ICOT Technical Report: TR-355

TR-355

Toward Mechanization of Mathematics

by
K. Sakai

March, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Toward Mechanization of Mathematics
— Proof Checker and Term Rewriting System —

Kô Sakai

ICOT Research Center
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

ABSTRACT

This paper describes a proof checker and a term rewriting system generator developed by ICOT in consultation with its working groups. Currently in ICOT, a new programming environment is designed as a typical application of these system. The proof checker supports us in inference on logical connectives and the term rewriting system generator in inference on equality for verifying program correctness in such a new environment.

1. Background

1.1 Programming Environment

Quite some time has passed since we began to be warned of the software crisis. A lot of ideas have been suggested to overcome this crisis and a lot of effort has been devoted to research in software engineering and mathematical theory of programs. However, the crisis seems to be getting more and more serious.

One of the main reasons for this situation lies in the great variety of programming styles. The area covered by a single approach is quite restricted, even if it is very effective. Depending on applications, characteristics of programs differ remarkably and, therefore, so does the knowledge required for programming. Moreover, programming in the

broad sense consists of a lot of stages such as analysis, design, implementation, documentation, and maintenance. The knowledge required in each stage is different.

Operating systems (OS) are the first programming aids available to ordinary programmers. Since an OS is aiming at general users of a computer system it has a highly flexible design. The editor is the most successful of the utilities an OS is equipped with, in the sense that it is designed as a general tool addressing a common activity of computer users. The secret of this success is that attention is devoted to the activity of making text without reference to applications or the stages of programming mentioned above.

However, OSs viewed as programming environments have a lot of shortcomings because of their generality; there is little room for incorporating tools especially useful to programming.

1.2 Correctness of Programs

An ideal programming system would help in making correct programs rapidly. The essence of such a system is the measure for program correctness. A naïve method conventionally employed to make sure that a program is correct is to run the program and test whether it produces the expected result. Though this method is very simple and general, it is far from efficient or reliable.

Some compilers and interpreters are equipped with debugging facilities such as tracers to make debugging more efficient. However, ordinarily, debugging by these methods is not called "program verification", since they provide no guarantee of the universal correctness of a program even if it is tested hundreds of times on different data.

The term "program verification" usually means to verify the correctness of a program statically by logical inference instead of running it. In this case, what a program has to do is represented separately. We call it the specification of the program. Whatever verification methods are adopted, the correctness of a program requires that it behaves in the

way the programmer intended. In this sense specification is the starting point for discussing correctness of programs. Some researchers claim that specification or verification is useless. Of course, if specification were more difficult than direct implementation, it would be no help for us in writing programs. Programming in the broad sense also includes analysis and design as we mentioned before, so correctness of programs involves a lot of factors. Indeed, the problem as stated above merely means correctness of implementations.

Correctness of algorithms on an abstract level is much more important, though in most actual cases it is checked manually (usually by mathematical methods). A computer-aided system to verify correctness is even more important during algorithm design, since debugging tools like tracers are fairly effective in the implementation stage. The most essential part of programming is the process of reducing abstract relations between input and output to detailed procedures with concrete data structures. The programming system would be of little use if it supported us only in the process of generating actual programs from the detailed procedures.

1.3 Programming and Theorem Proving

When we consider programming in the broad sense, we find it very similar to problem solving in mathematics. This similarity is illustrated in Fig.1.

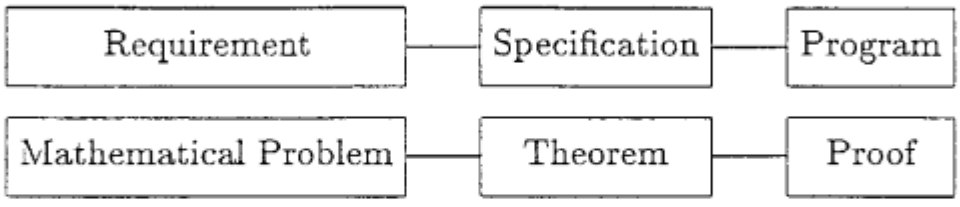


Fig.1 Correspondence between Programming and Theorem Proving

Programming begins with requirements; what we want computers to do. At first such requirements are seldom rigid. A lot of conditions are gradually clarified in the design process, and the details are often left to the common sense of the programmer. The first step in programming is

to convert vague requirements into rigid specifications. Abstract data structures are usually determined at this stage. Ease of programming depends to a great extent on the data structures. We are very often obliged to change the data structure because of serious problems that arise during implementation. Several recently developed programming languages provide us with abstract data types to represent data at this abstract level naturally.

The stage described above exactly corresponds to the stage in mathematics where a given problem is formalized and converted to a theorem (or a conjecture). We need formal definitions for many concepts to write down a theorem (or a conjecture), which correspond to abstract data structures in programming. Ease of proof very much depends on the definitions and we often change the definitions when we find that proof is difficult.

The biggest difference between programming and theorem proving is that programs as final products are most important in programming, while theorems as basically intermediate products seem much more important than proofs in mathematics. However, this difference is not essential. In fact, a program whose specification we did not know would be useless, and a theorem (or a conjecture in the precise sense) without proof could not be acknowledged as true.

The above similarity suggests that technologies cultivated for programming can be appropriated for theorem proving and vice versa. Based on this observation, we implemented a proof checker for linear algebra and a term rewriting system generator, aiming first at mathematical activity such as theorem proving, but then at programming. We believe that this approach will make significant contributions to the fifth generation computer system (especially to the intelligent programming system), since mathematics has a history of more than 2,000 years, whereas computer science is only 40 years old.

2. Proof Checker

2.1 An overview

Automated theorem proving, or automated deduction, has undergone a quarter of a century of research and development and is today one of the oldest areas in artificial intelligence. Many interesting theorem provers and proof checkers have been implemented [Gorden 78, Ketonen 83, Trybulec 85, Shanker 85].

Most of them are devoted to only "formal" proofs without gaps. When proofs to be checked are not stated "formally", the "informality" causes different kinds of problems not encountered in the systems dedicated to formal proofs, and hence deserves serious investigation which has yet to be undertaken. A proof checker for informally stated proofs has to fill possible gaps of inference occurring between proof steps. There are very often wide gaps. For instance, if a given proof consists of only a conclusion, the proof checker has to have virtually the same capability as a completely automated theorem prover to check it. In addition, the proof checker should have some facilities for acquiring knowledge of proofs and theorems to achieve user-friendly and efficient proof checking.

A proof checker for informal proofs need many functions such as automated reasoning and knowledge base functions for mathematical theories. There may be cases in which a certain mechanism for inductive inference, or inductive learning, based on the knowledge base is necessary in addition to that for deductive inference. Hence, it is extremely difficult to construct a practical and powerful theorem prover or proof checker, where the word "practical" means to handle informal proofs. Continuing efforts will be required to resolve the difficulties facing us in making a proof checker that is a good assistant or an efficient aid for education.

The perspectives of knowledge information processing will provide a powerful approach to such a proof checker, since it will be considered as an inference and computation (or ratiocination) system coupled with a

certain knowledge base. This approach will play a fundamental role in that it will:

(1) Give cues to formalization of inductive theory.

(2) Be able to clarify relations between various propositions and methods.

(3) Help to clarify mutual relations between mathematical theories. and

(4) Provide a new approach to metamathematics, that is, a metatheory not confined to a single particular mathematical theory, but a theory about multiple correlated mathematical theories.

2.2 the CAP project and the CAP-LA system

The final goal of the computer-aided proof (CAP) project is to construct a general proof checker incorporating a large amount of knowledge common to working mathematicians, with various utilities such as proof editor, pretty printer (with two-dimensional display), and symbolic manipulator of mathematical formulas.

As a matter of fact it took a lot of effort to implement even a simple proof checker. A formal language had to be designed for use in writing theorems and proofs. A lot of facilities for intelligent handling of mathematical proofs were implemented; proof editing facilities for mathematical text with complex structures; proof checking facilities to verify correctness of proofs; and knowledge base management facilities to store and retrieve, and maintain the consistency of, many definitions, theorems and proofs. Many experiments on computer-assistance for solving mathematical problems are now underway to develop an ideal interface for man-machine collaboration on such activities.

The first prototype of our proof checker is dedicated to linear algebla, the most familiar branch of college mathematics. We selected a textbook for a freshman course and designed a formal language for writing all theorems and their proofs in the book. The language and axiom system is based on Gentzen's natural deduction system (NK) with some

additional inference rule. This proof checker is named CAP-LA and the language is named PDL (Proof Description Language).

The first version of CAP-LA consists of the following four main modules:

(1) System controller,

(2) Proof editor,

(3) Proof checker, and

(4) Knowledge base manager.

The system controller controls the other modules of the system and provides a man-machine interface. The proof editor is a structured editor dedicated to mathematical proof and has special knowledge about the syntax of PDL. It includes a browser editor. The proof checker is a kernel module of the system. It controls general knowledge of logical inference and special knowledge of linear algebra and checks whether a given proof is correct or not. The checker also contains a term rewriting system to check equality of terms. The knowledge base manager contains various kinds of systematically organized knowledge such as definitions, theorems and proofs (checked or not) in the form of terms. It retrieves information necessary for proof checking, inserts new inference rules for following proof steps upon completion of checking, and checks the consistency of the stored knowledge. A sample session with the CAP-LA system is given in Appendix 3.

2.3 PDL

PDL was designed for the CAP-LA system and over 90 percent of theorems and their proofs in the text book mentioned above have been written in it.

PDL will play a central role in the man-machine interaction. PDL should facilitate reading and writing every mathematical proof, and at the same time meet the requirement that every written proof in it be convertible into machine-readable format for further use in proof checking. Moreover, PDL requires rich expressive power sufficient to

represent various kinds of mathematical proofs. We selected Gentzen's natural deduction system (NK) as its basis, since it seems to reflect the process of human intelligence intrinsically, and we preferred to have logical completeness for our system. Appendix 1 lists inference rules in NK with their counterparts in PDL. Many inference rules not in the original NK are necessary to develop mathematical theories, and each of them also has its counterpart in PDL.

The basic policies underlining the first version of PDL are as follows:

(1) The syntax of PDL includes the proof templates corresponding to the inference rules of NK.

(2) It should be a strongly typed system in which all elements are typed. The user should be able to define new types with parameters. Types are handled in the same way as other logical formulas.

(3) Constants (such as integers and 0-vector), function symbols, and variables are not distinguished syntactically. All the bound variables must be bound explicitly by a quantifier. The user should be able to define new constants and function symbols. Constants and free variables are logically handled in the same way.

(4) The user should be able to use conjunctions, adjectives, and adverbs naturally in a proof.

(5) The user should be able to specify the inference to be applied at each proof step or to skip checking.

(6) The system contains some special knowledge about linear algebra. Vectors are to be considered as one-column or one-row matrices.

The syntax of PDL will not be detailed here, since it is very easy to understand. Appendix 2 shows simple theorems and their proofs described in PDL.

2.4 Proof checking subsystem

The proof checking subsystem is the kernel of the CAP-LA system. Proof texts input via the proof editor are converted into proof trees for proof checking. The proof trees are examined by testing whether each step obeys the inference rules. If the proof includes logical equivalences or equalities, they are checked by the term rewriting technique, that is, whether both sides of the equality in the formula can be rewritten into the same term. Term rewriting is investigated by another system, which we will discuss later in detail. The checker also has specific knowledge on finite summation $\sum$, finite product $\prod$, and so on. Once a theorem has been proved it is converted to a rule ready for use in the proofs of succeeding theorems, and registered in the theory knowledge base.

In proof checking, backward reasoning is mainly performed at each inference step. For example, given an inference step

$$\frac{\begin{array}{ccc}\vdots & \vdots & \vdots \\ D & E & F\end{array}}{G},$$

it is first checked whether or not there exists a strategy to prove G when D, E, and F are proved. If not, it is checked whether G is verified or not by applying rules in the environment where D, E, and F are proved. After checking G, the system inserts it in the knowledge base and creates the new environment for checking the next inference step. Control is transferred to the term rewriting module when a formula with equality must be checked. After retrieving possible rewriting rules from the knowledge base, the term rewriting module tries to apply the rules to the terms on both sides of the equality sign, and tests whether the two terms become equal by this reduction.

2.5 Further plans for CAP projects

We are currently experimenting with a new system, which should be naturally called CAP-QJ, to describe some metamathematical theorems and proofs in QJ [Sato 85, Sato 84]. A compiler and an interpreter for the programming language Quty are scheduled to be implemented. This

new checker is closely related to the intelligent programming system we mentiond in the previous section, which aims to derive programs from proofs. Given a constructive proof for a formal specification written in QJ. a realizability interpreter derives a realizer as an executable part of the proof. that is, a program. The realizability interpreter is also now under experimentation.

We recognized from the experience of the first version CAP-LA that PDL can be used to express not only linear algebra but many branches of mathematics. We also found that the performance of proof checking is efficient. Design of the second version of CAP-LA has already begun with the followings features:

2.5.1 PDL

(1) Extension to higher order logic.

(2) Generic types or type variables, with which users can define more natural type hierarchies.

(3) User-defined proof templates and extraction of proof templates from checked proofs.

(4) Suppression of long, repetitious, and tedious proof by admitting "similarly", "..." and so on in a proof.

(5) A lot of syntax sugar so that a user can write theorems and proofs easily.

2.5.2 Proof Checking

(1) Combination of forward and backward reasoning.

(2) Type inference and type checking for proof checking.

(3) Utilizing conjunctions, adjectives, and adverbs in a proof to improve efficiency of proof checking.

(4) Checking higher order theorems or metamathematical theorems.

2.5.3 User Interface

(1) Two-dimensional display of $\sum$, $\prod$, matrices, etc., using a bit-map display.

(2) Fully interactive proof checking.

(3) Stepwise refinement of a proof by collaboration with the system.

(4) Detailed error messages and help messages for the novice user, and more natural English and Japanese proof forms.

3. Term Rewriting System Generator (Metis)

Manipulation of formulas is the most onerous part of mathematical activity by ordinary researchers. In general, it includes deep and complicated inference processes involving equality, "=". We implemented a system called Metis as an efficient support tool for this kind of inference.

A set of rewrite rules is called a term rewriting system or TRS. The theory of TRSs has a wide variety of both theoretical and practical applications. It provides models for abstract data types, operational semantics for functional programming languages, and inference engines for automated theorem proving with equality.

A TRS is called complete if it is terminating and confluent. Metis generates a complete TRS from a set of equations automatically, semi-automatically, or interactively. It is also an experimental tool with the various functions needed for the study of TRSs.

3.1 Preliminaries

We assume the reader is familiar with the following concepts [Huet 80]:

(1) *Terms, ground terms, occurrences, subterms, substitutions, unifiers,* and *most general unifiers.*

(2) *Reduction* (or *rewriting*) by a TRS. *Termination* and *confluence* of a TRS. *Irreducible forms* of a term with respect to a TRS.

(3) *Equational theories* and *word problems.*

(4) *Superpositions, critical pairs* and the *Knuth-Bendix completion procedure* [Knuth 70, Huet 81].

In what follows, we will denote the set of function symbols by F, the set of variables by V, the set of all terms constructed from F and V by $\mathcal{T}(F, V)$, and the set of all the ground terms constructed from F by $\mathcal{T}(F)$. The notation $c[s]$ denotes a term with s as its subterm. In this notation, $c[]$ represents the context where the subterm s occurs. Therefore, $c[s']$ denotes the term obtained by replacing the occurrence of s in $c[s]$ with s'. Similarly, we will use the notation $c[s_1, \ldots, s_n]$ to represent a term with $s_1, \ldots, s_n$ subterms, and $c[s'_1, \ldots, s'_n]$ for the term obtained by replacing each s_i in $c[s_1, \ldots, s_n]$ with s'_i. Substitutions are denoted by the greek letter θ, possibly with subscripts and primes.

One step of reduction is denoted by $\Rightarrow$ and the reflexive transitive closure of $\Rightarrow$, i.e. several steps of reduction (possibly no steps), by $\stackrel{*}{\Rightarrow}$. An irreducible form of t is denoted by $t{\downarrow}$. If R is a terminating TRS, then every term t has an irreducible form $t{\downarrow}$. Moreover, R is confluent if and only if the irreducible form $t{\downarrow}$ is unique. In this case, the TRS R is said to be *complete* and the irreducible form $t{\downarrow}$ is called the *normal form* of t.

We use the symbol $\simeq$ for equations in an equational theory, and the symbol $=$ is taken to mean syntactical identity. Let E be a set of equational axioms. $T(E)$ denotes the equational theory axiomatized by E, i.e. the least congruence including E.

Given an equational axiom system E, the Knuth-Bendix completion procedure is well known as a method to find a complete TRS R such that $t_1 \simeq t_2$ in $T(E)$ if and only if $t_1{\downarrow} = t_2{\downarrow}$ with respect to R for any two terms t_1 and t_2. It is obvious that such a TRS can be viewed as an algorithm to solve the word problem of $T(E)$.

The kernel function of Metis is the completion procedure, significantly improved with better capabilities and operability by the incorporation of many new facilities. It has a lot of functions needed before, during,

and after generation of TRSs. For example, Metis can provide us with several kinds of ordering methods of terms, but the user can orient an equation with little knowledge of the ordering methods, and obtain an appropriate rewrite rule that does not violate termination of the TRS. If the equation cannot be oriented to either direction, Metis offers the user several kinds of recipe. It manipulates inequations as well as equations and provides special handling of associative-commutative operators in the completion procedure.

Several characteristic features of Metis will now be described. Appendix 4 lists several Metis execution examples.

3.2 Well-founded ordering of terms

As can be seen from the above description, a key point of the completion procedure is ensuring termination of a TRS. The standard way to assure termination of a system is to introduce a well-founded order on the objects of the system and show that the operations in the system always reduce the objects with respect to the order.

Well-founded orders $\prec$ on $\mathcal{T}(F, V)$ with the following properties are usually used on TRSs.

(1) If $t_1 \prec t_2$, then $\theta(t_1) \prec \theta(t_2)$ for any substitution θ.

(2) If $t_1 \prec t_2$, then $c[t_1] \prec c[t_2]$.

Property (1) is called *stability* and (2) *monotonicity*. If there is a monotonic and stable well-founded order on $\mathcal{T}(F, V)$ such that $l \succ r$ for every rule $l \rightarrow r$, it is obvious that the TRS terminates. There has been a lot of research on such ordering methods, such as the well-known Dershowitz recursive path ordering [Dershowitz 82]. The original version of the recursive path ordering is defined on the set $\mathcal{T}(F)$ of ground terms. Here, however, we extend the definition to the set $\mathcal{T}(F, V)$ of all the terms.

Let $<$ be a partial order on the set of function symbols F. The recursive path ordering $\prec$ of $\mathcal{T}(F, V)$ is then defined recursively as follows:

(1) For a variable v, there are no terms t such that $t \prec v$.

(2) For a non-variable term $t = g(t_1, \cdots, t_n)$ and a term s, $s \prec t$ if and only if

(2-1) there exists j such that $s \preceq t_j$ or

(2-2) $s = f(s_1, \cdots, s_m)$ and $s_i \prec t$ for all i and

(2-2-1) $f < g$ or

(2-2-2) $f = g$ and $(s_1, \cdots, s_m) \overset{*}{\prec} (t_1, \cdots, t_n)$, where $\overset{*}{\prec}$ is the multi-set ordering [Dershowitz 79] induced by $\prec$.

In (2-2-2) of the above definition, adoption of the multi-set ordering is not always necessary. If the function symbol f is varyadic (i.e. takes an arbitrary number of arguments) and the order of the arguments does not affect the value of the function (for example, $\sum$ and $\prod$ representing finite sum and product), the multi-set ordering is probably the most reasonable. However, if the function symbol f has a fixed arity, the lexicographic ordering is more suitable in many cases. There may even be cases where the kachinuki ordering [Sakai 85] is the most appropriate.

Metis can handle any of these three versions of the recursive path ordering, namely multi-set, lexicographic, and kachinuki. The user can employ arbitrary combinations of them, function by function. As long as the lexicographic order is applied only to function symbols of fixed arity, any combination defines a monotone and stable well-founded order on $\mathcal{T}(F, V)$. Moreover, if the underlying order $<$ on F is total and the lexicographic or the kachinuki ordering are employed for any function symbol, then it is a total ordering on the limited domain $\mathcal{T}(F)$ of the ground terms, a very important property as we shall see later.

Metis converts axioms to rewrite rules $l \rightarrow r$ such that $l \succ r$. Metis allows the user to define the underlying partial order $<$ on F incrementally during the completion procedure. If the user knows little about the above ordering method, Metis can suggest what ordering is needed on F in order to orient an equation to a certain direction. Thus, when both are possible, the user just has to decide which direction an equation should be oriented to.

3.3 Associative and commutative operators

The weakest point of the Knuth-Bendix completion procedure is revealed by equations that cannot be converted to rules without violating the termination of the TRS. The most typical example of such axioms is the commutative laws, such as $A+B \simeq B+A$. Encounter with such an equation causes unsuccessful stop in the procedure. Metis has several countermeasures to deal with this situation. The general measures will be described later.

It is clearly the commutativity of operators that is the main source of the above failure. In many cases, commutative operators are also associative. Metis has a specific countermeasure effective only against the commutative laws combined with the associative laws of the same operators. A function symbol is called an *AC-operator* if it satisfies the associative and the commutative law. Metis is equipped with an algorithm of special unification for AC-operators (called AC-unification) devised by Fages [Fages 84] and can execute the AC-completion procedure based on Peterson and Stickel's principle [Peterson 81].

For example, if Metis is told that $+$ is an AC-operator, then the axioms $A+B \simeq B+A$ and $(A+B)+C \simeq A+(B+C)$ are acquired implicitly and AC-unification and AC-reduction are activated for $+$. Thus, Metis can generate $0+Y+(-(X+Y)) \simeq (-X)+0$ as a critical pair between the same two rules $(-X)+X \to 0$ by AC-unification, since

$$0+Y+(-(X+Y)) \Leftarrow (-X)+X+Y+(-(X+Y)) \Rightarrow (-X)+0.$$

If it has the rule $0+A \to A$, the above critical pair is reduced to $Y+(-(X+Y)) \simeq -X$ by AC-reduction.

As shown in the above example, an AC-operator is supposed to be a binary function symbol and Metis allows us to use infix notation for binary function symbols. Inside Metis an AC-operator is treated as if it were varyadic. For example, the term $t_1+\cdots+t_n$ is converted to $+(t_1,\ldots,t_2)$ with a varyadic function symbol $+$, in whatever order the operator $+$ is applied to the arguments. The multi-set ordering is assumed to be the ordering method for AC-operators unless otherwise specified, since the

above treatment makes it the most reasonable ordering, as mentioned above.

3.4 Orientation-free rules and S-strategy

There exist many equations other than commutative laws which cannot be converted to terminating rules. The approach of incorporating special unification algorithms for such equations has been studied systematically by Jouannaud and Kirchner [Jouannaud 84].

A simple trick to handle non-orientable equations is introducing a new function symbol. For example, if the equation $A^2 \simeq A \times A$ cannot be oriented to either direction, a new function symbol *square* is introduced and the problematic equation is divided into the two equations $A^2 \simeq square(A)$ and $A \times A \simeq square(A)$. Now Metis can continue the completion procedure, since both equations can be oriented left to right. This technique seems to be too simple, but the effect is worth implementation [Knuth 70, Sakai 84].

A more radical remedy for such equations is the adoption of orientation-free rules. This remedy is called the unfailing completion procedure [Hsiang 85, Bachmair 8]. Metis is equipped with an extended version of the unfailing completion procedure called S-strategy devised by Hsiang and Rusinowitch [Hsiang 85]. The S-strategy has enabled Metis to manipulate not only non-orientable equations, but also inequational axioms as well as equational axioms.

The S-strategy can be viewed as a kind of refutational theorem proving technique for systems of equations and inequations. Before introducing the S-strategy, we will extend the concepts of reduction and critical pairs and introduce the concept of extended narrowing and subsumption. Let us fix a monotonic and stable well-founded order $\prec$ on $\mathcal{T}(F, V)$.

A term t is said to be *reduced* to another term u by an equation $l \simeq r$ (or $r \simeq l$), if $t \succ u$ and there exists a substitution θ such that $c[\theta(l)] = t$ and $c[\theta(r)] = u$. This reduction is called *extended reduction (by an equation)* and denoted also by $t \Rightarrow u$.

Let $l_1 \simeq r_1$ (or $r_1 \simeq l_1$) and $l_2 \simeq r_2$ (or $r_2 \simeq l_2$) be equations. Let s be a non-variable subterm of l_2 such that l_1 and s have a most general unifier θ. Let $l_2 = c[s]$. If $\theta(l_1) \not\preceq \theta(r_1)$ and $\theta(l_2) \not\preceq \theta(r_2)$, then the pair $\theta(c[r_1]) \simeq \theta(r_2)$ is called an *extended critical pair* between $l_1 \simeq r_1$ (or $r_1 \simeq l_1$) and $l_2 \simeq r_2$ (or $r_2 \simeq l_2$).

If every rule $l \to r$ has the property that $l \succ r$, the above definitions are natural extensions of the ordinary reduction by a rule and the ordinary critical pairs between rules. For example, if $l \succ r$, the condition that $t \succ u$ in reducing t to u weakens the rewrite power of the equation $l \simeq r$ exactly to the same level as that of the rule $l \to r$, since $\prec$ is stable and monotonic. Similarly, if $l_1 \succ r_1$ and $l_2 \succ r_2$, the set of all extended critical pairs between equations $l_1 \simeq r_1$ and $l_2 \simeq r_2$ is equal to the set of all critical pairs between rules $l_1 \to r_1$ and $l_2 \to r_2$.

Let $l_1 \simeq r_1$ (or $r_1 \simeq l_1$) be an equation and $l_2 \not\simeq r_2$ (or $r_2 \not\simeq l_2$) be an inequation. Let s be a non-variable subterm of l_2 such that l_1 and s have a most general unifier θ. Let $l_2 = c[s]$. If $\theta(l_1) \not\preceq \theta(r_1)$, then the inequation $\theta(c[r_1]) \not\simeq \theta(r_2)$ is said to be *narrowed* from $l_2 \not\simeq r_2$ (or $r_2 \not\simeq l_2$) using $l_1 \simeq r_1$ (or $r_1 \simeq l_1$).

An equation $t \simeq u$ is said to be *subsumed* by other equations $l_1 \simeq r_1$ (or $r_1 \simeq l_1$), ..., $l_n \simeq r_n$ (or $r_n \simeq l_n$), if there exists a substitution θ such that $c[\theta(l_1), \ldots, \theta(l_n)] = t$ and $c[\theta(r_1), \ldots, \theta(r_n)] = u$. An inequation $t \not\simeq u$ is said to be *subsumed* by another inequation $l \not\simeq r$ (or $r \not\simeq l$), if there exists a substitution θ such that $\theta(l) = t$ and $\theta(r) = u$.

Unfailing completion is a modified version of ordinary completion employing extended critical pairs and extended reduction instead of the ordinary ones; and the S-strategy can be viewed as the unfailing completion with refutation by extended narrowing.

Suppose that a system of equational and inequational axioms is given together with an equation or inequation to be solved (called the target formula). The S-strategy is the following procedure:

Step 0: Set E to be the given axiom system plus the negation of the target formula (Skolemized if necessary). Set R to be empty. Go to Step 1.

Step 1: If E is empty, the current value of R is a complete set of equations and inequations deduced from the axioms and the negation of the target formula, in the sense that neither new equations nor new inequations can be derived. Since R is also consistent, the target formula cannot be deduced from the axioms. If E is not empty, go to Step 2.

Step 2: Remove an equation $t \simeq u$ or inequation $t \not\simeq u$ (called the ruling formula) from E. Go to Step 3.

Step 3: If the ruling formula is an equation, move all the equations $l \simeq r$ and all the inequations $l \not\simeq r$ from R to E such that either l or r is reducible by the ruling formula and remove all the equations subsumed by the ruling formula from R. If the ruling formula is an inequation, remove all the inequations subsumed by the ruling formula from R. Go to Step 4.

Step 4: Append the ruling formula to R. Construct all the extended critical pairs and all the narrowed inequations between the ruling formula and all the equations and inequations in R. Append them to E. For each equation $t \simeq u$ or inequation $t \not\simeq u$ in E, find irreducible forms $t\downarrow$ and $u\downarrow$ with respect to equations in R. If there is an inequation $t \not\simeq u$ such that $t\downarrow$ and $u\downarrow$ are unifiable, then stop. A contradiction is detected and, therefore, the target formula is deduced from the original axiom system. Otherwise, let the new E be the set of equations $t\downarrow \simeq u\downarrow$ such that $t\downarrow \neq u\downarrow$ not subsumed by any equation in R and inequations $t\downarrow \not\simeq u\downarrow$ not subsumed by any inequation in R. Go to Step 1.

The unfailing completion differs from the S-strategy only in that it does not treat non-ground inequations. If the ordering $\prec$ is total on the set $\mathcal{T}(F)$ of all ground terms, the S-strategy is logically complete and, therefore, so is the unfailing completion.

REFERENCES

[Bachmair 86] Bachmair, L., Dershowitz, N., and Plaisted, D.A.: *Completion without failure*, private communication (1986)

[Barendregt 84] Barendregt, H.P. *The lambda calculus:Its syntax and semantics*, Studies in Logic and the Foundations of Mathematics 103, North-Holland, revised edition (1984)

[Dershowitz 79] Dershowitz, N. and Manna, Z.: *Proving termination with multiset orderings*, Communications of the ACM 22 (1979) 465-467

[Dershowitz 82] Dershowitz, N.: *Orderings for term-rewriting systems*, Theoretical Computer Science 17 (1982) 279-301

[Fages 84] Fages, F.: *Associative-commutative unification*, 7th. International Conference on Automatic Deduction, Lecture Note on Computer Science 170, Springer (1984) 194-208

[Gorden 79] Gorden, M.J., Milner, A.J., and Wadsworth, C.P.: *Edinburgh LCF*, Lecture Notes on Computer Science 78, Springer (1979)

[Hilbert 39] Hilbert, D. und Bernays, P.: *Grundlagen der Mathematik II*, Springer (1939)

[Hindley 86] Hindley, J.R. and Seldin, J.P.: *Introduction to combinators and λ-calculus*, London Mathematical Society Student Text 1, Cambridge University Press (1986)

[Hirose 86] Hirose, K.: *An approach to proof checker*, Lecture Notes on Computer Science 233, Springer (1986)

[Hsiang 85] Hsiang, J. and Rusinowitch, M.: *On word problems in equational theories*, private communication (1985)

[Huet 80] Huet, G. and Oppen, D. C.: *Equations and Rewrite Rules: a survey*, Formal Language: Perspectives and Open Problems Academic Press (1980) 349-405

[Huet 81] Huet, G.: *A complete proof of correctness of the Knuth-Bendix completion algorithm*, J. Computer and System Science 23 (1981) 11-21

[Huet 82] Huet, G. and Hullot, J-M.: *Proofs by induction in equational theories with constructors*, J. Computer and System Science 25 (1982) 239-266

[Jouannaud 84] Jouannaud, J-P. and Kirchner, H.: *Completion of a set of rules modulo a set of equations*, 11th ACM POPL (1984)

[Ketonen 83] Ketonen, J and Weening, J.S.: *EKL — an interactive proof checker, user's reference manual*, Department of Computer Science, Stanford University (1983)

[Knuth 70] Knuth, D. E., Bendix, P. B.: *Simple word problems in universal algebras*, Computational Problems in Abstract Algebra, J. Leech (ed), Pergamon Press, Oxford, (1970) 263-297. also in: Automated Reasoning 2 (Siekmann and Wrightson eds.), Springer (1983)

[Lang 83] Lang, S.: *Linear Algebra*, Addison-Wesley, 2nd edition (1983)

[Ohsuga 86] Ohsuga, A. and Sakai, K.: *Metis: a term rewriting system generator*, RIMS Symposium on Software Science and Engineering (1986)

[Peterson 81] Peterson, G.E. and Stickel, M.: *Complete sets of reductions for equational theories with complete unification algorithms*, J.ACM, vol 28 (1981) 233-264

[Sakai 84] Sakai, K.: *An ordering method for term rewriting systems*, TR-062, ICOT (1984)

[Sakai 85] Sakai, K.: *Knuth-Bendix algorithm for Thue system based on kachinuki ordering*, TM-0087, ICOT (1985)

[Sato 84] Sato, M. and Sakurai, T.: *Qute: a functional language based on unification*, Proceedings of the International Conference on Fifth Generation Computer Systems (1984) 157-165

[Sato 85] Sato, M.: *Typed logical calculus*, TR-85-13, Department of Computer Science, University of Tokyo (1985)

[Shanker 85] Shanker, N.: *Towards mechanical metamathematics*, Journal of Automated Reasoning, 1 (1985) 407-434

[Trybulec 85] Trybulec, A. and Blair, H.: *Computer assisted reasoning with Mizar*, IJCAI'85 (1985) 26-28

Appendix 1. Proof Description Language (PDL) for Linear Algebra

1.1 Logical connectives

`contradiction`	$\bot$ (contradiction)
`\`	$\neg$ (negation)
`&`	$\wedge$ (and)
`|`	$\vee$ (or)
`->` and `<-`	$\rightarrow$ (if ... then ...)
`<->`	$\leftrightarrow$ (... if and only if ...)
`all`	$\forall$ (for all)
`some`	$\exists$ (for some)
`some!`	$\exists!$ (there exists only one ...)
`=`	$=$ (equality)
`:`	(... is of type ...)
`if` A `then` s `else` t	if-then-else function

1.2 Mathematical predicates

`=<`	$\leq$ (less than or equal to)
`<`	$<$ (less than)
`>=`	$\geq$ (greater than or equal to)
`>`	$>$ (greater than)

1.3 Primitive types and type constructors

`sca`	scalar
`mat`	matrix
`nat`	natural number (0-origin)
`seq(`τ`)`	sequence of τs
`perm`	permutation
m`..`n	integer from m to n

1.4 Primitive functions

`col_size(`α`:mat):nat`	column size of a matrix
`row_size(`α`:mat):nat`	row size of a matrix
`(`α`:mat[`i`:1..col_size(`α`),` `  `j`:1..row_size(`α`)]):sca`	element of a matrix
`(mat{`i`:1..`n`:nat, `j`:1..`m`:nat}` `  `$f(i,j)$`:sca):mat`	functional definition of a matrix
`length(`σ`:seq(`τ`)):nat`	length of a sequence
`(`σ`:seq(`τ`)[`i`:1..length(`σ`)]):`τ	element of a sequence
`(seq{`i`:1..`n`:nat}`$f(i)$`:`τ`):seq(`τ`)`	functional definition of a sequence
`domain(`p`:perm):nat`	domain of a permutation
`sgn(`p`:perm):sca`	signature of a permutation
`id(`n`:nat):perm`	identity permutation
`inv(`p`:perm):perm`	inverse of a permutation ($^{-1}$)
`(`p`:perm*`q`:perm):perm`	composition of permutations ($\circ$)
`(sum{`i`:`m`..`n`}`$f(i)$`:sca):sca` `(sum{`i`:perm<`m`>}`$f(i)$`:sca):sca`	finite indexed sum $\sum$
`(prod{`i`:`m`..`n`}`$f(i)$`:sca):sca` `(prod{`i`:perm<`m`>}`$f(i)$`:sca):sca`	finite indexed product $\prod$

1.5 Proper NK rules and their variants

1.5.1 ¬, ⊥ rules

S1	$\begin{array}{c}[A]\\ \vdots\\ \bot\\ \hline \neg A\end{array}$	\A since assume A ⋮ contradiction end_since
S16	$\begin{array}{c}[\neg A]\\ \vdots\\ \bot\\ \hline A\end{array}$	A since assume \A ⋮ contradiction end_since
S2	$\begin{array}{c}\vdots \quad \vdots\\ A \quad \neg A\\ \hline \bot\end{array}$	⋮ A ⋮ \A hence contradiction

1.5.2 $\wedge$ rules

S4	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}$	$\begin{array}{l}\vdots \\ A \\ \vdots \\ B \\ \texttt{hence } A\&B\end{array}$
S5	$\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{A[B]}$	$\begin{array}{l}\vdots \\ A\&B \\ \texttt{hence } A[B]\end{array}$
S4*	$\dfrac{\begin{array}{ccc}\vdots & & \vdots \\ A_1 & \cdots & A_n\end{array}}{A_1 \wedge \cdots \wedge A_n}$	$\begin{array}{l}\vdots \\ A_1 \\ \vdots \\ A_n \\ \texttt{hence } A_1\& \cdots \&A_n\end{array}$
S5*	$\dfrac{\begin{array}{c}\vdots \\ A_1 \wedge \cdots \wedge A_n\end{array}}{A_i}$	$\begin{array}{l}\vdots \\ A_1\& \cdots \&A_n \\ \texttt{hence } A_i\end{array}$

1.5.3 ∨ rules

S6	$\dfrac{\vdots\ A[B]}{A \vee B}$	$\vdots$ $A[B]$ hence $A \mid B$
S7	$\dfrac{\vdots\ A \vee B \qquad [A]\ \vdots\ C \qquad [B]\ \vdots\ C}{C}$	$\vdots$ $A \mid B$ hence C since divide and conquer case A … C case B … C end_since
S7'	$\dfrac{[A]\ \vdots\ C \qquad [\neg A]\ \vdots\ C}{C}$	C since divide and conquer case A … C else … C end_since

S6*	$\dfrac{\begin{array}{c}\vdots\\ A_i\end{array}}{A_1 \vee \cdots \vee A_n}$	$\begin{array}{l}\vdots\\ A_i\\ \texttt{hence } A_1 \mid \cdots \mid A_n\end{array}$
S7*	$\dfrac{\begin{array}{ccccc} & & [A_1] & & [A_n] \\ \vdots & & \vdots & \cdots & \vdots \\ A_1 \vee \cdots \vee A_n & & C & & C \end{array}}{C}$	$\begin{array}{l}\vdots\\ A_1 \mid \cdots \mid A_n\\ \texttt{hence } C\\ \texttt{since divide and conquer}\\ \boxed{\begin{array}{l}\texttt{case } A_1\\ \vdots\\ C\end{array}}\\ \vdots\\ \boxed{\begin{array}{l}\texttt{case } A_n\\ \vdots\\ C\end{array}}\\ \texttt{end\_since}\end{array}$
S7*'	$\dfrac{\begin{array}{ccccc} [A_1] & & [A_n] & & [\neg A_1 \wedge \cdots \wedge \neg A_n] \\ \vdots & \cdots & \vdots & & \vdots \\ C & & C & & C \end{array}}{C}$	$\begin{array}{l}C\\ \texttt{since divide and conquer}\\ \boxed{\begin{array}{l}\texttt{case } A_1\\ \vdots\\ C\end{array}}\\ \vdots\\ \boxed{\begin{array}{l}\texttt{case } A_n\\ \vdots\\ C\end{array}}\\ \boxed{\begin{array}{l}\texttt{else}\\ \vdots\\ C\end{array}}\\ \texttt{end\_since}\end{array}$

1.5.4 — rules

S8	$\begin{array}{c}[A]\\ \vdots\\ B\\ \hline A \rightarrow B\end{array}$	A->B [B<-A] since assume A $\vdots$ B end_since
S9	$\dfrac{\begin{array}{cc}\vdots & \vdots\\ A \rightarrow B & A\end{array}}{B}$	$\vdots$ A->B [B<-A] $\vdots$ A hence B
S8*	$\begin{array}{c}[A_1]\cdots[A_n]\\ \vdots\\ B\\ \hline A_1 \wedge \cdots \wedge A_n \rightarrow B\end{array}$	A_1& $\cdots$ &A_n->B [B<-A_1& $\cdots$ &A_n] since assume $A_1,\ldots,A_n$ $\vdots$ B end_since
S9*	$\dfrac{\begin{array}{cccc}\vdots & \vdots & & \vdots\\ A_1 \wedge \cdots \wedge A_n \rightarrow B & A_1 & \cdots & A_n\end{array}}{B}$	$\vdots$ A_1& $\cdots$ &A_n->B [B<-A_1& $\cdots$ &A_n] $\vdots$ A_1 $\vdots$ A_n hence B

1.5.5 ↔ rules

S10	$\dfrac{\begin{array}{cc}[A] & [B] \\ \vdots & \vdots \\ B & A\end{array}}{A \leftrightarrow B}$	A<->B since only_if_part assume A ⋮ B if_part assume B ⋮ A end_since
S11	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A \leftrightarrow B & A[B]\end{array}}{B[A]}$	⋮ A<->B ⋮ $A[B]$ hence $B[A]$

1.5.6 ∀ rules

Hereafter, we adopt boldface letters for representing sequences of n symbols (or combinations of simbols). For example, a and $\exists \mathbf{x} : \mathbf{t}$ denote $a_1, \ldots, a_n$ and $\exists x_1 : t_1 \ \ldots \ \exists x_n : t_n$, respectively.

† S12	$\dfrac{\begin{array}{c}[a : t] \\ \vdots \\ A(a)\end{array}}{\forall x : t\ A(x)}$	all{$x : t$}$A(x)$ since let $a : t$ be arbitrary $\vdots$ $A(a)$ end_since
S13	$\dfrac{\vdots \quad \forall x : t\ A(x) \qquad \vdots \quad a : t}{A(a)}$	$\vdots$ all{$x : t$}$A(x)$ $\vdots$ $a : t$ hence $A(a)$
‡ S12*	$\dfrac{\begin{array}{c}[\mathbf{a} : \mathbf{t}] \\ \vdots \\ A(\mathbf{a})\end{array}}{\forall \mathbf{x} : \mathbf{t}\ A(\mathbf{x})}$	all{x : t}$A(x)$ since let a : t be arbitrary $\vdots$ $A(\mathbf{a})$ end_since
S13*	$\dfrac{\vdots \quad \forall \mathbf{x} : \mathbf{t}\ A(\mathbf{x}) \qquad \vdots \quad \mathbf{a} : \mathbf{t}}{A(\mathbf{a})}$	$\vdots$ all{x : t}$A(x)$ $\vdots$ a : t hence $A(\mathbf{a})$

† a does not occur in the assumptions and the consequence of the proof figure and is a new free variable in the proof in PDL.

‡ a do not occur in the assumptions and the consequence of the proof figure and are new free variables in the proof in PDL.

1.5.7 ∃ rules

S14	$\dfrac{\vdots \quad A(a) \qquad \vdots \quad a : t}{\exists x : t\ A(x)}$	$\vdots$ $A(a)$ $\vdots$ $a : t$ `hence some{`$x : t$`}`$A(x)$
† S20	$\dfrac{\vdots \quad \exists x : t\ A(x)}{A(\epsilon x : t\ A(x))}$	$\vdots$ `some{`$x : t$`}`$A(x)$ `let` $a : t$ `be such_that` $A(a)$
S14*	$\dfrac{\vdots \quad A(\mathbf{a}) \qquad \vdots \quad \mathbf{a} : \mathbf{t}}{\exists \mathbf{x} : \mathbf{t}\ A(\mathbf{x})}$	$\vdots$ $A(\mathbf{a})$ $\vdots$ $\mathbf{a} : \mathbf{t}$ `hence some{`$\mathbf{x} : \mathbf{t}$`}`$A(\mathbf{x})$
‡ S20*	$\dfrac{\vdots \quad \exists \mathbf{x} : \mathbf{t}\ A(\mathbf{x})}{A(\epsilon \mathbf{x} : \mathbf{t}\ A(\mathbf{x}))}$	$\vdots$ `some{`$\mathbf{x} : \mathbf{t}$`}`$A(\mathbf{x})$ `let` $\mathbf{a} : \mathbf{t}$ `be such_that` $A(\mathbf{a})$

† $\epsilon x : t\ A(x)$ is the notation introduced by Hilbert [Hilbert 39] to represent an object x satifying $A(x)$ if there exists. a is a new free variable in the proof in PDL.

‡ $\epsilon \mathbf{x} : \mathbf{t}\ A(\mathbf{x})$ is the notation to represent a sequence of objects $\mathbf{x}(= x_1, \ldots, x_n)$ satifying $A(\mathbf{x})$ if there exists. $\mathbf{a}$ is a sequence of new free variables in the proof in PDL.

1.5.8 rules for other logical connectives

† U1	$\dfrac{\begin{array}{ccc}\vdots & [A(b)][b:t] & \\ & [A(c)][c:t] & \\ & \vdots \quad \vdots & \\ A(a) & a:t \quad b=c & \end{array}}{\exists!x:t\ A(x)}$	some! $\{x:t\}A(x)$ since existence $\vdots$ $A(a)$ $\vdots$ $a:t$ uniquness let $b,c:t$ be such_that $A(b), A(c)$ $\vdots$ $b=c$ end_since
‡ U2	$\dfrac{\begin{array}{c}\vdots \\ \exists!x:t\ A(x)\end{array}}{\begin{array}{r}A(\epsilon x:t\ A(x)) \wedge \forall\{y:t\}(A(y) \\ \rightarrow y=\epsilon x:t\ A(x))\end{array}}$	$\vdots$ some! $\{x:t\}A(x)$ let $a:t$ be such_that $A(a)$

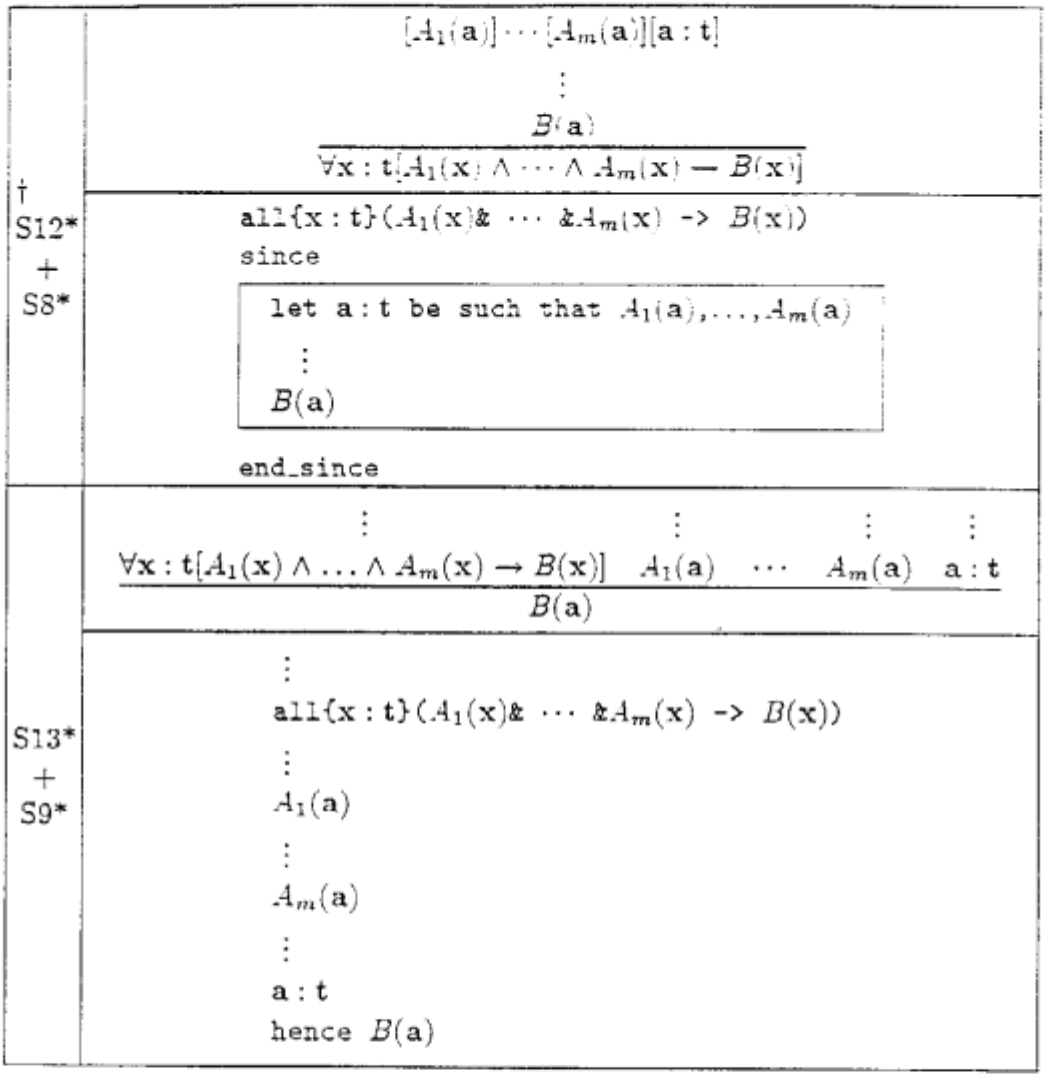
† b or c does not occur in the assumptions and the consequence of the proof figure and is a new free variable in the proof in PDL.

‡ a is a new free variable in the proof in PDL.

$$\frac{A}{s = \texttt{if } A \texttt{ then } s \texttt{ else } t} \qquad \frac{\neg A}{t = \texttt{if } A \texttt{ then } s \texttt{ else } t}$$

$$t = \texttt{if } A \texttt{ then } t \texttt{ else } t$$

1.5.9 Other useful rules



† **a** do not occur in the assumptions and the consequence of the proof figure and are new distinct free variables in the proof in PDL.

1.5.10 Axioms and definitions

A1	A is a mathematical axiom	`axiom`
		A
		`end_axiom`

D1†	$\forall \mathbf{x} : \mathbf{t}[A(\mathbf{x})$ is defined as $B(\mathbf{x})]$	`let` $A(\mathbf{a} : \mathbf{t})$`:<->`$B(\mathbf{a})$
‡ D2	$\forall \mathbf{x} : \mathbf{t}[A_1(\mathbf{x}) \wedge \cdots \wedge A_m(\mathbf{x}) \rightarrow$ $f(\mathbf{x})$ is defined as $g(\mathbf{x})]$	`let` $f(\mathbf{a} : \mathbf{t})$`:=`$g(\mathbf{a})$ `<-` $A_1(\mathbf{a})$`&` $\cdots$ `&`$A_m(\mathbf{a})$

‡
D3
	$[\mathbf{a} : \mathbf{t}][A_1(\mathbf{a})] \cdots [A_m(\mathbf{a})]$
	$\vdots$
	$\exists! y : t\ B(\mathbf{a}, y)$
	$\forall \mathbf{x} : \mathbf{t}[A_1(\mathbf{x}) \wedge \cdots \wedge A_m(\mathbf{x}) \rightarrow$
	$f(\mathbf{x})$ is defined as such $y : t$
	that satisfies $B(\mathbf{x}, y)]$
		`function`
		$f(\mathbf{a} : \mathbf{t}) : t$
		`assume` $A_1(\mathbf{a}), \ldots, A_m(\mathbf{a})$
		`attain` $B(\mathbf{a}, f(\mathbf{a}))$
		$\vdots$
		`some!{`$y : t$`}`$B(\mathbf{a}, y)$
		`end_function`

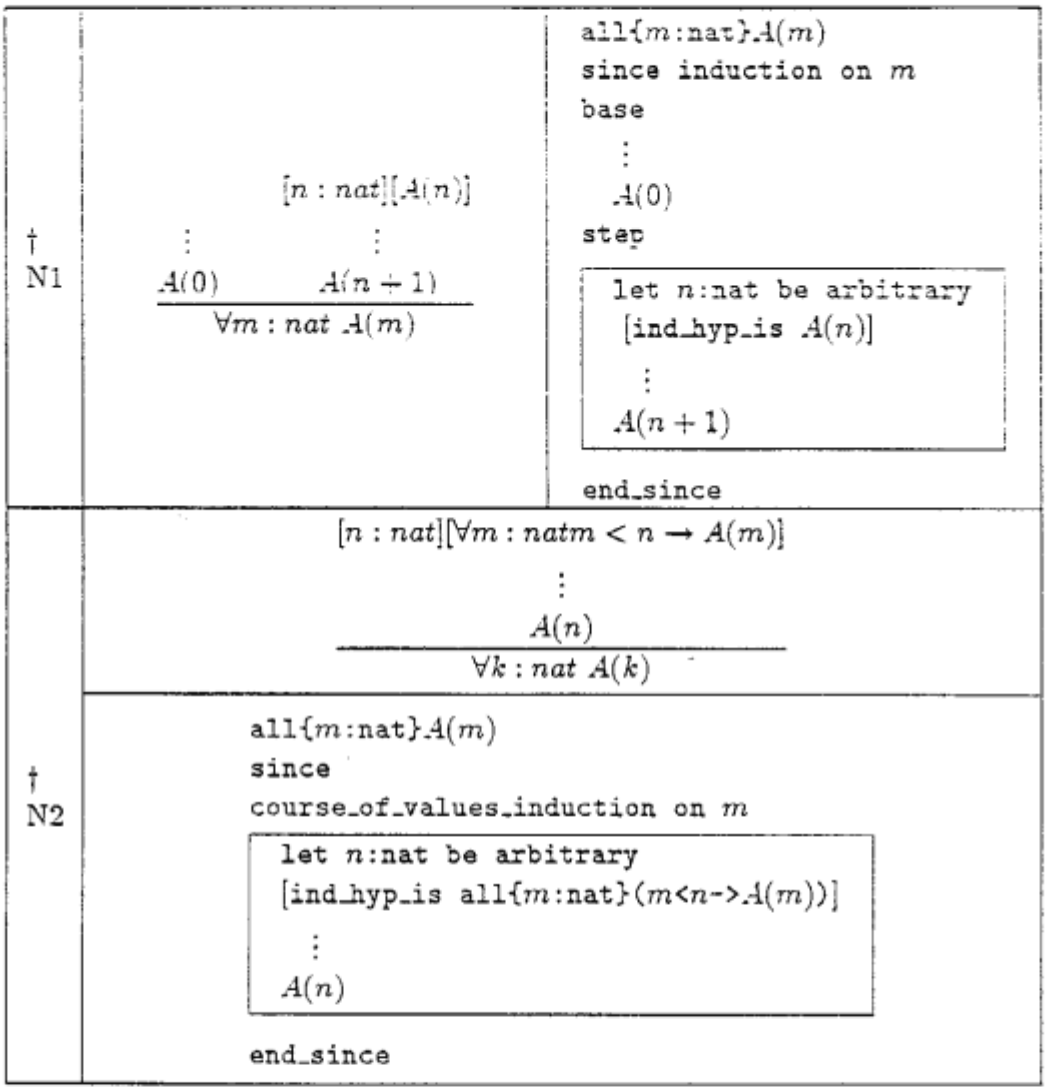
† A is a new predicate symbol.

‡ f is a new function symbol.

1.6 Rules for equality

E1	$a = a$	$a=a$
E2	$\dfrac{\begin{array}{c}\vdots\\ a = b\end{array}}{b = a}$	$\begin{array}{l}\vdots\\ a=b\\ \texttt{hence } b=a\end{array}$
E3	$\dfrac{\begin{array}{cc}\vdots & \vdots\\ a = b & b = c\end{array}}{a = c}$	$\begin{array}{l}\vdots\\ a=b\\ \vdots\\ b=c\\ \texttt{hence } a=c\end{array}$
E4	$\dfrac{\begin{array}{cc}\vdots & \vdots\\ a = b & A(a)\end{array}}{A(b)}$	$\begin{array}{l}\vdots\\ a=b\\ \vdots\\ A(a)\\ \texttt{hence } A(b)\end{array}$
E5	$\dfrac{\begin{array}{c}\vdots\\ a = b\end{array}}{t(a) = t(b)}$	$\begin{array}{l}\vdots\\ a=b\\ \texttt{hence } t(a)=t(b)\end{array}$
E6	$\dfrac{\begin{array}{c}\vdots\\ a = b\end{array}}{A(a) \leftrightarrow A(b)}$	$\begin{array}{l}\vdots\\ a=b\\ \texttt{hence } A(a)\texttt{<->}A(b)\end{array}$
E5*	$\dfrac{\dfrac{\dfrac{a_1 = a_1'}{t_1(a_1) = t_1(a_1')[\equiv t_2(a_2)]} \quad a_2 = a_2'}{\begin{array}{c}\vdots\\ t_1(a_1) = t_n(a_n)\end{array}}$	$\begin{array}{l}t_1(a_1)=t_2(a_2)\\ \quad\vdots\\ \quad=t_{n-1}(a_{n-1})\\ \quad=t_n(a_n)\end{array}$
E6*	$\dfrac{\dfrac{\dfrac{a_1 = a_1'}{A_1(a_1) \leftrightarrow A_1(a_1')[\equiv A_2(a_2)]} \quad a_2 = a_2'}{\begin{array}{c}\vdots\\ A_1(a_1) \leftrightarrow A_n(a_n)\end{array}}$	$\begin{array}{l}A_1(a_1)\texttt{<->}A_2(a_2)\\ \quad\vdots\\ \quad\texttt{<->}A_{n-1}(a_{n-1})\\ \quad\texttt{<->}A_n(a_n)\end{array}$

1.7 Rules for natural numbers



‡† N3	$\dfrac{\begin{array}{c}\vdots\\ \exists k : nat\ A(k)\end{array}}{\begin{array}{l}\mu x : nat\ A(x) : nat \wedge A(\mu x : nat\ A(x))\\ \quad \wedge \forall\{m : nat\}(m < \mu x : nat\ A(x) \rightarrow \neg A(m))\end{array}}$
	$\begin{array}{l}\vdots\\ \texttt{some\{}k\texttt{:nat\}}A(k)\\ \texttt{let } n\texttt{:nat be minimum such that } A(n)\end{array}$

† n does not occur in the assumptions and the consequence of the proof figure and is a new free variable in the proof in PDL.

‡ $\mu x : nat\ A(x)$ is the notation to represent the least natural number x satifying $A(x)$ if there exists.

$$\frac{n : nat}{n = 0 \vee \exists m : nat\ n = m + 1} \qquad \frac{n : nat}{n = n + 0 = 0 + n}$$

$$\frac{l : nat \quad n : nat \quad m : nat \quad n + l \leq m + l}{n \leq m} \qquad \frac{n : nat \quad m : nat}{n + m = m + n}$$

$$\frac{n : nat \quad m : nat \quad n < m + 1}{n < m \vee n = m} \qquad \frac{n : nat \quad m : nat \quad l : nat}{(n + m) + l = n + (m + l)}$$

$$\frac{n : nat \quad m : nat}{\exists l : nat(n \leq l \wedge m \leq l)} \qquad \frac{n : nat \quad m : nat}{\exists l : nat(l \leq n \wedge l \leq m)}$$

$$\frac{n : nat \quad m : nat \quad l : nat \quad n < m \quad m < l}{n < l} \qquad \frac{n : nat \quad m : nat}{n < m \vee n = m \vee m < n}$$

$$\frac{m : nat}{\neg(m < m)} \qquad \frac{m : nat \quad n : nat \quad m < n \quad n \leq m}{\bot}$$

$$\frac{m : nat \quad n : nat \quad m < n \quad n < m}{\bot} \qquad \frac{m : nat \quad n : nat \quad m \leq n \quad n \leq m}{m = n}$$

$$\frac{m : nat}{0 \leq m} \qquad \frac{l : nat \quad n : nat \quad m : nat \quad n \leq l}{m + n \leq m + l}$$

1.8 Rules for $\sum$ and $\prod$

† P1 [P1']	$\dfrac{\begin{array}{c}[A(n)]\\ \vdots \\ f(n)=g(n)\end{array}}{\sum\limits_{A(i)} f(i)=\sum\limits_{A(i)} g(i)}$ $\left[\prod\limits_{A(i)} f(i)=\prod\limits_{A(i)} g(i)\right]$	sum{$A(i)$}$f(i)$ = sum{$A(i)$}$g(i)$ [product{$A(i)$}$f(i)$ = product{$A(i)$}$g(i)$] since let n be arbitrary such that $A(n)$ ⋮ $f(n)=g(n)$ end_since
† P2 [P2']	$\dfrac{\begin{array}{c}[A(n)]\\ \vdots \\ \bot\end{array}}{\sum\limits_{A(i)} f(i)=0}$ $\left[\prod\limits_{A(i)} f(i)=1\right]$	sum{$A(i)$}$f(i)$ = 0 [product{$A(i)$}$f(i)$ = 1] since let n be arbitrary such that $A(n)$ ⋮ contradiction end_since
† P3 [P3']	$\dfrac{\begin{array}{cc} & [A(n)]\\ \vdots & \vdots \\ A(k) & n=k\end{array}}{\sum\limits_{A(i)} f(i)=f(k)}$ $\left[\prod\limits_{A(i)} f(i)=f(k)\right]$	⋮ $A(k)$ hence sum{$A(i)$}$f(i)$ = $f(k)$ [hence product{$A(i)$}$f(i)$ = $f(k)$] since let n be arbitrary such that $A(n)$ ⋮ $n=k$ end_since

† n does not occur in the assumptions and the consequence of the proof figure and is a new free variable in the proof in PDL

† P4	$\dfrac{\begin{array}{c}[A(n)]\\ \vdots \\ f(n)=0\end{array}}{\sum_{A(i)} f(i)=0}$	sum{$A(i)$}$f(i)$ = 0 since let n be arbitrary such that $A(n)$ $\vdots$ $f(n)$=0 end_since
† P4'	$\dfrac{\begin{array}{c}[A(n)]\\ \vdots \\ f(n)=1\end{array}}{\prod_{A(i)} f(i)=1}$	product{$A(i)$}$f(i)$ = 1 since let n be arbitrary such that $A(n)$ $\vdots$ $f(n)$=1 end_since
P5	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A(k) & f(k)=0\end{array}}{\prod_{A(i)} f(i)=0}$	$\vdots$ $A(k)$ $\vdots$ $f(k)$=0 hence product{$A(i)$}$f(i)$ = 0

† n does not occur in the assumptions and the consequence of the proof figure and is a new free variable in the proof in PDL

†P6 $$\sum_{i:1..n} C = nC$$

†P6' $$\prod_{i:1..n} C = C^n$$

P7 $$\sum_{A(i)} (t_1 + t_2) = \sum_{A(i)} t_1 + \sum_{A(i)} t_2$$

P7' $$\prod_{A(i)} (t_1 t_2) = \prod_{A(i)} t_1 \prod_{A(i)} t_2$$

P8 $$\sum_{A(i)} \sum_{B(j)} t = \sum_{B(j)} \sum_{A(i)} t$$

P8' $$\prod_{A(i)} \prod_{B(j)} t = \prod_{B(j)} \prod_{A(i)} t$$

†P9 $$\sum_{A(i)} (Ct) = C \sum_{A(i)} t$$

†P9' $$\prod_{A(i)} (t^C) = (\prod_{A(i)} t)^C$$

P10 $$\sum_{A(i)} t = \sum_{A(i) \wedge B(i)} t + \sum_{A(i) \wedge \neg B(i)} t$$

P10' $$\prod_{A(i)} t = \prod_{A(i) \wedge B(i)} t \prod_{A(i) \wedge \neg B(i)} t$$

‡P11 $$\sum_{A(i)} f(i) = \sum_{A(j)} f(j)$$

‡P11' $$\prod_{A(i)} f(i) = \prod_{A(j)} f(j)$$

† C does not have occurences of the free variable i.

‡ $f(\ )$ does not have occurences of the free variables i and j.

1.9 Rules for functionally defined matrices

M1	$\dfrac{\vdots \qquad \vdots}{k:1..m \quad l:1..n}$ $f(k,l) = \left(\underset{\substack{i:1..m \\ j:1..n}}{\mathtt{mat}}\, f(i,j)\right)[k,l]$	$\vdots$ k:1..m $\vdots$ l:1..n hence $f(k,l)$ = (mat{i:1..m,j:1..n}$f(i,j)$)[k,l]

M2 col_size(mat{i:1..m,j:1..n}$f(i,j)$)=m
M3 row_size(mat{i:1..m,j:1..n}$f(i,j)$)=n

1.10 Rules for sequences

$$\mathtt{length}(\mathtt{seq}\{i{:}1..m\}f(i)) = m$$

$$\frac{k:1..m}{(\mathtt{seq}\{i:1..m\}f(i))[k] = f(k)}$$

$$[\mathtt{seq}\{i:1..m\}f(i) = \mathtt{seq}\{j:1..m\}f(j)]$$

$$\frac{\mathtt{seq}\{i:1..m\}f(i) = \mathtt{seq}\{j:1..m\}g(j)}{m{=}n \;\;\&\;\; \mathtt{all}\{i:1..m\}f(i){=}g(i)}$$

$$\frac{\mathtt{all}\{i:1..m\}f(i){=}g(i)}{\mathtt{seq}\{j:1..m\}f(j) = \mathtt{seq}\{k:1..m\}g(k)}$$

1.11 Rules for permutations

PE1 $$id(n) : perm < n > \qquad sgn(id(n)) = 1$$

PE2 $$\frac{p : perm < n >}{p^{-1} : perm < n > \qquad sgn(p^{-1}) = sgn(p)}$$

PE3 $$\frac{p : perm < n > \quad q : perm < n >}{p \circ q : perm < n > \quad sgn(p \circ q) = sgn(p)sgn(q)}$$

PE4 $$\frac{p : perm < n >}{id(n) \circ p = p \circ id(n) = p}$$

PE5 $$\frac{p : perm < n >}{p \circ p^{-1} = id(n) \qquad p^{-1} \circ p = id(n)}$$

PE6 $$\frac{p : perm < n > \quad q : perm < n > \quad r : perm < n >}{(p \circ q) \circ r = p \circ (q \circ r)}$$

PE7 $$\frac{i : 1..n}{id(n)[i] = i}$$

PE8 $$\frac{p : perm < n > \qquad q : perm < n > \qquad i : 1..n}{(p \circ q)[i] = p[q[i]]}$$

PE9 $$\frac{p : perm < n >}{sgn(p) = 1 \vee sgn(p) = -1}$$

†PE10 $$\frac{p : perm < n >}{\prod_{i:1..n} f(i) = \prod_{i:1..n} f(p[i])}$$

‡PE11 $$\frac{q : perm < n >}{\sum_{p:perm<n>} f(p) = \sum_{p:perm<n>} f(p \circ q) = \sum_{p:perm<n>} f(q \circ p)}$$

‡PE12 $$\sum_{p:perm<n>} f(p^{-1}) = \sum_{p:perm<n>} f(p)$$

† $f(\ )$ does not have occurences of the free variable i.

‡ $f(\ )$ does not have occurences of the free variables p.

Appendix 2. Example Proof in PDL

The following is a simple theorem on the determinant of a transpose and its proof quoted from [Lang 83].

Theorem. *Let A be a square matrix. Then* $\mathrm{Det}(A)=\mathrm{Det}({}^tA)$.

Proof. In Theorem 5, we had

$$(\star) \qquad Det(A) = \sum_{\sigma} \epsilon(\sigma) a_{\sigma(1),1} \cdots a_{\sigma(n),n}.$$

Let σ be a permutation of $\{1,\ldots,n\}$. If $\sigma(j) = k$, then $\sigma^{-1}(k) = j$. We can therefore write

$$a_{\sigma(j),j} = a_{k,\sigma^{-1}(k)}.$$

In a product

$$a_{\sigma(1),1} \cdots a_{n,\sigma^{-1}(n)},$$

each integer k from 1 to n occurs precisely once among the integers $\sigma(1),\ldots,\sigma(n)$. Hence this poroduct can be written

$$a_{1,\sigma^{-1}(1)} \cdots a_{n,\sigma^{-1}(n)},$$

and our sum $(\star)$ is equal to

$$\sum_{\sigma} \epsilon(\sigma^{-1}) a_{1,\sigma^{-1}(1)} \cdots a_{n,\sigma^{-1}(n)},$$

because $\epsilon(\sigma)=\epsilon(\sigma^{-1})$. In this sum, each term corresponds to a permutation σ. However, as σ ranges over all permutations, so does σ^{-1} because a permutation determines its inverse uniquely. Hence our sum is equal to

$$(\star\star) \qquad \sum_{\sigma} \epsilon(\sigma) a_{1,\sigma(1)} \cdots a_{n,\sigma(n)}.$$

The sum $(\star\star)$ is precisely the sum giving the expanded form of the determinant of the transpose of A. Hence we have proved what we wanted.

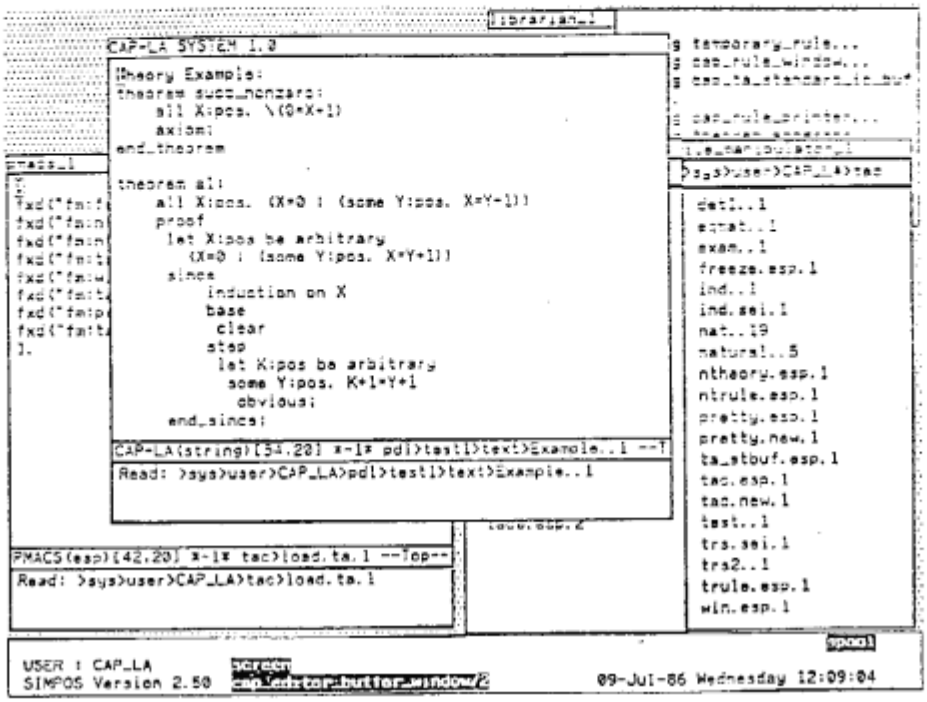
The following is the corresponding description in PDL.

```
theory determinant

  det(A:square)
    := sum{P:perm<col_size(A)>}
        sgn(P)*prod{I:1..col_size(A)}A[P[I],I]

  theorem determinant_of_transpose:
```

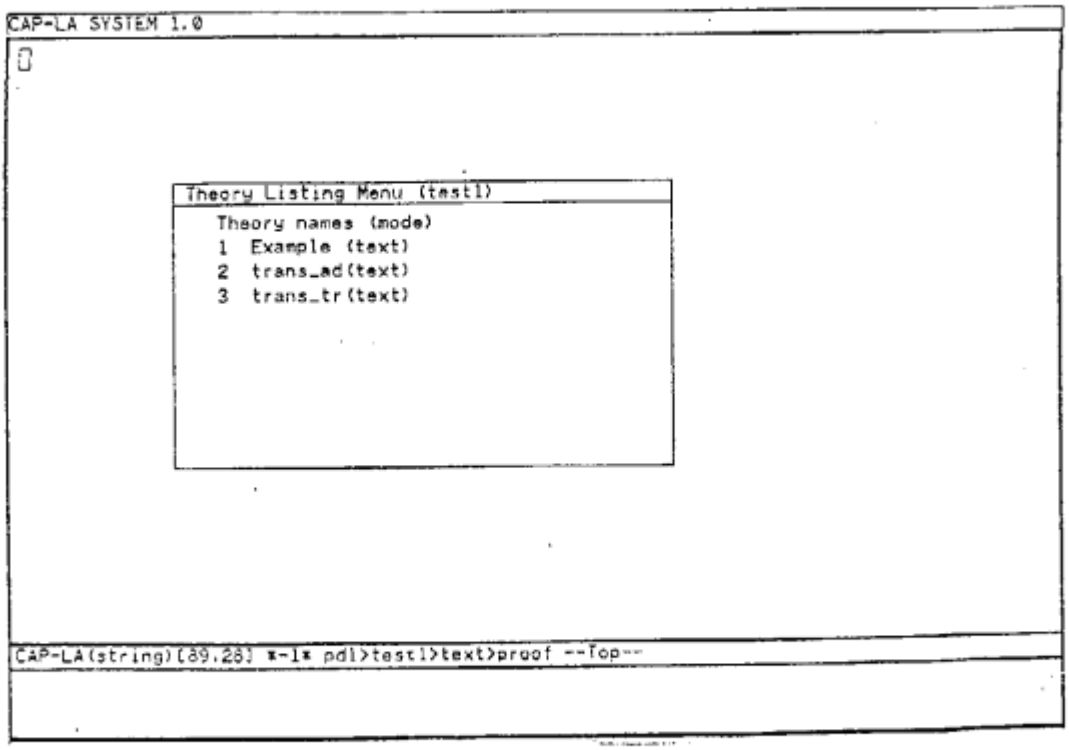
```
    all{A:square}det(A)=det(trans(A))

    proof
      let a:square be arbitrary
      n := col_size(a)
      then n = col_size(trans(a))
      det(a)
      =sum{P:perm<n>}sgn(P)*prod{I:1..n}a[P[I],I]    by definition
      =sum{P:perm<n>}sgn(inv(P))*prod{I:1..n}a[inv(P)[I],I]
      =sum{P:perm<n>}sgn(P)*prod{I:1..n}trans(a)[P[I],I]
      since
        let p:perm<n> be arbitrary
        prod{I:1..n}a[inv(p)[I],I]
        =prod{I:1..n}a[inv(p)[p[I]],p[I]]
        =prod{I:1..n}trans(a)[p[I],I]
        since
          let i:1..n be arbitrary
          a[inv(b)[p[i]],p[i]]
          =a[i,p[i]]
          =trans(a)[p[i],i]
        end_since
        sgn(inv(p)) = sgn(p)
      end_since
      = det(trans(a)) by definition
    end_since
  end_theorem

end_theory
```

Appendix 3. Example Session with the CAP-LA system

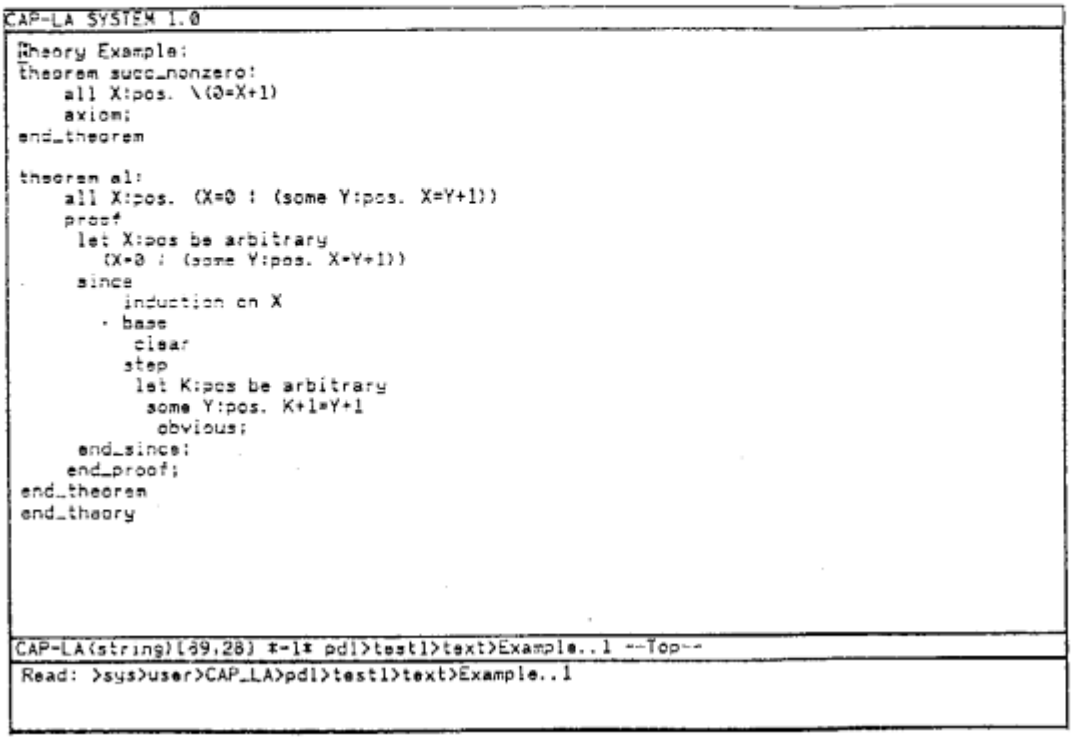
The session proceeds using various windows on a bit-map display.



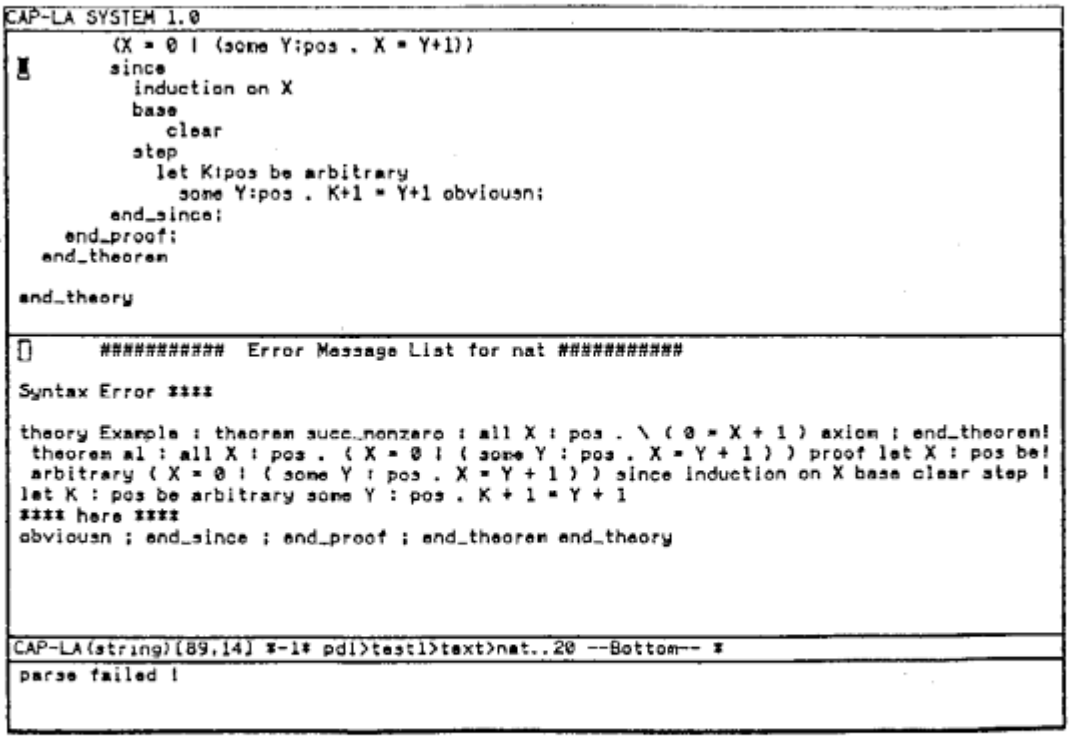
After entering the CAP-LA system, a system menu window appears for function selection. The user clicks the function in the menu to see a list of theories registered under his name and the next window appears.



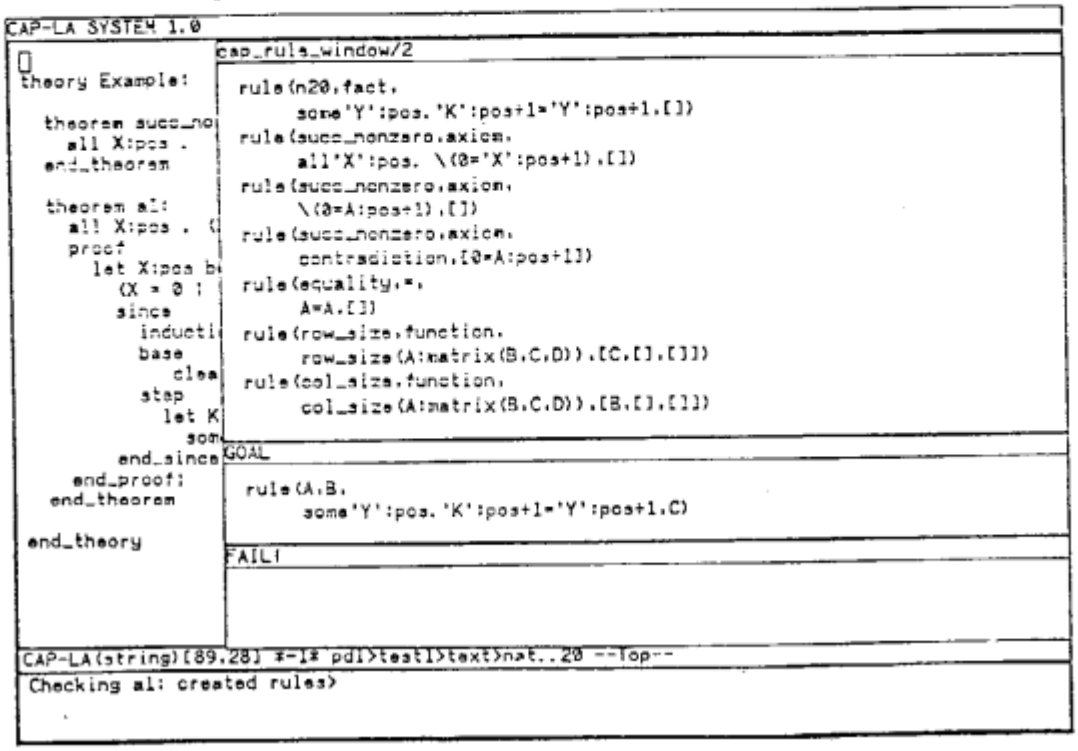
This list is also used for selection of the theory needed. If the theory "Example" is selected, its content appears.



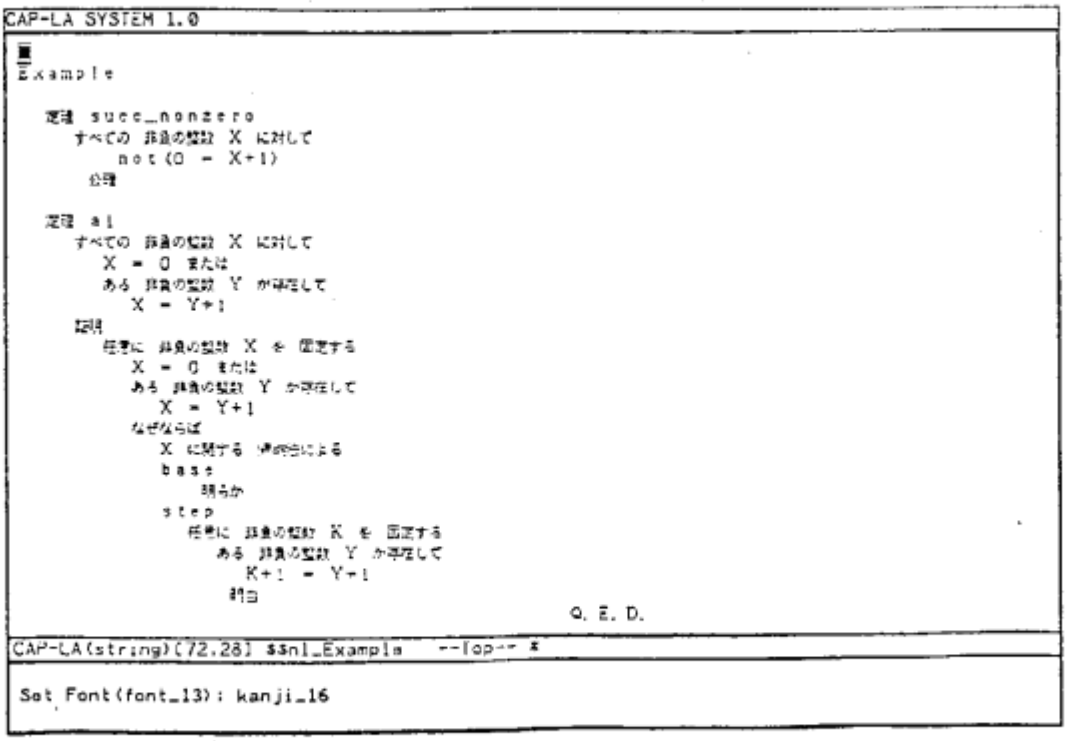
The user wants to check a text after editing and the text is checked against the syntax of PDL. If any errors are found, then the next window appears.



Once all syntactic errors are corrected, correctness of the proof is then checked. The user can trace the process of proof checking through another window if necessary.



Upon completion of proof checking, if a printout of the theorem and the proof is requested in a clean format, the system can print it in an English form or a Japanese form.



100)

Appendix 4. Experiments with Metis

Let us begin with purely algebraic examples. The first example is the word problem of ring theory.

Example 4.1

Metis was given an AC-operator $+$ and a binary operator $*$, (not AC in general) with the following axioms:

(1) $0 + A = A$

(2) $(-A) + A = 0$

(3) $(A * B) * C = A * (B * C)$

(4) $(A + B) * C = A * C + B * C$

(5) $A * (B + C) = A * B + A * C$

We had Metis run the completion procedure in automatic mode. Metis obtained $(A*B)*C = A*(B*C)$ and $0+A = A$ as the first and the second ruling formula and converted them to the rules $(A * B) * C \rightarrow A * (B * C)$ and $0 + A \rightarrow A$, respectively. The third ruling formula $(-A)+A = 0$ could be oriented left to right by the recursive path ordering, if $0 < +$ or $0 < -$. So Metis asked the user which should be introduced.

```
[METIS] -> k
<< Knuth - Bendix  (automatic execution) >>
New Rule is     r1:      (A*B)*C  ->  A*(B*C)
New Rule is     r2:      0+A  ->  A
You can orient -A+A  ->  0 by the following.
  [1]   0 << +
  [2]   0 << -
  else  exit
```

After selecting $0 < +$, we had Metis continue the procedure.

```
select no ?  1
[ 0 << + is asserted.  ]
New Rule is     r3:      -A+A  ->  0
New Rule is     r4:      -(-A)  ->  A
New Rule is     r5:      -(0)  ->  0
Which do you want to orient ?
  [1]   A*(B+C)  ->  A*B+A*C
  [2]   A*B+A*C  ->  A*(B+C)
  else  exit
```

The sixth ruling formula was the left distributive law and it could be oriented to either direction depending on the orderings on function symbols. Since we instructed Metis to convert it to the rule $A * (B + C) \rightarrow A * B + A * C$, the system automatically introduced $+ < -$ as the ordering on function symbols.

```
select no ?  1
[ + << * is asserted.  ]
New Rule is      r6:       A*(B+C)  ->  A*B+A*C
New Rule is      r7:       (A+B)*C  ->  A*C+B*C
New Rule is      r8:       A+ -(B+A)  ->  -B
[ + << - is asserted.  ]
```

The eighth ruling formula can be converted to the rule $-(A + (-B)) \rightarrow B + (-A)$ if and only if $+ < -$. So Metis introduces the ordering without interaction.

```
New Rule is      r9:       -(A+(-B))  ->  B+(-A)
New Rule is      r10:      -(A+B)  ->  -A+(-B)
DELETE           r8
DELETE           r8*
DELETE           r9
New Rule is      r11:      A*0+A*B  ->  A*B
New Rule is      r12:      A*0  ->  0
DELETE           r11
DELETE           r11*
New Rule is      r13:      0*A+B*A  ->  B*A
New Rule is      r14:      0*A  ->  0
DELETE           r13
DELETE           r13*
New Rule is      r15:      (-A)*B+A*B  ->  0
Which do you want to orient ?
  [1]   (-A)*B  ->  -A*B
  [2]   -A*B  ->  (-A)*B
  else  exit
select no ?  1
[ - << * is asserted.  ]
New Rule is      r16:      (-A)*B  ->  -A*B
DELETE           r15
DELETE           r15*
New Rule is      r17:      A*(-B)+A*B  ->  0
New Rule is      r18:      A*(-B)  ->  -A*B
DELETE           r17
DELETE           r17*

   Knuth - Bendix terminated.
```

```
Your system is [ COMPLETE ] .
```

The procedure terminated successfully. Here is the resulting complete TRS for the word problem of rings.

```
[METIS] -> list
<< state listing >>

        "ring"

    operators:
        + / AC  ( multiset ordering )
        0 / 0
        - / 1
        * / 2   ( left to right lexicographic ordering )

    orderings:
        0  <  "+"  <  *,-
        "0"  <  +,*,-
        +,0  <  "-"  <  *
        +,-,0  <  "*"

    equations:
        No equations.

    rules:
        r1:      (A*B)*C  ->  A*(B*C)
        r2:      0+A  ->  A
        r2*:     A+0+B  ->  A+B
        r3:      -A+A  ->  0
        r3*:     A+(-B)+B  ->  A+0
        r4:      -(-A)  ->  A
        r5:      -(0)  ->  0
        r6:      A*(B+C)  ->  A*B+A*C
        r7:      (A+B)*C  ->  A*C+B*C
        r10:     -(A+B)  ->  -A+(-B)
        r12:     A*0  ->  0
        r14:     0*A  ->  0
        r16:     (-A)*B  ->  -A*B
        r18:     A*(-B)  ->  -A*B
```

Huet and Hullot developed a method to prove inductive theorems without explicit induction [Huet 82] using a modified version of the Knuth-Bendix completion procedure. Their method is called inductionless induction and is effective for many theorems which usually require explicit induction.

In order to use the method, ground terms have to be classified into two categories, namely, *constructor terms* which are always irreducible and constructed only of special function symbols called constructors, and *non-constructor terms* which are always reducible and include a function symbol other than constructors. To prove an inductive theorem, we add the statement as an axiom and execute the completion procedure. The statement is an inductive theorem if the process succeeds to completion without yielding any rules to rewrite constructor terms.

Metis was given an ordinary definition of the append operation for two lists and two different definitions of the reverse operation of a list.

```
[METIS] -> list rule
<< state listing >>

        "--- append & reverse ---"

   rules:
        r1:  append([],A)   ->  A       [e3]
        r2:  rev([],A)   ->  A          [e5]
        r3:  reverse([])   ->  []       [e1]
        r4:  append([A|B],C)   ->  [A|append(B,C)]     [e4]
        r5:  rev([A|B],C)   ->  rev(B,[A|C])   [e6]
        r6:  reverse([A|B])   ->  append(reverse(B),[A])       [e2]
```

If we define [_|_] (*cons*) and [] (*nil*) as the constructors, then the above conditions are satisfied. We added an equation $rev(A,[]) = reverse(A)$ and had Metis execute the completion procedure.

```
[METIS] -> kB iNTERACTIVE
<< Knuth - Bendix (interactive execution) >>

Current ruling formula [ CAN ] be oriented.
<<<<     e7:  reverse(A)  =(<>)=  rev(A,[])     >>>>
Which do you want to orient ?
  [1]   reverse(A)  ->  rev(A,[])
  [2]   reverse(A)  <-  rev(A,[])
  else  exit
Which ?  1
[ rev << reverse is asserted.  ]
```

```
Current ruling formula is [ ORIENTED ] .
New Rule is     r7:     reverse(A)  ->  rev(A,[])
DELETE          r3
DELETE          r6

Current ruling formula [ CAN ] be oriented.
<<<<    e8:  rev(A,[B])  =(<>)=  append(rev(A,[]),[B])    >>>>
Which do you want to orient ?
  [1]   rev(A,[B])  ->  append(rev(A,[]),[B])
  [2]   rev(A,[B])  <-  append(rev(A,[]),[B])
  else  exit
Which ?  2
[ rev << append is asserted.  ]

Current ruling formula is [ ORIENTED ] .
New Rule is     r8:     append(rev(A,[]),[B])  ->  rev(A,[B])

Current ruling formula is [ ORIENTED ] .
New Rule is     r9:     append(rev(A,[B]),[C])  ->  rev(A,[B,C])

Current ruling formula is [ ORIENTED ] .
<<<<    e10:  append(rev(A,[B,C]),[D])  =>=  rev(A,[B,C,D])    >>>>
```

Since the current and the former ruling formulas suggested that a new lemma

$$append(rev(A, B), C) = rev(A, append(B, C))$$

would be useful, we added it.

```
[METIS/KB] -> new lEMMA
<< introduce a new lemma >>
Lemma > append(rev(A,B),C) = rev(A,append(B,C)).

Current ruling formula is [ ORIENTED ] .
New Rule is     r10:    append(rev(A,B),C)  ->  rev(A,append(B,C))
DELETE          r8
DELETE          r9

   Knuth - Bendix is terminated.
   Your system is [ COMPLETE ].
```

The completion terminated and, therefore, both the target statement and the lemma inserted on the way were proved to be inductive theorems.

Several examples were taken from the theory of λ-calculus and combinators [Hindley 86, Barendregt 84]. In the theory of combinators, the combinator $\mathbf{K} = \lambda XY.\ X$ and $\mathbf{S} = \lambda XYZ.\ X*Z*(Y*Z)$ (as usual we assume that symbols $*$, standing for function application, are left associative) are called basic combinators because all the λ-terms without free variables can be constructed from S and K only.

Example 4.2

It is well-known that the identity $\mathbf{I} = \lambda X.\ X$ is represented by S*K*K. Metis was given the two axioms $\mathbf{K}*X*Y = X$ and $\mathbf{S}*X*Y*Z = X*Z*(Y*Z)$ for K and S to derive the identity. The problem can be expressed as $\exists I.\forall X.\ I*X = X$. Metis converted its negation to Skolemized form $A*\$1(A) \neq \$1(A)$ ($1 is the so-called Skolem function).

```
[METIS] -> proVE ssTRATEGY TERMINAL
<< prove formulas by S-strategy >>

Formula > some(I,all(X, I*X = X )).

Try to prove formula :  A* $1(A)  =/=  $1(A)
Enter S-strategy...

Current ruling formula is [ INEQUATION ] .
New Rule is     r1:      A* $1(A)  <-/->  $1(A)

Current ruling formula is [ ORIENTED ] .
New Rule is     r2:      k*A*B  ->  A

Current ruling formula is [ INEQUATION ] .
New Rule is     r3:      A  <-/->  $1(k*A)

Current ruling formula is [ NOT ] orientable.
New Rule is     r4:      s*A*B*C  <->  A*C*(B*C)

Current ruling formula is [ ORIENTED ] .
New Rule is     r5:      s*k*A*B  ->  B


   e13:  $1(s*k*A) =/= $1(s*k*A) [r5/r1] is a contradiction.
   Then [ PROVED ].
```

The first ruling formula was the target formula $A*\$1(A) \neq \$1(A)$, and the second was the axiom for K, which was oriented left to right. The third formula was an extended narrowing from the first using the second, since $A = \mathrm{K}*A*\$1(\mathrm{K}*A) \neq \$1(\mathrm{K}*A)$. The fourth was the axiom for S which could not be oriented. The fifth was an extended critical pair between the fourth and the second, since $\mathrm{S}*\mathrm{K}*A*B = \mathrm{K}*B*(A*B) = B$. Using this, a contradictory narrowing was obtained from the first ruling formula. By examining this process, we easily find that all terms of the form $\mathrm{S}*\mathrm{K}*A$ are equal to the identity function, and $\mathrm{S}*\mathrm{K}*\mathrm{K}$ is merely an instance of such terms.

Example 4.3

Next, we made Metis try to prove the fixed-point theorem, i.e. that there exists a fixed-point for any combinator, with the existence of the combinators $\mathrm{B} = \lambda XYZ.\, X*(Y*Z)$ of composition of functions and $\mathrm{M} = \lambda X.\, X*X$ of self-application, which are defined by $\mathrm{B} = \mathrm{S}*(\mathrm{K}*\mathrm{S})*\mathrm{K}$ and $\mathrm{M} = \mathrm{S}*\mathrm{I}*\mathrm{I}$. Metis was given the axioms $\mathrm{B}*X*Y*Z = X*(Y*Z)$ and $\mathrm{M}*X = X*X$. The theorem can be expressed as $\forall F.\exists P.\, F*P = P$.

```
[METIS] -> list all
<< state listing >>
   operators:
        * / 2 ( lexicographic ordering left to right )
        b / 0
        m / 0

   orderings:
        No orderings

   equations:
        e1:      m*A  =  A*A      [axiom]
        e2:      b*A*B*C  =  A*(B*C)      [axiom]

   rules:
        No rules.

[METIS] -> prove sstrategy terminal
<< prove equations by S-strategy >>

Equation > all(F,some(P, F*P = P )).

Try to prove equation :  $1*A  =/=  A
Enter S-strategy...

Current ruling formula is [ INEQUATION ] .
New Rule is      r1:      $1*A  <-/->  A
```

```
Current ruling formula is [ NOT ] orientable.
<<<<    e1:  m*A  =  A*A    >>>>
```

Since the above ruling formula could not be oriented, we let Metis introduce a new function symbol s and rewrite both $A * A$ and $M * A$ to $s(A)$. Acquisition of the new function symbol and orientation of new equations was done interactively as follows:

```
[METIS/PROVE/S-STRA] -> new function
<< introduce a new function >>
Operator?s
[ e3:  m*A  =  s(A) (axiom) is asserted.  ]
[ e4:  A*A  =  s(A) (axiom) is asserted.  ]

Current ruling formula [ CAN ] be oriented.
<<<<    e4:  A*A  =(<>)=  s(A)    >>>>
[METIS/PROVE/S-STRA] -> suggestion current
<< suggestion for ordering >>
Which do you want to orient ?
  [1]   A*A  ->  s(A)
  [2]   A*A  <-  s(A)
  else  exit
Which ?  1
[ s << * is asserted.  ]

Current ruling formula is [ ORIENTED ] .
New Rule is     r2:     A*A  ->  s(A)

Current ruling formula is [ ORIENTED ] .
New Rule is     r3:     m*A  ->  s(A)

Current ruling formula is [ INEQUATION ] .
New Rule is     r4:     s($1)  <-/->  $1

Current ruling formula is [ ORIENTED ] .
New Rule is     r5:     b*A*B*C  ->  A*(B*C)

Current ruling formula is [ ORIENTED ] .
New Rule is     r6:     s(b)*A*B  ->  b*(A*B)

Current ruling formula is [ ORIENTED ] .
New Rule is     r7:     s(b*A)*B  ->  A*(b*A*B)

Current ruling formula is [ ORIENTED ] .
```

```
New Rule is     r8:      A*(B*(b*A*B))  ->  s(b*A*B)

Current ruling formula is [ ORIENTED ] .
New Rule is     r9:      s(s(b))*A  ->  b*(s(b)*A)

Current ruling formula is [ INEQUATION ] .
New Rule is     r10:    s(b*$1*A)  <-/->  A*(b*$1*A)


   e32:  s(b*$1*m) =/= s(b*$1*m) [r3/r10] is a contradiction.
   Then [ PROVED ].
```

Metis finally derived a contradictory inequation. The inequation was obtained by substituting M to A in r10 and rewriting the right hand side by r3. The inequation r10 was from r1 and r8, since

$$s(\mathrm{B} * \$1 * A) = \$1 * (A * (\mathrm{B} * \$1 * A)) \neq A * (\mathrm{B} * \$1 * A).$$

and the rule r8 was from r2 and r5, since

$$A * (B * (\mathrm{B} * A * B)) = \mathrm{B} * A * B * (\mathrm{B} * A * B) = s(\mathrm{B} * A * B).$$

Examining this process of refutation showed us that m*(B*$1*m) is the value substituted to the original variable A in the inequality obtained by the negation of the target formula. In fact, it is a fixed point of $1, since

$$\mathrm{M} * (\mathrm{B} * \$1 * \mathrm{M}) = \mathrm{B} * \$1 * \mathrm{M} * (\mathrm{B} * \$1 * \mathrm{M}) = \$1 * (\mathrm{M} * (\mathrm{B} * \$1 * \mathrm{M}))$$