Advanced Proof Presentation in *Theorema*

Florina PIROI and Tudor JEBELEAN

RISC-Linz
Research Institute for Symbolic Computation
A-4232 Hagenberg, Austria
Florina.Piroi@risc.uni-linz.ac.at

Abstract

The *Theorema* system is implemented in *Mathematica* and aims at providing the working mathematician with a tool for computing/solving/reasoning. In particular, it supports computer–aided reasoning in form of various provers both for predicate logic and for specific domains. The provers of *Theorema* work in "natural style", that is, they produce output which is similar to the proofs written by human mathematicians, and also they use inference rules and heuristics which are similar to the one used by human provers. In order to improve the readability and the accessibility of the *Theorema* provers, we implemented several simplification techniques, which remove redundant or unnecessary information from the generated proofs, and even combine some simple inference steps into more complex ones, that are more intuitive. Moreover, we implement a general interaction mechanism which allows the users to control the proofs during their development and to introduce additional information, as e.g. hints for the instantiation of universal assumptions or of existential goals.

1 Introduction

The *Theorema* project aims at supporting, within one consistent logic and one coherent software system, the entire mathematical exploration cycle including the phase of proving. The *Theorema* system is built upon the computer algebra software *Mathematica* [Wol96], which offers unique facilities for the input/output of logical expressions (including complex graphics), for programming by rewrite rules, and for interactivity. An important application of *Theorema* is computer–aided education of students for using the language of predicate logic and for proof training. While *Theorema* has many different features (see e.g. [Bu-EAl00], www.theorema.org) and includes provers which use various techniques for various special domains, we concentrate in this paper on proving in general predicate logic. In order to make proofs easy to read and easy to interact with, the provers of *Theorema* follow a *natural style* approach: the inferences imitate the natural steps used by human provers, and the rendering of the proofs is done in natural language. This approach has been pioneered by Buchberger, (see e.g. [Bu96]) who designed and implemented the first versions of the predicate logic prover. Proving in natural style gained increased interest recently, on one hand because in many practical applications the (non-specialist) user needs to see an understandable trace of the proof, on the other hand because such a trace makes interaction (for such users) possible and thus allows to approach non–trivial problems. Predicate logic proving in natural style raises new problems which are yet poorly addressed by the research in automatic reasoning, mainly because the established "classical" proving techniques which have the most efficient implementations (resolution, tableaux, connectionist) are not appropriate for natural presentation of proofs. The only acceptable methods are based on sequent calculus, which is difficult to implement due to its high nondeterminism and non–linearity. (For an overview of automatic reasoning methods see [RobVor01].)

Any proving method involves some search in the space of possible solutions, therefore the provers will typically produce some unnecessary information, hence a consistent simplification of the proof is needed before presentation to the user. Moreover, sometimes it is easier to find the proof if one uses some simple, albeit somehow unnatural, inference steps. By simplification we also understand the merging of such steps into more complex, but more intuitive inferences.

Even for powerful proving methods, many proof–problems arising in the current practice of mathematicians are too difficult to solve, mostly because of the large search involved, but also because the impossibility to anticipate in advance all the necessary facts which the formal proof will need as assumptions. In order to improve the effectiveness of the *Theorema* provers, we implement a general mechanism which allows a flexible interaction between the user and the system during the development of the proof. Using this, one can choose between fully automatic (for a predetermined number of steps) and interactive proof–development, one can indicate the inference methods to be used, one can easily navigate through the proof–object, and one can provide various hints (e.g. suitable instantiations) to the prover.

It is difficult to compare the work presented in this paper with existing work, mainly because proving in natural style is a relatively new approach to automatic reasoning (we see this different from what is usually called natural deduction – e.g. based on Gentzen calculus). Because of the variety of inference types used, the proof–tree produced by the provers of *Theorema* is more complex than the tracing information produced by resolution provers and even by tableaux or connectionist methods, therefore proof simplification, proof presentation, and interaction with the provers need a more sophisticated approach. Moreover, due to the general goal of the system of being a working tool for the mathematicians, all these activities are much related to the proper organization of the user interface, which must respond well to the subjective feeling and understanding of the user. Therefore, much of the work presented here has an experimental flavour: one has to try out various types of simplification and interaction schemes in a real environment in order to take the appropriate decisions, and one has to implement the system in a flexible way in order to be able to easily implement the changes suggested by practical experiments with the users (currently, besides the members of the *Theorema* group, we also use as alpha-testers the PhD students enrolled in a special proof–training program at our institute).

2 The Provers of *Theorema*

The *Theorema* system contains various provers for general and specific domains: a propositional and a predicate logic prover [BuJe97][BuEAl00], the Prove–Solve–Compute (PCS) prover for predicate logic with equality [Vasa00][VaPhD00], induction provers over natural numbers and over lists [BuVa97], a set–theory prover [Wind01], a Groebner Bases based prover for boolean combinations of polynomial equalities and inequalities, etc.

The provers operate on "proof–situations" consisting of a set of assumptions and a goal (sequents with only one goal). Each prover is composed of a set of inference rules, each rule is typically expressed as a rewrite rule which transforms a proof–situation into one or more proof–situations.

In order to consistently combine different proving techniques, each prover uses a uniform control mechanism (the "meta–prover") which handles a standard representation of the proof: the "proof–object" [TomuPhD98][BuJeVa98]. The proof–object is a tree representation of the development of the proof: each node corresponds to a proof–situation, while its successors in the tree correspond to the inferred proof–situations. The root of the proof–object corresponds to the initial proof–situation, while the leaves correspond to "proved" (successfully solved), "disproved", "failed" or "pending" (not yet resolved) proof–situations. These "proof–values" are in fact also part of the annotation of the proof–object nodes and their content is computed bottom–up by the meta–prover.

The proof–object contains all the information about the development of the proof: each node is annotated with: a keyword identifying the inference step (e.g. "ModusPonens"), a list of the formulae which have been used for this inference, a list of the resulting formulae, and possibly other information. The representation of this information is standardized across the system, such that certain operations on the proof–object are independent on the particular prover(s) which produced it.

The proof–object is used as an internal representation of the proof. The external representation of the proof (which is displayed to the user in a proof–notebook) is produced by a post–processing of the proof–object, which generates a structured cell representation in a *Mathematica* notebook. The structure of the cells reflects the tree structure of the proof–object, and each inference step is typically represented as a phrase in natural language accompanied by some formulae and referring to some other formulae by their labels (see e.g. Appendix 1, Proof 1).

3 Simplification of Proof–Objects

Proof simplification in *Theorema* is not specific to a particular prover of the system, in the sense that no prover is implied in the simplification process. Proof simplification is a transformer that is applied to the proof–object that is produced by any combination of provers.

The user controls the behavior of the proof–simplifier by specifying certain options. This is consistent with the general philosophy of *Theorema* (and similar to the one of *Mathematica* functions), in which the behavior of the main commands (Prove, Compute, Solve, etc.) is determined by options. (This does not impose an additional burden on the user, because options always have default values which can also be modified by the user.) The options recognized by the proof simplifier refer to branches and steps. Each of these will be detailed in the following.

The simplification options can be classified by their use of informations provided by the provers into two classes: simplification that need no information from the prover that generated the proof (Useful, Lifted, Branch simplification) and simplification that uses certain information provided by the prover that was used (Combined, Essential).

A typical proof call that produces a proof–object is

```
Prove[Proposition["Goal"],
  using → Theory["Assumption"], by → Prover (∗ , other options ∗)]
```

Simplification within *Theorema* may either be invoked at the same time with the **Prove** command, or as a separate **Transform** command. Namely:

```
Prove[Proposition["Goal"], using → Theory["Assumption"],
    by → Prover (∗ , prover options ∗),  transformBy → ProofSimplifier,
    TransformerOptions → {(∗simplification options∗)}];
Transform[proof–object,  by → ProofSimplifier,
  TransformerOptions → {(∗ simplification options ∗)}]
```

Detailed examples of the simplification methods usages explained in the following are illustrated in paragraph 3.3, the proofs may be found in the appendices.

3.1 Branches

A *Theorema* proof–object contains four types of branches, depending on their proof value: *proved*, *disproved*, *failed* and *pending*. A proof built by *Theorema* may contain more than one type of branches, depending on the proving technique that was chosen. If the proof was successful then there are one or more branches with the proof value *proved*, but there might be branches with the proof value *pending* coresponding to the alternatives that were not anymore inspected by the prover after the proof succeeded. In displaying the proof one might be interested in seing only the *proved* branches, or, in case of a failing proof (which does not necessarily mean that the theorem that the user tried to prove is not true) one might want to inspect the *pending* branches. The option Branches indicates which type of branches are retained in the proof–object.

The most common usage of the option, **branches → Proved**, is to remove unsuccessful and pending branches from the proof object. For instance, when the assumptions include a formula of the type $\mathbf{A} \Longrightarrow \mathbf{G}$, where $\mathbf{G}$ is the current goal, the propositional prover will produce two proof alternatives: one is to reduces the proof of $\mathbf{G}$ to proving $\mathbf{A}$ (which may not succeed, but – if successful – produces a more natural proof), and another one is to assume $\neg\mathbf{A}$ (which always succeeds, but is less natural).

3.2 Steps

This option concerns the proof steps (individual inferences) occurring in the proof. Simplifying proof steps means removing, merging, or moving proof steps such that the result of all these action is a shorter, condensed and hopefully easier to understand proof. Usually, these simplifications make sense only if the proof is successful.

• **Useful:** Removes all steps that are not necessary for proving the goal. This option also triggers the simplification **branches → Proved**. and it is done in two phases: **identification** of the useful steps, and **removal** of the un–useful steps. The first phase is done in a bottom–up manner. The proof–object tree is traversed in postorder. As the nodes of the tree are visited, a list that contains labels of the used formulae is built up in the following way:

1. If the node is a leaf node then add the labels of the formulae used and the label of the formula generated (if any) to the list of used formulae–labels.

2. For any other node that has only one son, if the labels of the formula inferred are not contained into the list of used formulae–labels then remove the node from the tree. Otherwise, we remove the label of the formula inferred at the current node from the list of useful formula–labels, and we add the labels of the formulae used for inferring.

3. Nodes that have more than one son are treated similarly with the ones in step 2, the only difference is that we do not delete them if they do not bring useful formulae in the proof.

We note that this kind of simplification checks the assumptions given by the formula as well. If they are not used they are *not* removed from the proof–object since this should be the decision of the user and not of the *Theorema* system. The user will be notified which of the assumptions were not used by printing the labels of these formulae after the proving/displaying has ended.

• **Combined:** Combines simpler steps into more complex ones, by applying a set of rules that are specific to the particular inferences. The rationale for this type of simplification is better explained by an example. For instance, one version of the predicate logic prover uses level–saturation by forward inference on ground formulae combined with universal formulae. If the set of assumptions contains the formulae: P[a], Q[b] and $(\forall x,y)(P[x] \wedge Q[y] \Longrightarrow R[x,y])$, one inference step will produce the intermediary formula $(\forall y)(Q[y] \Longrightarrow R[a,y])$ (which looks less "natural") and another step will produce the final result R[a,b]. By combining the two inferences one obtains in one step the final result (without showing the intermediary formula) which is more human–like. (On the other hand, using such a complex inference rule during the search for the proof may be quite inefficient, especially if we want to use arbitrary many ground formulae.)

One notes that this type of simplification is dependent on the particular prover, because only certain types of inferences can be combined in a meaningful way. Since it is not desirable to write a simplification routine for every particular combination of inferences, we implemented a flexible and universal "combining" routine which takes the specification of such combinations in a standard form from the implementor of the prover. In the example above, the simple inference steps are identified by the keyword "ObtainFromBy": obtain from ground formula ("base") by universal formula ("rule"). The complex inference step is identified by the keyword "ObtainFromManyBy": obtain from many ground bases by a universal rule. The specification of the combination is given as:

```
{{¢ObtainFromBy, {base1_, rule_} → {intermediary_}},
    {¢ObtainFromBy, {base2_, intermediary_} → {final_}}} →
  {¢ObtainFromManyBy, {base1, base2, rule} → {final}}
```

The specification is given as a rewrite rule, having on the left–hand side a list of inference steps (to be combined) and on the right-hand side the resulting inference. Each inference is represented as a pair (*Mathematica* uses curly brackets for denoting tuples or lists) consisting of the keyword corresponding to the inference and another rule which specifies the formulae involved in the inference: the list of used formulae on the left–hand side and the list of resulting formulae on the right–hand side. The underscores occurring on the left–hand side of the main rule qualify the corresponding names as variables (they can represent any formula) – this notation is borrowed from the syntax of *Mathematica* [Wol96]. The underscores are necessary in order to be able to express recursive combination rules, which makes it possible to specify, in this example, the use of an arbitrary number of "bases" in the following way:

```
{{¢ObtainFromManyBy, {bases__, rule_} → {intermediary_}},
    {¢ObtainFromBy, {base_, intermediary_} → {final_}}} →
  {¢ObtainFromManyBy, {bases, base, rule} → {final}}
```

Here, the variable "bases__" (with two underscores) has the meaning "an arbitrary number of formulae" (but at least one) – again this notation is borrowed from *Mathematica*.

The list of simplification rules used in the examples described in paragraph *** and given by the implementor of the prover may look like:

```
{{{¢ExpandAssmByDef, {base1_, def1_} → {rule2_}},
        {¢ObtainFromBy, {base2_, rule2_} → {base3_}}} →
      {¢ObtainFromManyBy, {base1, base2, def1} → {base3}},
   {{¢ObtainFromBy, {base1_, rule1_} → {rule2_}},
        {¢ObtainFromBy, {base2_, rule2_} → {final_}}} →
      {¢ObtainFromManyBy, {base1, base2, rule1} → {final}}};
```

meaning that whenever an assumption expanded by a definition, or a formula inferred at some moment in the proof, is used later to infer another formula we can combine these two steps in one step. The following command will apply the above simplification rules on the proof–object.

• **Lifted:** Will move nodes that satisfy certain conditions upwards in the proof–object tree. In proving a proposition or a theorem the prover might come to the situation where it has to prove a goal which is a conjunction or a disjunction of formulae. The meta–prover, being in charge of building up the proof tree, will create a branching node that will split the goal, i.e. a node that has one son for each formula in the conjunction/disjunction.

The proving process will continue for each of these nodes. In proving these subgoals, the prover invoked may infer formulae that do not involve formulae obtained after the branching node was created. Therefore one might want these formulae to be presented before the splitting of the goal. Using the **Lifted** option in the prove call will trigger this operation. Further more, if same formulae are inferred at nodes on different branches, depending of the formulae they use, the nodes will be removed, and a single node will be inserted in the tree at a higher level in the proof tree. Any information that refers to the formulae generated at the removed nodes will be modified to refer to the information available in the newly inserted node.

3.3 Examples

• For illustrating the **Branches** simplifications we consider the following proposition:

Proposition$\left[\text{"ForAll Distributivity"},\ \left(\underset{x}{\forall}\ (P[x] \Longrightarrow Q)\right) \Longrightarrow \left(\left(\underset{y}{\exists}\ P[y]\right) \Longrightarrow Q\right)\ \ \text{"}\forall\text{d"}\right]$;

The next **Prove** call will generate the proof that can be seen in Appendix 1, Proof 1.

```
Prove[Proposition["ForAll Distributivity"],
  by → PredicateProver, ProverOptions → {BackChaining → True}]
```

When using the following command, only the successful branch of the proof is shown, and the inference step TryAlternatives is removed (Appendix 1, Proof 2)

```
Prove[Proposition["ForAll Distributivity"],
  by → PredicateProver, ProverOptions → {BackChaining → True},
  transformBy → ProofSimplifier, TransformerOptions → {branches → Proved}]
```

• We illustrate the **Lifted** simplification option with the following example.

Theorem$\left[\text{"D"}, \left(\left(\underset{x}{\forall} P[x]\right) \Rightarrow Q\right) \Longrightarrow \left(\underset{x}{\exists} (P[x] \Rightarrow Q)\right)\right]$

```
Prove[Theorem["D"], by → PredicateProver, ProverOptions → {PNDLevel → 0},
  transformBy → ProofSimplifier, TransformerOptions → {steps → Useful}]
```

```
Prove[Theorem["D"], by → PredicateProver, ProverOptions → {PNDLevel → 0},
  transformBy → ProofSimplifier, TransformerOptions → {steps → {Useful, Lifted}}]
```

Appendix 2 presents the proofs obtained for this example. The proof is split on two branches immediately after the formula (3) is obtained, and we observe that formulae (6), (7), (8), (9) on the left branch of the Proof 1 and formulae (10), (11), (12), (13) on the right branch are, respectively the same, therefore in the Proof 2 they are removed and inserted before the proof–step that splits the proving goal.

• The following example will show the effects of the **Useful** simplification. We want to prove that a congruence relation defined by means of ≤ is transitive. The only knowledge we need to prove this is the formal definition of the congruence, and the transitivity of ≤.

Definition$\left[\text{"Congruence Def by LessOrEqual"},\right.$
$\left.\underset{x,y}{\forall} ((x \sim y) :\Leftrightarrow ((x \leq y) \wedge (y \leq x))) \quad \text{"}{\sim}{:}{\leq}{\leq}\text{"}\right];$

Proposition$\left[\text{"LessOrEqual Transitivity"}, \underset{x,y,z}{\forall} (((x \leq y) \wedge (y \leq z)) \Longrightarrow (x \leq z)) \quad \text{"}{\leq}{\leq}{\Rightarrow}{\leq}\text{"}\right];$

Proposition$\left[\text{"Congruence Transitivity"}, \underset{x,y,z}{\forall} (((x \sim y) \wedge (y \sim z)) \Longrightarrow (x \sim z)) \quad \text{"}{\sim}{\sim}{\Rightarrow}{\sim}\text{"}\right];$

The proof calls are:

```
Prove[Proposition["Congruence Transitivity"],
    using → {Definition["Congruence Def by LessOrEqual"],
        Proposition["LessOrEqual Transitivity"]}, by → PredicateProver];
```

The command above will produce a proof–object (see Appendix 3, Proof 1) that contains some steps that do not contribute to the proof thread. Evaluating the following command in *Theorema* will remove these un–useful steps. For the result of this command see Appendix 3, Proof 2.

```
Transform[$TmaProofObject,
    by → ProofSimplifier, TransformerOptions → {steps → Useful}];
```

• for illustrating the **Combined** simplification we remain at the example above.

```
Transform[$TmaProofObject, by → ProofSimplifier,
    TransformerOptions → {steps → {Useful, Combined}}];
```

For the result of this call please inspect Appendix 3, Proof 3. Comparing with Proof 2 one notices that the combined steps involve formulae (5), (6), (7), (20) and (Proposition (LessOrEqual Transitivity): <<⇒<). Formulae (7) and (20) are removed after the simplification took place.

One can find more examples of proofs where the **Combined** simplification would be useful. If, for example, the system is given to prove a universal goal (∀x)P[x]⇒Q[x] the first action taken by the prover will be to instantiate the variable x with an arbitrary fixed value x_0, and generate a new formula P[x_0]⇒Q[x_0] which will be the new goal to be proven. Next, the prover will assume P[x_0] and try to prove Q[x_0]. For each of these actions a node in the proof tree is created, and corresponding text cells are printed in the proof notebook. It is closer to the natural style proving to combine these two steps into one, that instantiates the variable x to x_0 and, assuming P[x_0], will try to prove Q[x_0].

Another case that is often met in the proofs generated by *Theorema* is the case where the last proof–step concludes that the goal is true because it is identical with a certain formula that was generated from the assumptions in a previous proof–step (see Appendix 3, Proof 3, last two steps of the proof). Combining the concluding step with the step that infers the formula identical with the goal will result in proof closer to a natural style proof.

If the prover implementor desires such simplifications to be performed, all he has to do is to write simplification rules for each of the two cases, as described above, and include it in the list of rules that are applied when calling the **Combined**

In Appendix 3, Proof 4 we see the result of the following simplification call, which applies the three step simplifications described in this article.

```
Transform[$TmaProofObject, by → ProofSimplifier,
    TransformerOptions → {steps → {Useful, Combined, Lifted}}];
```

4 Interactive Proving

Normally, a proof call will try to solve the goal by applying the inferences and the heuristics implemented in the prover (or the combination of provers) which is used. In many practical situations it is very difficult to choose automatically the sequence of inferences which produce a successful (or simple) proof, thus leading to a large search in the space of possible solutions. In human proving practice, we usually try various possibilities and rely on our intuition and previous experience in order to find an elegant proof.

One of the advantages of natural–style proving is that the user can easily understand and follow both the inference rules which are available in the prover, as well as the proof situation at any given moment. We capitalize on this advantage by delivering an interaction engine which can be used together with the provers of *Theorema*. When calling a prover in interactive mode, the system displays the proof in a special *proof–window*, and shows additionally a *proof–situation-window*, a *log–window* and several *Mathematica* style *menu–palettes*.

The proof–window displays the current situation of the (possibly unfinished) proof. This is in fact the user–friendly representation of the proof–object, and allows the user to navigate inside the proof–tree in order to continue the proof on a certain branch, introduce new branches, etc.

The proof–situation–window displays the proof situation (current goal and assumptions) associated to the node which is currently selected in the proof–tree. This window gives the user a good overview of the current proof situation, and also allows him to indicate the goal or assumption which is to be manipulated in the next proof step (e.g. for applying a certain inference). Pending (still unresolved) proof obligations are shown with gray background, and the result of the most recent inference is shown in framed cells.

The log–window logs the main commands of the user and displays the errors and the warnings.

The menu–palettes contain various interactive commands and settings, each being represented by a button which triggers the respective action. Some of the palettes are general, some of them are specific to the particular prover(s) which is/are used. Some of the commands require arguments (e.g. Execute needs a Prove command, NewGoal needs a formula). In order to correctly use such a command, the user selects first the cell containing the needed information.

Figures 1 and 2 in Appendix 4 show a screen–shot of a typical situation when proving interactively. In the sequel we present the functions of the menu–palettes.

• **Prove palette:** This palette contains the main commands for controlling the development of the proof. **Execute** starts the execution of the **Prove** command which is currently selected. **Interaction** toggles between automatic and interactive proving. The other buttons are active only in interactive mode. **Next Step** and **Next K Steps** advance the proof on the current branch with 1, resp. K steps chosen automatically by the system. **Finish Execution** tries to finish the proof automatically, and **Stop Execution** aborts the current proof.

• **Proof palette: Focus** sets the current node in the proof–tree to be the one selected in the proof–window. If the user selects a cell containing a formula and then presses **New Goal**, then the system creates an additional branch from the current node, in which the formula is set as the current goal, while on the old branch it is added as an assumption. **Instantiate ∃** works in a similar mode, but the user provides a substitution which is used in order to instantiate the current existential goal for providing a witness. **Instantiate ∀** needs a substitution and the selection of a current assumption (in the proof–situation–window) and adds the appropriate instance to the assumptions. **Add Branch** creates an alternative branch in the proof–tree, while **Remove Branch** and **Remove Assumption** perform the obvious operations, on the condition that the user selects the appropriate elements in the proof–window or in the proof–situation–window.

• **Simplification palette:** Provides "shortcuts" to various simplification operations which are mostly used. The simplification applies only to the displayed version of the proof–object, hence one can re–display all the details using **Full Proof**.

• **Provers palette:** Allows the user to select the domain–specific prover which will be used for the next inferences.

• **Options palette:** This is a prover specific menu (changes dynamically depending on the prover which is currently used) toggles the various options of the respective prover.

• **Inferences palette:** Lists the (nontrivial) inference rules of the current prover. This menu allows the most fine–grained interaction and can have dramatic consequences on the size of the search space.

The implementation of the palettes is done in such a way that it is easy to add new palettes and new buttons, and also to arrange them in various ways on the user–screen. We plan to enhance these menus according to the results of the experiments with the users. In particular, we plan to add a special dynamic palette which is shows suggestions for the next inference step.

References

[Bu96] B. Buchberger. Using Mathematica for Doing Simple Mathematical Proofs. InProceedings of the 4th Mathematica Users' Conference, Tokyo, November 2, 1996, pages 80–96, Wolfram Media Publishing, 1996.

[BuEAl00] B. Buchberger and C. Duprè and T. Jebelean and K. Kriftner and K. Nakagawa and D. Vasaru and W. Windsteiger. The Theorema Project: A Progress Report.In M.Kerber and M.Kohlhase, editor, Proceedings of Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning,A.K.Peters,Natick,Massatchussets, 2000.

[BuJe97] B.Buchberger, T. Jebelean. The Predicate Logic Prover. In Proceedings of *The First International Theorema Workshop*, Hagenberg, Austria, June 1997. RISC report 97-20

[BuJeVa98] B. Buchberger and T. Jebelean and D. Vasaru. *Theorema:* an Integrated System form Computation and Deduction in Natural Style. In Proceedings of CADE 98 (Satellite workshop on integration of computing and proving).

[BuVa97] B. Buchberger and D. Vasaru. An Induction Prover for Equalities over Lists. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria, June 9-10, 1997*, RISC-Reports series No. 97-20, 1997.

[RobVor01] A. Robinson, A. Voronkov (eds.). Handbook of Automated Reasoning. Elesevier, 2001.

[TomuPhD98] E. Tomuta. *An Architecture for Combining Provers and its Applications in the Theorema System*. PhD Thesis, RISC, Johannes-Kepler University, 1998. RISC Report 98-14.

[Vasa00] D. Vasaru Dupre. Automated Theorem Proving by Integrating Proving, Solving, and Computing. (PhD Thesis). RISC-Report 00-19, 2000. http://www.risc.uni-linz.ac.at/library.

[VaPhD00] D.Vasaru-Duprè. Automated Theorem Proving by Integrating Proving, Solving and Computing. The Research Institute for Symbolic Computation, Johannes Kepler University, May 2000. RISC report 00-19.

[Wind01] W. Windsteiger. A Set Theory Prover in Theorema: Implementation and Practical Applications. The Research Institute for Symbolic Computation, Johannes Kepler University, May 2001. RISC report 01-07.

[Wol96] S.Wolfram.The Mathematica Book, Wolfram Media and Cambridge University Press, 1996.

Appendices

Appendix 1

Proof 1.

Prove:

(Proposition (ForAll Distributivity): ∀d) $\underset{x}{\forall} (P[x] \Rightarrow Q) \Rightarrow \left(\underset{y}{\exists} P[y] \Rightarrow Q\right)$,

with no assumptions.

We prove (Proposition (ForAll Distributivity): ∀d) by the deduction rule.

We assume

(1) $\underset{x}{\forall} (P[x] \Rightarrow Q)$

and show

(2) $\underset{y}{\exists} P[y] \Rightarrow Q$.

Formula (1) is simplified to:

(3) $\underset{x}{\exists} P[x] \Rightarrow Q$.

We prove (2) by the deduction rule.

We assume

(4) $\underset{y}{\exists} P[y]$

and show

(5) Q.

By (4) we can take appropriate values such that:

(6) $P[y_0]$.

Because the conclusion of (3) is identical to the goal (5), we try two alternatives:

Alternative proof 1: proved

For proving the goal, by (3), it suffices to prove

(8) $\underset{x}{\exists} P[x]$.

Formula (8) is proved because (6) is an instance of it.

Alternative proof 2: pending

Because the goal (5)is identical to the conclusion of (3), we transform it into:

(7) $\underset{x}{\nexists} P[x]$.

Pending proof of (5).

Proof 2.

The proof obtained by using the option **branches→Proved** is identical to Proof 1 above until the formula (6), but after that it shows only the successful branch:

By (4) we can take appropriate values such that:

(6) $\mathrm{P}[y_0]$.

For proving the goal, by (3), it suffices to prove

(8) $\underset{x}{\exists}\, \mathrm{P}[x]$.

Formula (8) is proved because (6) is an instance of it.

Appendix 2

Proof 1.

Prove:

(Theorem (D)) $\left(\underset{x}{\forall}\, \mathrm{P}[x] \Rightarrow \mathrm{Q}\right) \Rightarrow \underset{x}{\exists}\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$,

with no assumptions.

We prove (Theorem (D)) by the deduction rule.

We assume

(1) $\underset{x}{\forall}\, \mathrm{P}[x] \Rightarrow \mathrm{Q}$

and show

(2) $\underset{x}{\exists}\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$.

Formula (1) is transformed into:

(3) $\left(\neg \left(\underset{x}{\forall}\, \mathrm{P}[x]\right)\right) \bigvee \mathrm{Q}$.

We prove (2) by case distinction using (3).

Case (3.1) $\neg \left(\underset{x}{\forall}\, P[x]\right)$:

Formula (3.1) is simplified to:

(4) $\underset{x}{\exists}\, \neg\, (\mathrm{P}[x])$.

By (4) we can take appropriate values such that:

(5) $\neg\, (\mathrm{P}[x_0])$.

We prove (2) by contradiction.

We assume

(6) $\underset{x}{\nexists}\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$,

and show *a* contradiction.

Formula (6) is simplified to:

(7) $\underset{x}{\forall}\, \neg\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$.

Formula (7) is simplified to:

(8) $\underset{x}{\forall}(\mathrm{P}[x] \wedge (\neg\, \mathrm{Q}))$.

Formula (8) is simplified to:

(9) $\underset{x}{\forall}\, \mathrm{P}[x] \bigwedge (\neg\, \mathrm{Q})$.

Formula (a contradiction) is proved because (5) and (9.1) are contradictory.

Case (3.2) Q:

We prove (2) by contradiction.

We assume

(10) $\underset{x}{\nexists}(\mathrm{P}[x] \Rightarrow \mathrm{Q})$,

and show *a* contradiction.

Formula (10) is simplified to:

(11) $\underset{x}{\forall} \neg\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$.

Formula (11) is simplified to:

(12) $\underset{x}{\forall}(\mathrm{P}[x] \wedge (\neg\, \mathrm{Q}))$.

Formula (12) is simplified to:

(13) $\underset{x}{\forall}\, \mathrm{P}[x] \bigwedge (\neg\, \mathrm{Q})$.

Formula (a contradiction) is proved because (13.2) and (3.2) are contradictory.

□

Proof 2.

Prove:

(Theorem (D)) $\left(\underset{x}{\forall}\, \mathrm{P}[x] \Rightarrow \mathrm{Q}\right) \Rightarrow \underset{x}{\exists}(\mathrm{P}[x] \Rightarrow \mathrm{Q})$,

with no assumptions.

We prove (Theorem (D)) by the deduction rule.

We assume

(1) $\underset{x}{\forall}\, \mathrm{P}[x] \Rightarrow \mathrm{Q}$

and show

(2) $\underset{x}{\exists}(\mathrm{P}[x] \Rightarrow \mathrm{Q})$.

We prove (2) by contradiction.

We assume

(6) $\underset{x}{\nexists}(\mathrm{P}[x] \Rightarrow \mathrm{Q})$,

and show *a* contradiction.

Formula (6) is simplified to:

(7) $\underset{x}{\forall} \neg\, (\mathrm{P}[x] \Rightarrow \mathrm{Q})$.

Formula (7) is simplified to:

(8) $\underset{x}{\forall}(\mathrm{P}[x] \wedge (\neg\, \mathrm{Q}))$.

Formula (8) is simplified to:

(9) $\underset{x}{\forall} \mathrm{P}[x] \bigwedge (\neg \mathrm{Q})$.

Formula (1) is transformed into:

(3) $\left(\neg \left(\underset{x}{\forall} \mathrm{P}[x]\right)\right) \bigvee \mathrm{Q}$.

We prove (a contradiction) by case distinction using (3).

Case (3.1) $\neg \left(\underset{x}{\forall} P[x]\right)$:

Formula (3.1) is simplified to:

(4) $\underset{x}{\exists} \neg (\mathrm{P}[x])$.

By (4) we can take appropriate values such that:

(5) $\neg (\mathrm{P}[x_0])$.

Formula (a contradiction) is proved because (5) and (9.1) are contradictory.

Case (3.2) Q:

Formula (a contradiction) is proved because (9.2) and (3.2) are contradictory.

Appendix 3

Proof 1.

Prove:

(Proposition (Congruence Transitivity): ~~⇒~) $\underset{x,y,z}{\forall} (x \sim y \wedge y \sim z \Rightarrow x \sim z)$,

under the assumptions:

(Definition (Congruence Def by LessOrEqual): ~:≤≤) $\underset{x,y}{\forall} (x \sim y \;:\Leftrightarrow\; x \leq y \wedge y \leq x)$,

(Proposition (LessOrEqual Transitivity): ≤≤⇒≤) $\underset{x,y,z}{\forall} (x \leq y \wedge y \leq z \Rightarrow x \leq z)$.

For proving (Proposition (Congruence Transitivity): ~~⇒~) we take all variables arbitrary but fixed and prove:

(1) $x_0 \sim y_0 \wedge y_0 \sim z_0 \Rightarrow x_0 \sim z_0$.

We prove (1) by the deduction rule.

We assume

(2) $x_0 \sim y_0 \wedge y_0 \sim z_0$

and show

(3) $x_0 \sim z_0$.

For proving the current goal, by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤), it suffices to prove:

(4) $x_0 \leq z_0 \wedge z_0 \leq x_0$.

The formula (2.1) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(5) $x_0 \leq y_0 \wedge y_0 \leq x_0$.

The formula (2.2) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(6) $y_0 \leq z_0 \wedge z_0 \leq y_0$.

We prove the individual conjunctive parts of (4):

Proof of (4.1) $x_0 \leq z_0$:

From (6.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(10) $\underset{z}{\forall} (y_0 \leq z \Rightarrow z_0 \leq z)$.

From (6.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(9) $\underset{z}{\forall} (z_0 \leq z \Rightarrow y_0 \leq z)$.

From (5.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(8) $\underset{z}{\forall} (x_0 \leq z \Rightarrow y_0 \leq z)$.

From (5.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(7) $\underset{z}{\forall} (y_0 \leq z \Rightarrow x_0 \leq z)$.

From (6.1), by (10), we obtain:

(16) $z_0 \leq z_0$.

From (5.2), by (10), we obtain:

(15) $z_0 \leq x_0$.

From (6.2), by (9), we obtain:

(14) $y_0 \leq y_0$.

From (5.1), by (8), we obtain:

(13) $y_0 \leq y_0$.

From (6.1), by (7), we obtain:

(12) $x_0 \leq z_0$.

From (5.2), by (7), we obtain:

(11) $x_0 \leq x_0$.

Formula (4.1) is true because it is identical to (12).

Proof of (4.2) $z_0 \leq x_0$:

From (4.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(21) $\underset{z}{\forall} (z_0 \leq z \Rightarrow x_0 \leq z)$.

From (6.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(20) $\underset{z}{\forall} (y_0 \leq z \Rightarrow z_0 \leq z)$.

From (6.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(19) $\underset{z}{\forall} (z_0 \leq z \Rightarrow y_0 \leq z)$.

From (5.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(18) $\underset{z}{\forall} (x_0 \leq z \Rightarrow y_0 \leq z)$.

From (5.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(17) $\underset{z}{\forall}\,(y_0 \leq z \Rightarrow x_0 \leq z)$.

From (6.2), by (21), we obtain:

(29) $x_0 \leq y_0$.

From (6.1), by (20), we obtain:

(28) $z_0 \leq z_0$.

From (5.2), by (20), we obtain:

(27) $z_0 \leq x_0$.

From (6.2), by (19), we obtain:

(26) $y_0 \leq y_0$.

From (4.1), by (18), we obtain:

(25) $y_0 \leq z_0$.

From (5.1), by (18), we obtain:

(24) $y_0 \leq y_0$.

From (6.1), by (17), we obtain:

(23) $x_0 \leq z_0$.

From (5.2), by (17), we obtain:

(22) $x_0 \leq x_0$.

Formula (4.2) is true because it is identical to (27).

□

Proof 2.

Prove:

(Proposition (Congruence Transitivity): ~~⇒~) $\underset{x,y,z}{\forall}\,(x \sim y \wedge y \sim z \Rightarrow x \sim z)$,

under the assumptions:

(Definition (Congruence Def by LessOrEqual): ~:≤≤) $\underset{x,y}{\forall}\,(x \sim y \;:\Leftrightarrow\; x \leq y \wedge y \leq x)$,

(Proposition (LessOrEqual Transitivity): ≤≤⇒≤) $\underset{x,y,z}{\forall}\,(x \leq y \wedge y \leq z \Rightarrow x \leq z)$.

For proving (Proposition (Congruence Transitivity): ~~⇒~) we take all variables arbitrary but fixed and prove:

(1) $x_0 \sim y_0 \wedge y_0 \sim z_0 \Rightarrow x_0 \sim z_0$.

We prove (1) by the deduction rule.

We assume

(2) $x_0 \sim y_0 \wedge y_0 \sim z_0$

and show

(3) $x_0 \sim z_0$.

For proving the current goal, by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤), it suffices to prove:

(4) $x_0 \leq z_0 \wedge z_0 \leq x_0$.

The formula (2.1) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(5) $x_0 \leq y_0 \wedge y_0 \leq x_0$.

The formula (2.2) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(6) $y_0 \leq z_0 \wedge z_0 \leq y_0$.

We prove the individual conjunctive parts of (4):

Proof of (4.1) $x_0 \leq z_0$:

From (5.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(7) $\underset{z}{\forall} (y_0 \leq z \Rightarrow x_0 \leq z)$.

From (6.1), by (7), we obtain:

(12) $x_0 \leq z_0$.

Formula (4.1) is true because it is identical to (12).

Proof of (4.2) $z_0 \leq x_0$:

From (6.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(20) $\underset{z}{\forall} (y_0 \leq z \Rightarrow z_0 \leq z)$.

From (5.2), by (20), we obtain:

(27) $z_0 \leq x_0$.

Formula (4.2) is true because it is identical to (27).

□

Proof 3.

Prove:

(Proposition (Congruence Transitivity): ~~⇒~) $\underset{x,y,z}{\forall}(x \sim y \wedge y \sim z \Rightarrow x \sim z)$,

under the assumptions:

(Definition (Congruence Def by LessOrEqual): ~:≤≤) $\underset{x,y}{\forall}(x \sim y :\Leftrightarrow x \leq y \wedge y \leq x)$,

(Proposition (LessOrEqual Transitivity): ≤≤⇒≤) $\underset{x,y,z}{\forall}(x \leq y \wedge y \leq z \Rightarrow x \leq z)$.

For proving (Proposition (Congruence Transitivity): ~~⇒~) we take all variables arbitrary but fixed and prove:

(1) $x_0 \sim y_0 \wedge y_0 \sim z_0 \Rightarrow x_0 \sim z_0$.

We prove (1) by the deduction rule.

We assume

(2) $x_0 \sim y_0 \wedge y_0 \sim z_0$

and show

(3) $x_0 \sim z_0$.

For proving the current goal, by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤), it suffices to prove:

(4) $x_0 \leq z_0 \wedge z_0 \leq x_0$.

The formula (2.1) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(5) $x_0 \leq y_0 \wedge y_0 \leq x_0$.

The formula (2.2) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(6) $y_0 \leq z_0 \wedge z_0 \leq y_0$.

We prove the individual conjunctive parts of (4):

Proof of (4.1) $x_0 \leq z_0$:

From (5.1) and (6.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(12) $x_0 \leq z_0$.

Formula (4.1) is true because it is identical to (12).

Proof of (4.2) $z_0 \leq x_0$:

From (6.2) and (5.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(27) $z_0 \leq x_0$.

Formula (4.2) is true because it is identical to (27).

□

Proof 4.

Prove:

(Proposition (Congruence Transitivity): ~~⇒~) $\underset{x,y,z}{\forall}(x\sim y\land y\sim z\Rightarrow x\sim z)$,

under the assumptions:

(Definition (Congruence Def by LessOrEqual): ~:≤≤) $\underset{x,y}{\forall}(x\sim y :\Leftrightarrow x\leq y\land y\leq x)$,

(Proposition (LessOrEqual Transitivity): ≤≤⇒≤) $\underset{x,y,z}{\forall}(x\leq y\land y\leq z\Rightarrow x\leq z)$.

For proving (Proposition (Congruence Transitivity): ~~⇒~) we take all variables arbitrary but fixed and prove:

(1) $x_0\sim y_0\land y_0\sim z_0\Rightarrow x_0\sim z_0$.

We prove (1) by the deduction rule.

We assume

(2) $x_0\sim y_0\land y_0\sim z_0$

and show

(3) $x_0\sim z_0$.

For proving the current goal, by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤), it suffices to prove:

(4) $x_0\leq z_0\land z_0\leq x_0$.

The formula (2.1) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(5) $x_0\leq y_0\land y_0\leq x_0$.

The formula (2.2) is expanded by the definition (Definition (Congruence Def by LessOrEqual): ~:≤≤) into:

(6) $y_0\leq z_0\land z_0\leq y_0$.

From (5.1) and (6.1), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(12) $x_0\leq z_0$.

From (6.2) and (5.2), by (Proposition (LessOrEqual Transitivity): ≤≤⇒≤), we obtain:

(27) $z_0\leq x_0$.

We prove the individual conjunctive parts of (4):

Proof of (4.1) $x_0\leq z_0$:

Formula (4.1) is true because it is identical to (12).

Proof of (4.2) $z_0\leq x_0$:

Formula (4.2) is true because it is identical to (27).

□

Appendix 4

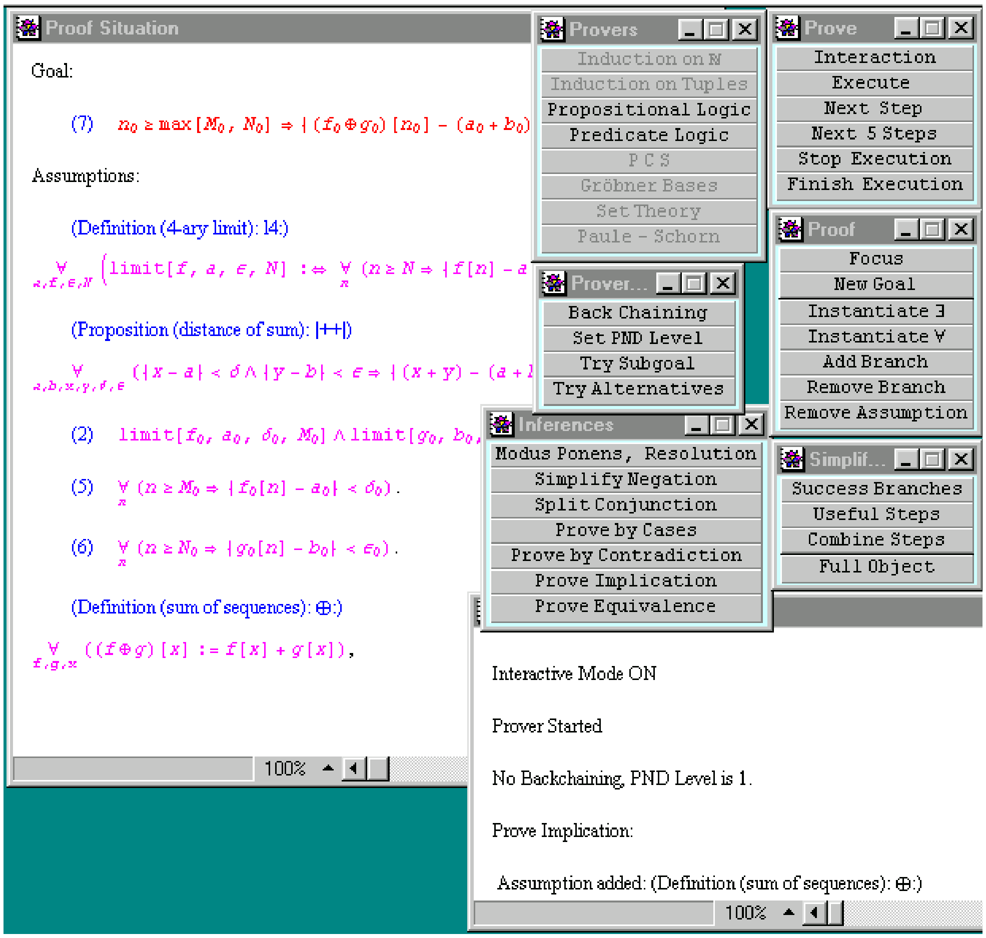

Figure 1

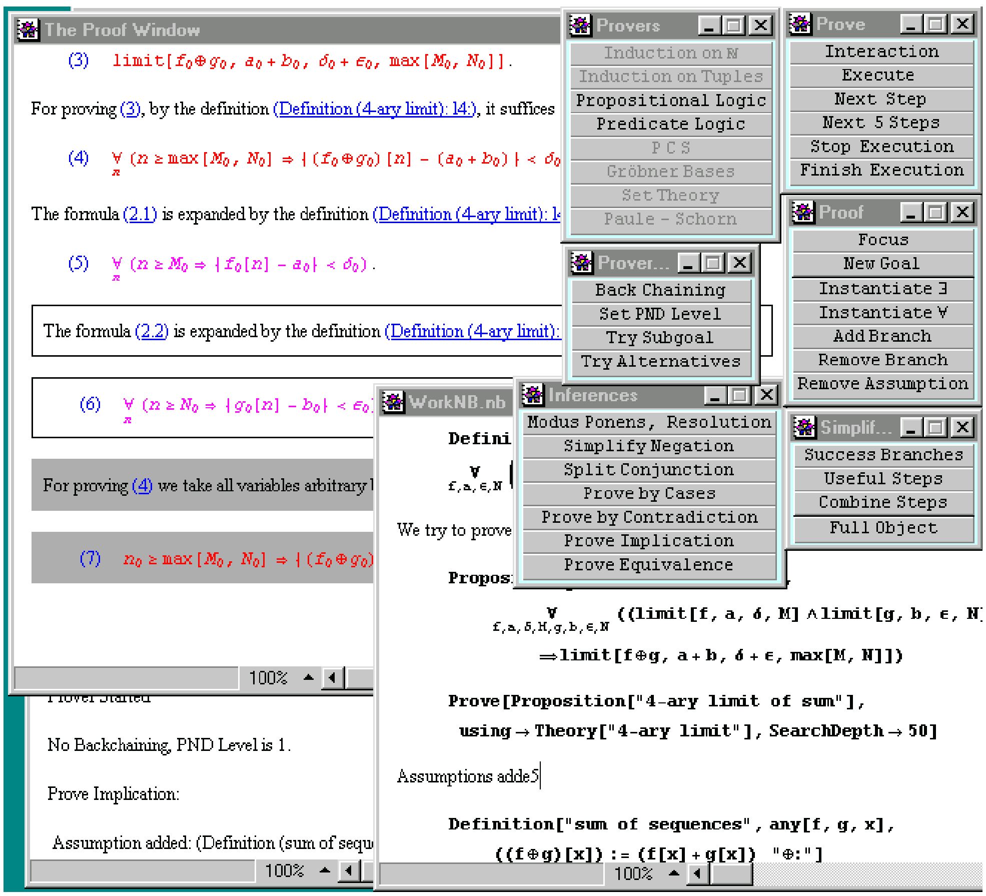
The Proof Window
For proving (3), by the definition (Definition (4-ary limit): l4:), it suffices
The formula (2.1) is expanded by the definition (Definition (4-ary limit): l
The formula (2.2) is expanded by the definition (Definition (4-ary limit):
For proving (4) we take all variables arbitrary
No Backchaining, PND Level is 1.
Prove Implication:
Provers
Induction on N
Induction on Tuples
Propositional Logic
Predicate Logic
P C S
Gröbner Bases
Set Theory
Paule - Schorn
Prove
Interaction
Execute
Next Step
Next 5 Steps
Stop Execution
Finish Execution
Proof
Focus
New Goal
Instantiate ∃
Instantiate ∀
Add Branch
Remove Branch
Remove Assumption
Prover...
Back Chaining
Set PND Level
Try Subgoal
Try Alternatives
Inferences
Modus Ponens, Resolution
Simplify Negation
Split Conjunction
Prove by Cases
Prove by Contradiction
Prove Implication
Prove Equivalence
Simplif...
Success Branches
Useful Steps
Combine Steps
Full Object
WorkNB.nb
We try to prove
Prove[Proposition["4-ary limit of sum"],
using → Theory["4-ary limit"], SearchDepth → 50]
Definition["sum of sequences", any[f, g, x],
((f⊕g)[x]) := (f[x] + g[x]) "⊕:"]

Figure 2