Definition["Natural Numbers", any[x, y, P],
 isNatural[0]                                          "0"
 isNatural[x] ⇒ isNatural[x⁺]                          "x₊"
 isNatural[x] ⇒ ¬ (0 = x⁺)                             "0≠x₊"
 (isNatural[x] ∧ isNatural[y] ∧ (x⁺ = y⁺)) ⇒ (x = y)   "x₊≠y₊"]
```

First we want to prove that 0^+ is a natural number:

```
Proposition["0₊",
 isNatural[0⁺]  "0₊"]
```

The proof is initialised with the following command:

```
Prove[Proposition["0₊"], using -> {Definition["Natural Numbers"]},
 by -> PredicateProver, ProverOptions -> {Interaction -> "stepwise"}]
```

(Note that the keywords of *Theorema* 'formal text' such as 'Definition' or 'Proposition' are highlighted, whereas commands such as 'Prove' are not.)

Starting the proof opens a new window – the proof window – displaying the initial proof situation, ie it lists the goal and the assumptions. Here, every proof step will be recorded sequentially in a natural–style language:

Prove:

(0+) $\text{isNatural}[0^+]$,

under the assumptions:

(0) $\text{isNatural}[0]$,

(x+) $\forall_x (\text{isNatural}[x] \Rightarrow \text{isNatural}[x^+])$,

(0≠x+) $\forall_x (\text{isNatural}[x] \Rightarrow (0 \neq x^+))$,

(x+≠y+) $\forall_{x,y} (\text{isNatural}[x] \wedge \text{isNatural}[y] \wedge (x^+ = y^+) \Rightarrow (x = y))$.

Pending proof of (0+).

Different colours are highlighting different elements: all labels of formulae are in blue, the goal is red, and assumptions are displayed in magenta. (These are the default colours, the user can change them by means of a global option.)

Another new window that appears is the Interactive Proof Situation notebook:

The Proof Situation

Prove:

0 + isNatural[0⁺]

Under the assumptions:

0	isNatural[0]
x +	$\forall_x$ (isNatural[x] ⇒ isNatural[x⁺])
0 ≠ x +	$\forall_x$ (isNatural[x] ⇒ (0 ≠ x⁺))
x + ≠ y +	$\forall_{x,y}$ (isNatural[x] ∧ isNatural[y] ∧ (x⁺ = y⁺) ⇒ (x = y))

Click on the tag of a formula to use it in the next step.

Options:

Add a new subgoal

Perform next step automatically

Finish automatically

Add text to the proof

Abort the proof / subproof

We click on the button of the second assumption, "x+", because we want to transform this assumption in the first step. A new window pops up with a list of inference rules which are – in principle – available for the next step:

Available Inference Rules

Select a rule to be applied on the assumption:

(x +) $\underset{x}{\forall}$ (isNatural[x] ⇒ isNatural[x^+])

Conclusion is assumption

⋮

Substitute universally quantified variable

⋮

Replace using equivalence

Options:

Remove Assumption from the Knowledge Base

Abort

We select "Substitute universally quantified variable". Then we are prompted to enter a term in another new window as the rule selection window is closed:

Enter a term:

▪

OK

The placeholder – the square in the middle cell – is already selected so that we just have to start writing. (If one would click somewhere else inside this window, the placeholder would

look like this: □. Then one has to click it to make it selected again. Input must be given in the very cell with the placeholder, and the placeholder must be *replaced*.)

We enter "0" (without the quotes!) and press the "OK" button. The window disappears, and in the new proof situation window we find a new formula added to the end of the knowledge base:

```
1  isNatural[0] ⇒ isNatural[0⁺]
```

In the proof window, we find an explanation of what happened in this proof step:

Prove:

(0+) $\text{isNatural}[0^+]$,

under the assumptions:

(0) $\text{isNatural}[0]$,

(x+) $\forall_x (\text{isNatural}[x] \Rightarrow \text{isNatural}[x^+])$,

(0≠x+) $\forall_x (\text{isNatural}[x] \Rightarrow (0 \neq x^+))$,

(x+≠y+) $\forall_{x,y} (\text{isNatural}[x] \wedge \text{isNatural}[y] \wedge (x^+ = y^+) \Rightarrow (x = y))$.

In the assumption (x+) we substitute the universally quantified variable x by 0 and obtain

(1) $\text{isNatural}[0] \Rightarrow \text{isNatural}[0^+]$.

Pending proof of (0+).

Note that, in the proof notebook, the new formula is *not* added to the assumptions, as only the original assumptions are listed here. So we always see the proof situation where we initially started.

For the next step, we click the new formula "1" (the label is created automatically and is guaranteed to be unique). From the inference rules, we choose "Modus Ponens". Modus Ponens can only be applied if we have an assumption which matches the left−hand side of the implication, therefore a new window appears from where we have to select such an assumption:

Select the other assuption required for the rule Modus Ponens:

0	isNatural[0]
x +	$\forall_x$ (isNatural[x] ⇒ isNatural[x^+])
0 ≠ x +	$\forall_x$ (isNatural[x] ⇒ (0 ≠ x^+))
x + ≠ y +	$\forall_{x,y}$ (isNatural[x] ∧ isNatural[y] ∧ (x^+ = y^+) ⇒ (x = y))
1	isNatural[0] ⇒ isNatural[0^+]

Formula "0" is obviously matching. (Note that if one accidently selects a wrong formula, an error message will be written to the proof window, but nothing else happens and one can simply continue with the proof. The error message will disappear after the next proof step. The same happens if the selected rule cannot be applied to the selected formula, or if other input is incorrect.)

After we clicked on formula "0", the right–hand side of assumption "1" is added as a new assumption to the knowledge base:

2 isNatural[0^+]

The respective text in the proof window reads:

From (0) and (1) we obtain by modus ponens

(2) isNatural[0^+].

We see that formula "2" is identical with the proof goal. However, the proof process is still running, and the proof window still says 'pending'. Also this final step has to be made explicitly. We click formula "2", and from the rules we choose "Conclusion is assumption". Now the proof situation window disappears, and the final proof looks like this:

Prove:

(0+) $\text{isNatural}[0^+]$,

under the assumptions:

(0) $\text{isNatural}[0]$,

(x+) $\forall_x (\text{isNatural}[x] \Rightarrow \text{isNatural}[x^+])$,

(0≠x+) $\forall_x (\text{isNatural}[x] \Rightarrow (0 \neq x^+))$,

(x+≠y+) $\forall_{x,y} (\text{isNatural}[x] \wedge \text{isNatural}[y] \wedge (x^+ = y^+) \Rightarrow (x = y))$.

In the assumption (x+) we substitute the universally quantified variable x by 0 and obtain

(1) $\text{isNatural}[0] \Rightarrow \text{isNatural}[0^+]$.

From (0) and (1) we obtain by modus ponens

(2) $\text{isNatural}[0^+]$.

Formula (0+) is true because it is identical to (2).

2.6.2 Manipulating the Knowledge Base

Let us now demonstrate how to control the knowledge base during a proof. Under the same assumptions as in the first example, we try the following proof:

```
Proposition["0≠x++", any[x],
  isNatural[x] ⇒ 0 ≠ ((x⁺)⁺)   "0≠x++"]

Prove[Proposition["0≠x++"], using -> {Definition["Natural Numbers"]},
 by -> PredicateProver, ProverOptions -> {Interaction -> "stepwise"}]
```

Prove:

(0≠x++) $\forall_{x}$ (isNatural[x] ⇒ (0 ≠ (x^{+})$^{+}$)),

under the assumptions:

(0) isNatural[0],

(x+) $\forall_{x}$ (isNatural[x] ⇒ isNatural[x^{+}]),

(0≠x+) $\forall_{x}$ (isNatural[x] ⇒ (0 ≠ x^{+})),

(x+≠y+) $\forall_{x,y}$ (isNatural[x] ∧ isNatural[y] ∧ (x^{+} = y^{+}) ⇒ (x = y)).

First of all, one might have the feeling that the assumption (x+≠y+) will not be needed for this proof. If the knowledge base is very large, it may be useful to get rid of redundant assumptions to get a better overview. The knowlege base may not be very large in this case, but we will show how it works.

To remove assumption (x+≠y+), we click it in the proof situation window. in the rule selection window, we click the option 'Remove assumption from the knowledge base' (at the bottom of the window).

In the new proof situation, the formula has disappeared. However, it is still listed in the proof window:

Prove:

(0≠x++) $\underset{x}{\forall}\,(\text{isNatural}[x] \Rightarrow (0 \neq (x^{+})^{+}))$,

under the assumptions:

(0) $\text{isNatural}[0]$,

(x+) $\underset{x}{\forall}\,(\text{isNatural}[x] \Rightarrow \text{isNatural}[x^{+}])$,

(0≠x+) $\underset{x}{\forall}\,(\text{isNatural}[x] \Rightarrow (0 \neq x^{+}))$,

(x+≠y+) $\underset{x,y}{\forall}\,(\text{isNatural}[x] \wedge \text{isNatural}[y] \wedge (x^{+} = y^{+}) \Rightarrow (x = y))$.

Removed Assumption (x+≠y+) from the knowledge base.

Pending proof of (0≠x++).

The reason is that in the proof trace it must always be clear what the initial assumptions were under which the goal was finally proved or not. By the way, one can remove an assumption at any time, not just in the beginning, even if it was already used in a previous step. The effect of removal is that one cannot use the respective assumption any more below the proof node where it was removed (in the respective branch of the proof tree).

Now we proceed with the proof by taking the 'x' in the goal arbitrary but fixed: we click the goal in the proof situation window and choose the rule 'Arbitrary but fixed'. The name of the new constant is determined by the system – see the lines in the proof window:

For proving (0≠x++) we take all variables arbitrary but fixed and prove:

(1) $\text{isNatural}[x_0] \Rightarrow (0 \neq (x_0{}^{+})^{+})$.

Now we select the new goal and apply the 'Deduction rule':

We prove (1) by the deduction rule.

We assume

(2) isNatural[x_0]

and show

(3) $0 \neq (x_0{}^+)^+$.

The new proof situation looks like this:

Prove:

3 $0 \neq (x_0{}^+)^+$

Under the assumptions:

2	isNatural[x_0]
0	isNatural[0]
x +	$\forall_x$ (isNatural[x] ⇒ isNatural[x^+])
0 ≠ x +	$\forall_x$ (isNatural[x] ⇒ (0 ≠ x^+))

Now we see that the right–hand side of formula (0≠x+) would match the goal if we would substitute 'x' by ($x_0{}^+$). In order to obtain a match for the left–hand side, we have to show that ($x_0{}^+$) is a natural number. We try to prove this as a 'lemma'. In the proof situation window, we click the option 'Add a new subgoal'. A new window opens, similar to the one where we entered a substitution term in the first example:

Enter a formula:

■

Enter a label:

"□"

OK

The first placeholder is selected; here we enter the formula:

```
isNatural[x0+] .
```

The placeholder for the label is already enframed in double quotes to make it a string. We enter (eg):

```
"isNat[x0+]"
```

and press 'OK'. The new subgoal is depicted in the new proof situation:

isNat[x0 +] isNatural[x_0^+]

In the proof window, the following text was added:

⋮

We try to prove the formula

(isNat[x0+]) isNatural[x_0^+]

to use it as a lemma.

Alternative proof 1: pending

If we can prove (isNat[x0+]), we can add it to the knowledge base and continue to prove (3).

We try to prove (isNat[x0+]).

Pending proof of (isNat[x0+]).

Having proved (isNat[x0+]), we add it to the knowledge base and continue to prove (3).

Pending proof of (3).

Alternative proof 2: pending

If we are not able to prove (isNat[x0+]), we continue to prove (3) without it.

Pending proof of (3).

The interpretation is that three new branches were created in the proof tree – to which the 'Pending proof' lines correspond. There are two possibilities to proceed with the proof of (3): either we are able to prove (isNat[x0+]), then we can continue the proof with the additional assumption (isNat[x0+]), or we have to continue without it (see the last line) – just as if we had never tried that new subgoal. (If one wants to abandon the proof attempt of such a subgoal and go on with the alternative branch, one just has to click 'Abort the proof / subproof' under the options of the proof situation window.)

To prove (isNat[x0+]), we first substitute the variable 'x' in formula (x+) by x_0. We click (x+) in the proof situation window and select the rule 'Substitute universally quantified variable'. We are prompted to enter a term and enter 'x_0'.

This last step is done in the first branch of the previous proof node, and the text that refers to it is inserted into the proof window accordingly:

⋮

We try to prove (isNat[x0+]).

In the assumption (x+) we substitute the universally quantified variable x by x_0 and obtain

(4) $\text{isNatural}[x_0] \Rightarrow \text{isNatural}[x_0{}^+]$.

Pending proof of (isNat[x0+]).

⋮

Now we click formula (4) and apply 'Modus Ponens'; (2) is the appropriate formula that matches the left–hand side. The new formula

5 $\text{isNatural}[x_0{}^+]$

is identical with the current goal, so we click it and choose the rule 'Conclusion is assumption'. Now that the subgoal (isNat[x0+]) is proved, it is moved to the knowledge base, and the initial goal (3) reappears:

Prove:

3 $0 \neq (x_0^+)^+$

Under the assumptions:

2	isNatural$[x_0]$
0	isNatural$[0]$
x +	$\forall_x$ (isNatural$[x] \Rightarrow$ isNatural$[x^+]$)
0 ≠ x +	$\forall_x$ (isNatural$[x] \Rightarrow (0 \neq x^+)$)
isNat[x0 +]	isNatural$[x_0^+]$

In the proof window, the 'Pending' line of the first branch has disappeard, and we can continue with the second branch, ie we try to prove (3) under the additional assumption (isNat[x0+]).

Now we use formula (0≠x+) and substitute the universally quantified variable x by x_0^+. To the new formula

6 isNatural$[x_0^+] \Rightarrow (0 \neq (x_0^+)^+)$

we apply Modus Ponens using (isNat[x0+]) as the match for the left–hand side. We get

7 $0 \neq (x_0^+)^+$

which is identical with the goal. This is the final proof trace:

Prove:

(0≠x++) $\forall_{x}$ (isNatural[x] ⇒ (0 ≠ (x^{+})$^{+}$)),

under the assumptions:

(0) isNatural[0],

(x+) $\forall_{x}$ (isNatural[x] ⇒ isNatural[x^{+}]),

(0≠x+) $\forall_{x}$ (isNatural[x] ⇒ (0 ≠ x^{+})),

(x+≠y+) $\forall_{x,y}$ (isNatural[x] ∧ isNatural[y] ∧ (x^{+} = y^{+}) ⇒ (x = y)).

Removed Assumption (x+≠y+) from the knowledge base.

For proving (0≠x++) we take all variables arbitrary but fixed and prove:

(1) isNatural[x_0] ⇒ (0 ≠ ($x_0{}^{+}$)$^{+}$).

We prove (1) by the deduction rule.

We assume

(2) isNatural[x_0]

and show

(3) 0 ≠ ($x_0{}^{+}$)$^{+}$.

We try to prove the formula

(isNat[x0+]) isNatural[$x_0{}^{+}$]

to use it as a lemma.

Alternative proof 1: proved

If we can prove (isNat[x0+]), we can add it to the knowledge base and continue to prove (3).

We try to prove (isNat[x0+]).

In the assumption (x+) we substitute the universally quantified variable x by x_0 and obtain

(4) isNatural[x_0] ⇒ isNatural[$x_0{}^{+}$].

From (2) and (4) we obtain by modus ponens

(5) isNatural[$x_0{}^+$].

Formula (isNat[x0+]) is true because it is identical to (5).

Having proved (isNat[x0+]), we add it to the knowledge base and continue to prove (3).

In the assumption (0≠x+) we substitute the universally quantified variable x by $x_0{}^+$ and obtain

(6) isNatural[$x_0{}^+$] ⇒ $(0 \neq (x_0{}^+)^+)$.

From (isNat[x0+]) and (6) we obtain by modus ponens

(7) $0 \neq (x_0{}^+)^+$.

Formula (3) is true because it is identical to (7).

Alternative proof 2: pending

If we are not able to prove (isNat[x0+]), we continue to prove (3) without it.

Pending proof of (3).

The last branch is still pending. As it is a branch of an 'Or' node and the corresponding other branch was proved, the whole proof succeded nevertheless. (If we would have chosen the option 'presentation–>"NoPending"' in the proof call, the last three lines would not show up any more.)

Of course, we could have performed this proof without inserting a new subgoal – even the proof steps would have been basically the same. We would have arrived at formula (5) and we could have used it instead of the subgoal (isNat[x0+]) as these two formulae are identical.

However, inserting a subgoal can help to structure a proof, just as a lemma does in proving by hand. Also the self–chosen label can add to comprehensibility.

2.6.3 A More Sophisticated Example

Let us now consider a 'more interesting' example which most automated provers will find difficult to prove. We want to prove that the sum of two zero sequences is again a zero sequence:

```
Proposition["sum of zero sequences", any[f, g],
  ((f⟶0) ∧ (g⟶0)) ⇒ ((f⊕g)⟶0)  "sum of zero sequences"
 ]
```

under the assumptions:

```
Definition["zero sequence", any[f],
  (f⟶0) ⇔ (∀ ∃ ∀ (|f[n]| < ∈))  "zero sequence"]
          ∈ N n
          ∈>0  n>N

Definition["sum of sequences", any[f, g, n],
  (f⊕g)[n] = f[n] + g[n]  "sum of sequences"]
```

As neither *Mathematica*'s nor *Theorema*'s evaluation power is currently available in *Theorema* provers, we have to add a the following assumptions:

```
Assumption["x > 0 ⇒ x/2 > 0", any[x],
  (x > 0) ⇒ (x/2 > 0)  "x > 0 ⇒ x/2 > 0"]

Assumption["x+y < a", any[x, y, a],
  ((|x| < a/2) ⋀ (|y| < a/2)) ⇒ (|x + y| < a)  "x+y < a"]

Assumption["x > Max[y,z]", any[x, y, z],
  (x > Max[y, z]) ⇒ ((x > y) ∧ (x > z))  "x > Max[y,z]"]
```

We start the proof by the call

```
Prove[Proposition["sum of zero sequences"],
 using -> {Definition["zero sequence"],
   Definition["sum of sequences"],
   Assumption["x > 0 ⇒ x/2 > 0"],
   Assumption["x+y < a"], Assumption["x > Max[y,z]"]},
 by -> PredicateProver,
 ProverOptions -> {Interaction -> "stepwise"}]
```

First we want to apply the definition of 'zero sequence' to the proposition. In the proof situation window, we click the proposition and select the rule 'Replace using equivalence'. We are prompted to select an formula to be used for the replacement, ie an appropriate equivalence from the knowledge base. We choose definition 'zero sequence'. The proof situation window shows the resulting formula as the new sub−proof goal. The corresponding line in the proof window reads:

Using (zero sequence), the goal (sum of zero sequences) is transformed into:

$$(1)\quad \forall_{f,g}\left(\forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|f[n]|<\epsilon)\bigwedge\forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|g[n]|<\epsilon)\Rightarrow \forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|(f\oplus g)[n]|<\epsilon)\right).$$

Applying replacement using the 'sum of sequences' works likewise, only that now we have to choose the rule 'Replace using equality':

Using (sum of sequences), the goal (1) is transformed into:

$$(2)\quad \forall_{f,g}\left(\forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|f[n]|<\epsilon)\bigwedge\forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|g[n]|<\epsilon)\Rightarrow \forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|f[n]+g[n]|<\epsilon)\right).$$

The outermost symbol is 'for all'. In order to proof formula (2), we want to introduce new constants for f and g arbitrary but fixed. We click formula (2) in the proof situation window and select 'Arbitrary but fixed'. The system chooses new but similar symbols for the new constants. The new proof goal is:

$$(3)\quad \forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|f_0[n]|<\epsilon)\bigwedge\forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|g_0[n]|<\epsilon)\Rightarrow \forall_{\epsilon>0}\exists_{N}\forall_{n>N}\ (|f_0[n]+g_0[n]|<\epsilon).$$

These three steps could have been done in a different order. Actually, *Theorema* automated provers usually start with the logically outermost symbol, so the last step – eliminating the universal quantifier – would have been the first. In interactive proving, we have more freedom.

The next step will be to assume the left–hand side of the implication and take the right–hand side as the new subgoal. We click the current goal (3) and select 'Deduction rule':

We assume

$$(4)\quad \underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f_0[n]| < \epsilon) \bigwedge \underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|g_0[n]| < \epsilon)$$

and show

$$(5)\quad \underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f_0[n] + g_0[n]| < \epsilon)\,.$$

In the proof situation, (5) is the new goal and (4) has been added to the knowledge base:

Prove:

5 $\underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f_0[n] + g_0[n]| < \epsilon)$

Under the assumptions:

4	$\underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f_0[n]| < \epsilon) \bigwedge \underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|g_0[n]| < \epsilon)$
zero sequence	$\underset{f}{\forall}\left(f \longrightarrow 0 \Leftrightarrow \underset{\substack{\epsilon\\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f[n]| < \epsilon)\right)$
sum of sequences	$\underset{f,g,n}{\forall}\ ((f \oplus g)[n] = f[n] + g[n])$
x > 0 ⇒ x / 2 > 0	$\underset{x}{\forall}\ (x > 0 \Rightarrow \frac{x}{2} > 0)$
x + y < a	$\underset{a,x,y}{\forall}\ (|x| < \frac{a}{2} \bigwedge |y| < \frac{a}{2} \Rightarrow |x + y| < a)$
x > Max[y, z]	$\underset{x,y,z}{\forall}\ (x > \text{Max}[y, z] \Rightarrow x > y \wedge x > z)$

We continue with a routine step which we already exercised before:

For proving (5) we take all variables arbitrary but fixed and prove:

$$(6)\quad \epsilon_0 > 0 \Rightarrow \underset{N}{\exists}\ \underset{\substack{n\\ n>N}}{\forall}\ (|f_0[n] + g_0[n]| < \epsilon_0)\,.$$

We see that the condition for the quantified variable, $\epsilon > 0$, has been taken out ($\underset{\substack{x\\ P[x]}}{\forall}(Q[x])$ is interpreted by *Theorema* as $\underset{x}{\forall}(P[x] \Rightarrow Q[x])$). Therefore, we apply the deduction rule on (6):

We assume

(7) $\epsilon_0 > 0$

and show

(8) $\exists_N \forall_{n,\, n>N} \left(|f_0[n] + g_0[n]| < \epsilon_0 \right)$.

In order to prove (8), we have to find a suitable N' that satisfies the body of the existentially quantified formula. In order to get an idea for such an N', we turn to the knowledge base for the moment. First, we click assumption (4), which is a conjunction, and apply the rule 'Assumption is conjunction: assume the individual parts':

The conjunction (4) splits into (9) and (10).

The new proof situation:

Prove:

8 $\exists_N \forall_{n,\, n>N} \left(|f_0[n] + g_0[n]| < \epsilon_0 \right)$

Under the assumptions:

7	$\epsilon_0 > 0$
9	$\forall_{\epsilon,\, \epsilon>0} \exists_N \forall_{n,\, n>N} \left(|f_0[n]| < \epsilon \right)$
10	$\forall_{\epsilon,\, \epsilon>0} \exists_N \forall_{n,\, n>N} \left(|g_0[n]| < \epsilon \right)$
zero sequence	$\forall_f \left(f \longrightarrow 0 \Leftrightarrow \forall_{\epsilon,\, \epsilon>0} \exists_N \forall_{n,\, n>N} \left(|f[n]| < \epsilon \right) \right)$
sum of sequences	$\forall_{f,g,n} \left((f \oplus g)[n] = f[n] + g[n] \right)$
x > 0 ⇒ x / 2 > 0	$\forall_x \left(x > 0 \Rightarrow \frac{x}{2} > 0 \right)$
x + y < a	$\forall_{a,x,y} \left(|x| < \frac{a}{2} \bigwedge |y| < \frac{a}{2} \Rightarrow |x + y| < a \right)$
x > Max[y, z]	$\forall_{x,y,z} \left(x > \text{Max}[y,\, z] \Rightarrow x > y \wedge x > z \right)$

Now comes a first really creative step: we have to find an appropriate constant to substitute the universally quantified variable 'ϵ' in assumption (9). This is a point where automated provers will usually have problems. We click (9), and from the rules we choose 'Substitute universally quantified variable'. We are prompted to enter a term in a new window, and we enter

$$\frac{\epsilon_0}{2}.$$

The respective line in the proof window says:

In the assumption (9) we substitute the universally quantified variable $\in$ by $\frac{\epsilon_0}{2}$ and obtain

$$(11) \quad \frac{\epsilon_0}{2} > 0 \Rightarrow \underset{N}{\exists} \underset{n>N}{\forall} \left(|f_0[n]| < \frac{\epsilon_0}{2}\right).$$

ϵ_0 is a constant we took 'arbitrary but fixed' in a previous step. Such constants are represented in *Theorema* in a special internal form, which is reflected in its graphical representation: in contrast to other constants, 'arbitrary but fixed' constants are written in italics (like variables). When we enter such a term, the system checks identifiers with subscripts whether there exists a corresponding 'arbitrary but fixed' constant, and if so, the system transforms it into the latter.

Now we do exactly the same with assumption (10) to get

$$(12) \quad \frac{\epsilon_0}{2} > 0 \Rightarrow \underset{N}{\exists} \underset{n>N}{\forall} \left(|g_0[n]| < \frac{\epsilon_0}{2}\right).$$

Next, in order to be able to apply Modus Ponens to (11) and (12), we have to show

$$\frac{\epsilon_0}{2} > 0\,.$$

Therefore we make this our new subgoal for the moment. We click the option 'Add a new subgoal' in the proof situation window and enter

$$\frac{\epsilon_0}{2} > 0$$

as the formula and (eg) "ϵ / 2 > 0" as the label. The effect on the proof window is the following:

⋮

We try to prove the formula

$$(\epsilon/2 > 0) \quad \frac{\epsilon_0}{2} > 0$$

to use it as a lemma.

Alternative proof 1: pending

If we can prove $(\epsilon/2 > 0)$, we can add it to the knowledge base and continue to prove (8).

We try to prove $(\epsilon/2 > 0)$.

Pending proof of $(\epsilon/2 > 0)$.

Having proved $(\epsilon/2 > 0)$, we add it to the knowledge base and continue to prove (8).

Pending proof of (8).

Alternative proof 2: pending

If we are not able to prove $(\epsilon/2 > 0)$, we continue to prove (8) without it.

Pending proof of (8).

The meaning of the output in the proof window with the three branches (which are now pending) was discussed in detail in the previous example. The current proof situation looks like this:

Prove:

$\epsilon / 2 > 0$	$\frac{\epsilon_0}{2} > 0$

Under the assumptions:

7	$\epsilon_0 > 0$
9	$\forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert f_0[n] \rvert < \epsilon)$
10	$\forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert g_0[n] \rvert < \epsilon)$
zero sequence	$\forall_f \left(f \longrightarrow 0 \Leftrightarrow \forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert f[n] \rvert < \epsilon) \right)$
sum of sequences	$\forall_{f,g,n} ((f \oplus g)[n] = f[n] + g[n])$
$x > 0 \Rightarrow x / 2 > 0$	$\forall_x (x > 0 \Rightarrow \frac{x}{2} > 0)$
$x + y < a$	$\forall_{a,x,y} (\lvert x \rvert < \frac{a}{2} \bigwedge \lvert y \rvert < \frac{a}{2} \Rightarrow \lvert x + y \rvert < a)$
$x > \text{Max}[y, z]$	$\forall_{x,y,z} (x > \text{Max}[y, z] \Rightarrow x > y \wedge x > z)$
11	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_N \forall_{n > N} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
12	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_N \forall_{n > N} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$

Now, in order to prove ($\epsilon/2 > 0$), we select the assumption ($x > 0 \Rightarrow x/2 > 0$) and substitute the universally quantified variable 'x' by 'ϵ_0'. To the new formula (13) we can apply Modus Ponens, as (7) matches the left–hand side. The resulting formula

(14) $\frac{\epsilon_0}{2} > 0$

is identical with the subgoal ($\epsilon/2 > 0$). We select (14) and click the rule 'Conclusion is assumption'. Now that the subgoal is proved, it is automatically moved to the knowledge base, and the initial goal (8) reappears in the proof situation as the current proof goal:

Prove:

8 $\exists_{N} \forall_{\substack{n \\ n>N}} (|f_0[n] + g_0[n]| < \epsilon_0)$

Under the assumptions:

7	$\epsilon_0 > 0$
9	$\forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} (|f_0[n]| < \epsilon)$
10	$\forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} (|g_0[n]| < \epsilon)$
zero sequence	$\forall_{f} \left(f \longrightarrow 0 \Leftrightarrow \forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} (|f[n]| < \epsilon) \right)$
sum of sequences	$\forall_{f,g,n} ((f \oplus g)[n] = f[n] + g[n])$
x > 0 ⇒ x / 2 > 0	$\forall_{x} (x > 0 \Rightarrow \frac{x}{2} > 0)$
x + y < a	$\forall_{a,x,y} (|x| < \frac{a}{2} \bigwedge |y| < \frac{a}{2} \Rightarrow |x + y| < a)$
x > Max[y, z]	$\forall_{x,y,z} (x > \text{Max}[y, z] \Rightarrow x > y \wedge x > z)$
11	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_{N} \forall_{\substack{n \\ n>N}} (|f_0[n]| < \frac{\epsilon_0}{2})$
12	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_{N} \forall_{\substack{n \\ n>N}} (|g_0[n]| < \frac{\epsilon_0}{2})$
∈ / 2 > 0	$\frac{\epsilon_0}{2} > 0$

The proof window reads:

Formula (∈/2 > 0) is true because it is identical to (14).

Having proved (∈/2 > 0), we add it to the knowledge base and continue to prove (8).

Again, it would not have been necessary to introduce (∈/2 > 0) as a new subgoal or 'lemma', but we believe that this feature makes life a little easier. Now we can make use of (∈/2 > 0):

From (∈/2 > 0) and (11) we obtain by modus ponens

(15) $\exists_{N} \forall_{\substack{n \\ n>N}} (|f_0[n]| < \frac{\epsilon_0}{2})$.

From (∈/2 > 0) and (12) we obtain by modus ponens

(16) $\underset{N}{\exists}\ \underset{n>N}{\forall}\ (|g_0[n]| < \frac{\epsilon_0}{2})$.

In the next step, we substitute the 'N' in (15) and (16) by 'Skolem constants'. This means that, knowing that in both cases an 'N' exists such that the body of the formula is true, we take a particular object, N_0 and N_1, respectively, for which we assume that the body of the respective formula holds. We click the formulae in turn and apply the rule 'Introduce a Skolem constant for existentially quantified variable'. The new names are chosen automatically:

By (15) we can take appropriate values such that:

(17) $\underset{n>N_0}{\forall}\ (|f_0[n]| < \frac{\epsilon_0}{2})$.

By (16) we can take appropriate values such that:

(18) $\underset{n>N_1}{\forall}\ (|g_0[n]| < \frac{\epsilon_0}{2})$.

Here is the current proof situation:

Prove:

8 $\underset{N}{\exists}\ \underset{n>N}{\forall}\ (|f_0[n] + g_0[n]| < \epsilon_0)$

Under the assumptions:

7	$\epsilon_0 > 0$
9	$\underset{\epsilon>0}{\forall}\ \underset{N}{\exists}\ \underset{n>N}{\forall}\ (|f_0[n]| < \epsilon)$
10	$\underset{\epsilon>0}{\forall}\ \underset{N}{\exists}\ \underset{n>N}{\forall}\ (|g_0[n]| < \epsilon)$
zero sequence	$\underset{f}{\forall}\left(f \longrightarrow 0 \Leftrightarrow \underset{\epsilon>0}{\forall}\ \underset{N}{\exists}\ \underset{n>N}{\forall}\ (|f[n]| < \epsilon)\right)$
sum of sequences	$\underset{f,g,n}{\forall}\ ((f \oplus g)[n] = f[n] + g[n])$
x > 0 ⇒ x / 2 > 0	$\underset{x}{\forall}\ (x > 0 \Rightarrow \frac{x}{2} > 0)$
x + y < a	$\underset{a,x,y}{\forall}\ (|x| < \frac{a}{2} \bigwedge |y| < \frac{a}{2} \Rightarrow |x + y| < a)$
x > Max[y, z]	$\underset{x,y,z}{\forall}\ (x > \text{Max}[y, z] \Rightarrow x > y \wedge x > z)$

11	$\frac{\epsilon_0}{2} > 0 \Rightarrow \underset{N}{\exists} \underset{\substack{n \\ n>N}}{\forall} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
12	$\frac{\epsilon_0}{2} > 0 \Rightarrow \underset{N}{\exists} \underset{\substack{n \\ n>N}}{\forall} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$
$\epsilon / 2 > 0$	$\frac{\epsilon_0}{2} > 0$
17	$\underset{\substack{n \\ n>N_0}}{\forall} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
18	$\underset{\substack{n \\ n>N_1}}{\forall} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$

(Please note that (15) and (16) have disappeared from the knowledge base as they are no longer needed.)

Now we can turn back on the goal. Again, we have to take a step which requires creativity: we might now have an idea what an example for the 'N' in (8) could look like such that the body of the goal is satisfied. For this we make use of the new constants which we introduced in the last two steps:

In the goal (8) we substitute the existentially quantified variable N by $\text{Max}[N_0, N_1]$ and try to prove

(19) $$\underset{\substack{n \\ n>\text{Max}[N_0,N_1]}}{\forall} (\lvert f_0[n] + g_0[n] \rvert < \epsilon_0).$$

The next two steps are routine steps again:

For proving (19) we take all variables arbitrary but fixed and prove:

(20) $$n_0 > \text{Max}[N_0, N_1] \Rightarrow \lvert f_0[n_0] + g_0[n_0] \rvert < \epsilon_0.$$

We prove (20) by the deduction rule.

We assume

(21) $$n_0 > \text{Max}[N_0, N_1]$$

and show

(22) $$\lvert f_0[n_0] + g_0[n_0] \rvert < \epsilon_0.$$

Now we turn to the knowledge base again, with another two basically creative steps. But, of course, creativity also lies in this specific alternation of simplifying the goal and transforming the knowledge base, and the selection of particular assumptions at a certain proof situation.

In the assumption (17) we substitute the universally quantified variable n by n_0 and obtain

(23) $n_0 > N_0 \Rightarrow |f_0[n_0]| < \frac{\epsilon_0}{2}$.

In the assumption (18) we substitute the universally quantified variable n by n_0 and obtain

(24) $n_0 > N_1 \Rightarrow |g_0[n_0]| < \frac{\epsilon_0}{2}$.

In order to apply Modus Ponens on (23) and (24), we have to construct the left–hand sides somehow within the knowledge base. This time we make it directly, without explicitly starting with a new subgoal (compare with the proof of $(\epsilon/2 > 0)$ above). Let us have a look at the current proof situation:

Prove:

22 $|f_0[n_0] + g_0[n_0]| < \epsilon_0$

Under the assumptions:

21	$n_0 > \text{Max}[N_0, N_1]$
7	$\epsilon_0 > 0$
9	$\forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert f_0[n] \rvert < \epsilon)$
10	$\forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert g_0[n] \rvert < \epsilon)$
zero sequence	$\forall_f \left(f \longrightarrow 0 \Leftrightarrow \forall_{\epsilon > 0} \exists_N \forall_{n > N} (\lvert f[n] \rvert < \epsilon) \right)$
sum of sequences	$\forall_{f,g,n} ((f \oplus g)[n] = f[n] + g[n])$
x > 0 ⇒ x / 2 > 0	$\forall_x (x > 0 \Rightarrow \frac{x}{2} > 0)$
x + y < a	$\forall_{a,x,y} (\lvert x \rvert < \frac{a}{2} \bigwedge \lvert y \rvert < \frac{a}{2} \Rightarrow \lvert x + y \rvert < a)$
x > Max[y, z]	$\forall_{x,y,z} (x > \text{Max}[y, z] \Rightarrow x > y \wedge x > z)$
11	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_N \forall_{n > N} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
12	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_N \forall_{n > N} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$
ϵ / 2 > 0	$\frac{\epsilon_0}{2} > 0$
17	$\forall_{n > N_0} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$

18	$\forall_{\substack{n \\ n>N_1}} \left(\left| g_0[n] \right| < \frac{\epsilon_0}{2} \right)$
23	$n_0 > N_0 \Rightarrow \left| f_0[n_0] \right| < \frac{\epsilon_0}{2}$
24	$n_0 > N_1 \Rightarrow \left| g_0[n_0] \right| < \frac{\epsilon_0}{2}$

We proceed as follows:

In the assumption (x > Max[y,z]) we substitute the universally quantified variable x by n_0 and obtain

(25) $\forall_{y,z} \left(n_0 > \text{Max}[y, z] \Rightarrow n_0 > y \wedge n_0 > z \right)$.

In the assumption (25) we substitute the universally quantified variable y by N_0 and obtain

(26) $\forall_{z} \left(n_0 > \text{Max}[N_0, z] \Rightarrow n_0 > N_0 \wedge n_0 > z \right)$.

In the assumption (26) we substitute the universally quantified variable z by N_1 and obtain

(27) $n_0 > \text{Max}[N_0, N_1] \Rightarrow n_0 > N_0 \wedge n_0 > N_1$.

From (21) and (27) we obtain by modus ponens

(28) $n_0 > N_0 \wedge n_0 > N_1$.

The conjunction (28) splits into (29) and (30).

The proof situation now:

Prove:

22 $\left| f_0[n_0] + g_0[n_0] \right| < \epsilon_0$

Under the assumptions:

21	$n_0 > \text{Max}[N_0, N_1]$
7	$\epsilon_0 > 0$
9	$\forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} \left(\left| f_0[n] \right| < \epsilon \right)$
10	$\forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} \left(\left| g_0[n] \right| < \epsilon \right)$
zero sequence	$\forall_{f} \left(f \longrightarrow 0 \Leftrightarrow \forall_{\substack{\epsilon \\ \epsilon>0}} \exists_{N} \forall_{\substack{n \\ n>N}} \left(\left| f[n] \right| < \epsilon \right) \right)$

sum of sequences	$\forall_{f,g,n} ((f \oplus g)[n] = f[n] + g[n])$
$x > 0 \Rightarrow x/2 > 0$	$\forall_{x} (x > 0 \Rightarrow \frac{x}{2} > 0)$
$x + y < a$	$\forall_{a,x,y} (\lvert x \rvert < \frac{a}{2} \wedge \lvert y \rvert < \frac{a}{2} \Rightarrow \lvert x + y \rvert < a)$
$x >$ Max[y, z]	$\forall_{x,y,z} (x >$ Max$[y, z] \Rightarrow x > y \wedge x > z)$
11	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_{N} \forall_{n, n>N} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
12	$\frac{\epsilon_0}{2} > 0 \Rightarrow \exists_{N} \forall_{n, n>N} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$
$\epsilon/2 > 0$	$\frac{\epsilon_0}{2} > 0$
17	$\forall_{n, n>N_0} (\lvert f_0[n] \rvert < \frac{\epsilon_0}{2})$
18	$\forall_{n, n>N_1} (\lvert g_0[n] \rvert < \frac{\epsilon_0}{2})$
23	$n_0 > N_0 \Rightarrow \lvert f_0[n_0] \rvert < \frac{\epsilon_0}{2}$
24	$n_0 > N_1 \Rightarrow \lvert g_0[n_0] \rvert < \frac{\epsilon_0}{2}$
25	$\forall_{y,z} (n_0 >$ Max$[y, z] \Rightarrow n_0 > y \wedge n_0 > z)$
26	$\forall_{z} (n_0 >$ Max$[N_0, z] \Rightarrow n_0 > N_0 \wedge n_0 > z)$
27	$n_0 >$ Max$[N_0, N_1] \Rightarrow n_0 > N_0 \wedge n_0 > N_1$
29	$n_0 > N_0$
30	$n_0 > N_1$

Now we can apply Modus Ponens on (23) and (24):

From (29) and (23) we obtain by modus ponens

(31) $\lvert f_0[n_0] \rvert < \frac{\epsilon_0}{2}$.

From (30) and (24) we obtain by modus ponens

(32) $\lvert g_0[n_0] \rvert < \frac{\epsilon_0}{2}$.

Now we can already see that the current goal (22) can be derived from (31) and (32) with the employment of assumption (x+y < a). However, we have to make it step by step: not only because the system requires it, but also to make sure that we made no mistake in our minds. The quantifier of (x+y < a) covers three variables; we have to substitute them in turns (in the current implementation, we cannot do this in one step):

In the assumption (x+y < a) we substitute the universally quantified variable a by ϵ_0 and obtain

(33) $\underset{x,y}{\forall} (|x| < \frac{\epsilon_0}{2} \bigwedge |y| < \frac{\epsilon_0}{2} \Rightarrow |x + y| < \epsilon_0)$.

In the assumption (33) we substitute the universally quantified variable x by $f_0[n_0]$ and obtain

(34) $\underset{y}{\forall} (|f_0[n_0]| < \frac{\epsilon_0}{2} \bigwedge |y| < \frac{\epsilon_0}{2} \Rightarrow |f_0[n_0] + y| < \epsilon_0)$.

In the assumption (34) we substitute the universally quantified variable y by $g_0[n_0]$ and obtain

(35)

$|f_0[n_0]| < \frac{\epsilon_0}{2} \bigwedge |g_0[n_0]| < \frac{\epsilon_0}{2} \Rightarrow |f_0[n_0] + g_0[n_0]| < \epsilon_0$.

Now all there is left to do is to combine (31) and (32) such that we have a perfect match for the left–hand side of (35). For this step we select (31) and apply the rule 'Combine with another assumption to a conjunction'. We are prompted to select a second assumption and choose (32). Please mind that the order of the formulae is important, as the inference rules do not take the commutativity of 'And' into account. The resulting formula is now identical to the left–hand side of (35):

We combine (31) and (32) to the conjunction

(36) $|f_0[n_0]| < \frac{\epsilon_0}{2} \bigwedge |g_0[n_0]| < \frac{\epsilon_0}{2}$.

The very last steps are rather routine:

From (36) and (35) we obtain by modus ponens

(37) $|f_0[n_0] + g_0[n_0]| < \epsilon_0$.

Formula (22) is true because it is identical to (37).

The proof is completed, although one branch of the proof tree is still pending:

`Alternative proof 2: pending`

`If we are not able to prove (∈/2 > 0),we continue to prove (8) without.`

```
Pending proof of (8).
```

However, as already explained in the previous example, this is just an alternative branch (resulting from an 'Or' node).

We have demonstrated the most important features of the interactive proof interface now. We may advise the interested reader to repeat these examples and to experiment with them by varying single steps before finally starting with own examples.

3. Conclusion

The goal of this thesis, namely to implement an interface for interactive proving in *Theorema,* was fulfilled to the point that it can be used in a first operational test. This test will be conducted when the system is employed to train students in proving techniques at the Polytechnical University for Software Engineering in Hagenberg, Austria, starting in autumn, 1999.

The Departmentso f Mathematics and Computer Science at the Kepler University in Linz are pondering to follow suit. A small−scale user test is planned for the CALCULEMUS 99 workshop in July this year, where the author will present his work.

The actual implementation has some drawbacks from the software engineering point of view:

- It was not possible to use the existing automated provers as the data structure is insufficient.
- This caused much doubling of code.
- Some work−arounds make the system hard to maintain.

However, we believe that we have shown where the problems lie. A better implementation will require a discussion on a new parameter structure for basic provers in the first place. It must also be seen whether improvements are possible using *Mathematica* 4.0©.

Some features demanded in the original specification still remain to be done:

- An important feature will be the halting of an otherwise automatical proof at 'critical points'.
- It should be possible to check the applicability of inference rules before listing them.
- It should be possible to go back in the proof tree and start again at an arbitrary point.
- It should be possible to combine several basic provers (which is already possible in Theorema in fully−automated proving).

The implementation could also be improved in the following ways:

- The list of inference rules should be presented in a structured way.
- There should be a sort of online help available for the rules.
- The extra rules for special interactive features should maybe go into an extra basic prover.

Another important feature which is still missing is the use of *Theorema*'s (and *Mathematica*' *s*©) evaluational power during a proof (which we missed in the last of the

tutorial examples in this thesis). However, this was not a goal of this thesis. It is one of *Theorema*'s goals for the near future to enable this in the provers in general.

Despite the flaws and missing features listed above, we think that the system is ready for use in proof training for students. We also hope that at least some of the improvements and supplements mentioned above will make it an interesting new feature of *Theorema* also for professional users.

Compared with many other interactive proof systems, it has the advantage that the user is not required to programme anything (as, for instance, in Isabelle, where knowledge of the programming language ML is required for more advanced features [Paulson,1995]). All features can be used by simply pressing buttons and entering mathematical expressions or text when prompted.

Finally, we hope that we have achieved a useful contribution to the comprehensive mathematical software system *Theorema* and provided an interesting tool for students, teachers, researchers, and engineers of mathematics, computer science, software engineering, and related fields.