Metavariable Method for Natural Deduction Within *Theorema*

Boris Konev*

Russian Academy of Sciences
Steklov Institute of Mathematics
St. Petersburg Department
27 Fontanka, St. Petersburg, 191011, Russia
email: konev@pdmi.ras.ru

Abstract

A problem of finding of terms that have to be substituted during the proof search in places of quantified variables (like in the case of $\exists x P(x)$) is a well-known problem of automated theorem proving. In this paper we present an efficient implementation of the metavariable method for Natural Deduction. We show how to find terms when the proof search is performed in AND-OR tree like structures; the implementation does not depend on the search strategy and allows integration with other provers.

1 Introduction

A well-known problem of automated theorem proving is finding of witnessing terms. For instance, in order to prove a statement of the form $\exists x P(x)$, a proof search procedure should find a term to substitute in place of x. The problem can be relatively easy solved by machine-oriented proof systems (like resolution theorem provers) and is not that easy for human-oriented systems (like tableau systems or proofs by Natural Deduction).

The proving style of *Theorema* [4, 6, 5, 10, 8, 9] tends to be close to the proving style of working mathematicians. In fact, the system offers a natural deduction style calculus for higher order logic that reflects the *Theorema* group's view on proving. Objects of the calculus are *proof situations*—sequents of the form

$$A_1, A_2, \ldots, A_n \vdash G,$$

where the formulas $A_1, A_2, \ldots, A_n$ form the list of assumptions and the formula G is the goal. Proof situations are transformed by means of inference rules. Logically, Natural Deduction is a sequent calculus (cf. [15]) where all sequents have the only formula on the right hand side of $\vdash$.

A variety of methods for finding of witnessing terms has been developed for sequent-style calculi [17, 1, 14, 16, 12]. A promising way is to use *metavariables* (in the literature, several other names have been used for the same idea, e.g., *parameters*, *dummy variables*, *free variables*) [21, 23, 17, 3, 2, 13]. The main idea of the method is to introduce a new metavariable and substitute it for a quantified variable, instead guessing terms. Metavariables are instantiated when the process of proof search is finished.

*Supported by INTAS, Project 96-0760 and Austrian Forschunsfoerderungsfonds (FWF), SFB project F1302

An essential part of the method is a procedure that instantiates metavariables. In fact, a substitution that maps metavariables into terms is a unifier for a unification problem constructed during the proof search. The straightforward implementation of the method requires constructing of a substitution after each deduction step and checking if the constructed substitution is correct. In this paper we develop an efficient procedure that constructs a correct substitution.

The *Theorema* system aims at unifying in one system different specialized proof engines like the Knuth-Bendix procedure, the Gröbner Bases computation, proving by induction or simplifier. The system contains a set of independent modules (called *basic provers*) each of which makes one or several steps of proof search. The control procedure combines the output of these modules and forms a usual for artificial intelligence object—`AND-OR` tree presenting both the proof and the trace of proof search (called a *deduction tree*). The user may easily guide the behavior of the system and choose which subset of the predefined set of basic provers to use.

The modulized nature of the system impose restrictions on an implementation the method within *Theorema*. A mechanism that introduces metavariables and then instantiates them should neither depend nor influence other modules of the system. Obviously, this cannot be done in frames of a module, the control procedure should also be involved.

In this paper we present a technique that efficiently implements the metavariable method. The procedure does not influence other basic provers and does not depend on their output. It also does not depend on the search strategy used for constructing of a deduction tree.

Organization of the paper. In Section 2 we present the Natural Deduction calculus. In Section 3 we describe the metavariable method. Section 4 contains a sketch of the proof search mechanism of *Theorema*. Section 5 gives a basic prover for Natural Deduction. In section 6 we present an algorithm that implements the metavariable method.

The *Theorema* system is a result of work of the *Theorema* group (see `http://www.theorema.org`). The control procedure is mainly written by Elena Tomuta [22], the basic prover for Natural Deduction was developed by Bruno Buchberger and Tudor Jebelean [7]. The metavariable method was incorporated to the system by the author.

The main part of the work was done at the Research Institute for Symbolic Computation, Linz, Austria. The author wants to thank the staff of the Institute and, in particular, the *Theorema* group for cordial acceptance and useful cooperation.

2 Natural Deduction

The Natural Deduction calculus corresponds to the human way of proving. Indeed, usually a mathematician tries to prove a statement (theorem, lemma or proposition) basing on a set of previously proved statements. In the Natural Deduction calculus a statement to prove is called a *goal* and a list of known statements is called an *assumption list*. The union of a goal and an assumption list is called a *proof situation*.

Formally we think about proof situations as about *sequents* of the form

$$A_1, A_2, \ldots, A_n \vdash G,$$

where the assumptions $A_1, A_2, \ldots, A_n$ form the assumption list and G is the goal. We usually denote lists of assumptions by capital Greek letters (Γ, Δ, etc.); if we want to refer to a particular formula in an assumption list and do not mention other formulas, we write

$$\Gamma, A_2, \Delta \vdash G.$$

To find out if an assumption list implies a goal, we transform the corresponding proof situation to one or several proof situations (hopefully, simpler ones) until we reduce it to a set of proof situations such that a goal is undoubtedly a consequence of a list of assumptions for each of them. For example, a list of assumptions for sure implies a goal if the goal coincides with one of assumptions. We call such proof situations *final proof situations*. The proof situation from which the proof starts is the *initial proof situation*. Reductions are performed by means of *inference rules* of the form

$$\frac{S_1; \quad S_2; \quad \dots; \quad S_n}{S}\ \mathrm{R},$$

where S and $S_1, \dots, S_n$ are proof situations. We call the proof situations $S_1, \dots, S_n$ the *premises* of the rule R and the proof situation S the *conclusion*.

A proof by Natural Deduction, informally described above, is a tree whose nodes are marked by proof situations, the root is marked by the initial proof situation and the leaves are marked by final proof situations. In addition, the mark of each non-leaf node is the result of application of some inference rule to the marks of its immediate successors. In this paper we consider just the first-order fragment of Natural Deduction; the inference rules for it are given in Section 5.

In general, a deduction system is suitable for automated reasoning if its rules are *reversible*, that is, one should be able to uniquely construct the premises of a deduction rule given a conclusion. It is not hard to see that all inference rules of Natural Deduction that introduce propositional connectives, like

$$\frac{\Gamma \vdash G_1; \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \wedge G_2},$$

are reversible. However, the situation differs for quantifier rules.

3 Metavariable method

Let us look at the classical rules that introduce universal quantifiers into an assumption list and existential quantifiers into a goal (cf. [15]).

$$\frac{\Gamma \vdash [F]_{x \to t} \vee \exists x F}{\Gamma \vdash \exists x F}, \qquad \frac{\Gamma, [F]_{x \to t} \wedge \forall x A, \Delta \vdash G}{\Gamma, \forall x A, \Delta \vdash G}, \tag{1}$$

where t is a term, x is an object variable, t is free for x in F, and $[F]_{x \to t}$ is the result of substituting of the term t in places of all free occurrences of x in F.

It is clear that to compute the premises of a given proof situation with an existential goal (or a universal assumption) *a technique* is needed that generates appropriate terms to be substituted in places of variables. The main idea of the metavariable method is to postpone finding of concrete terms until the moment when the process of proof construction is finished. Instead of replacing quantified variable by a term, that has to be "guessed", a new metavariable is substituted for it. The value of the metavariable will be calculated later. Then the inference rules (1) are replaced by

$$\frac{\Gamma \vdash [F]_{x \to \xi} \vee \exists x F}{\Gamma \vdash \exists x F}, \qquad \frac{\Gamma, [F]_{x \to \xi} \wedge \forall x A, \Delta \vdash G}{\Gamma, \forall x A, \Delta \vdash G}, \tag{2}$$

where ξ is a new metavariable.

The initial proof situation is transformed by means of inference rules exactly as described above, forming a tree that we call a *preproof*. The only difference between proofs and preproofs is that preproofs may contain metavariables. A preproof is called *correct* if there exist a substitution of terms in places of metavariables such that it transforms the preproof into a real proof. We call such a substitution a *correct* substitution. Thus, the problem of proof search can be decomposed into finding of a correct preproof and a correct substitution.

Note that not every substitution makes a preproof a proof. First, a correct substitution should map the leaves of a preproof into final proof situations. Second, after applying of an arbitrary substitution, some inference rules can become invalid. Namely, the conclusion of an applications of one of the rules that introduce existential quantifiers into an assumption list and universal quantifiers into a goal

$$\frac{\Gamma \vdash [F]_{x \to a}}{\Gamma \vdash \forall x F}, \qquad \frac{\Gamma, [F]_{x \to a}, \Delta \vdash G}{\Gamma, \exists x A, \Delta \vdash G}, \tag{3}$$

should not contain the Skolem constant a.

The first requirement shows the way how we can find a correct substitution, the second says about restrictions on it.

It is not hard to see that the leaf of a preproof is transformable into a final proof situation if and only if one of assumptions A_i is unifiable with a goal G ("the goal is an assumption") or an assumption A_i is unifiable with the negation of an assumption A_j ("assumptions are contradictory"). The union of all such pairs of formulas forms a unification problem for the preproof.

Suppose that the conclusion of an application of one of the rules (3) contains a metavariable ζ. Then one should require that the value of a correct substitution on ζ does not contain a. Similarly for the rules (2), if a quantified variable x occurs to the range of a quantifier Qy (where Q is one of $\forall$, $\exists$), then the value of a correct substitution on ξ should not contain y. We denote the described type of restrictions by

$$t \notin \xi, \tag{4}$$

where t is a term, ξ is a metavariable. We say that a substitution σ *satisfies* the restriction if t does not occur to $\xi\sigma$. The union of the sets of restrictions corresponding to each application of a quantifier rule forms a set of restrictions for the preproof.

It was independently proved by Orevkov [20], and Krajíček and Pudlák [18] that

1. if the preproof is correct, there exist a unifier for the unification problem that satisfies the set of restrictions;
2. any unifier for the unification problem that satisfies the set of restriction is a correct substitution;
3. if the preproof is not correct, the unification problem does not have a solution or it's unifier does not satisfy the restrictions;
4. if some unifier satisfies the set of restrictions, a most general unifier also satisfies the set of restrictions.

So we see that it suffices to find a correct preproof, the substitution then can be computed "free of charge". The main disadvantage of the metavariable method is that the substitution can be computed only *after* finding a preproof. It follows, it is difficult to check if a preproof is correct ad hoc; we need special techniques to perform preproof constructing and checking its correctness.

4 Proof search procedure of *Theorema*

The proof search procedure of *Theorema* is described in details in [11, 22]. In this section we give a sketch of how it works and stress on the points that need to be changed to implement the metavariable method.

As we said above, while constructing a proof, a proof situation is reduced by a deduction rule to several simpler proof situations. However, when the actual proof search is performed, we do not know in advance which deduction rule to apply if several of them can be applied simultaneously. It is well possible that while searching for a proof we must examine several or all such alternatives before we find the right way. Note that if we succeed in proving any of this alternatives, we succeed in proving our proof situation.

So, we have reductions of two types. The first, when a proof situation is proved if *all* resulting proof situations are proved and the second, when a proof situation is proved if *at least one* of the resulting proof situations is proved.

These cases are represented in *Theorema* by means of *deduction trees*. A deduction tree contains nodes of two types—`AND` and `OR` (in [22] nodes of `IF` type were also introduced but they are not actually used till now, so we do not pay attention to them in this paper). Nodes of a deduction tree correspond to steps of proof search in such a way that `AND` nodes correspond to the first case, and `OR` nodes—to the second one. The evident advantage of usage of deduction trees is that they represent both a proof and the trace of actions that were performed during the proof search.

Theorema aims at unifying in one system specialized proof engines, like the Knuth-Bendix procedure, the Gröbner Bases computation, proving by induction or simplifier. So the proof search procedure is divided to two parts. The first is the set of *basic provers* that implement different specialized proof engines, the other is the control procedure. Each basic prover is organized as a module that takes a proof situation and generates a deduction tree representing one or several steps of proof. The root of the constructed deduction tree corresponds to the proof situation passed to the basic prover. The control procedure combines deduction trees generated by different basic provers into one deduction tree.

The communication between provers is realized by means of the global and local *proof context*. The global proof context is a collection of facts common to the whole deduction tree. The local proof context is a collection of facts that is associated with a node and is transmitted from a node to its successors. The local context describes properties of proof situations.

A *proof value* is assigned to each node of a deduction tree. A proof value of a node can be either *proved*, that is, the proof search procedure succeeded in proving the corresponding proof situation, *failed*, that means that the proof search procedure failed to prove the corresponding proof situation, or *pending* otherwise.

The proof search procedure starts with the deduction tree T consisting of a single `AND` node corresponding to the initial proof situation. The proof value of the node is *pending*. Then the control procedure proceeds by constructing the deduction tree as follows.

The procedure takes a leaf L of T having the *pending* proof value such that no predecessor of L has the *proved* proof value, the choice of L is determined by the search strategy. The condition on the choice of L follows that we do not try to prove again an already proved `OR` node.

As we said above, each basic prover that can proceed with the proof situation corresponding to L, returns a deduction tree. Let k of all available basic provers $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n$ are applicable to the proof situation and let the deduction trees $T_{i_1}, T_{i_2}, \dots, T_{i_k}$ are built by $\mathcal{P}_{i_1}, \mathcal{P}_{i_2}, \dots, \mathcal{P}_{i_k}$ respectively.

The control procedure proceeds with L depending on the number of applicable provers.

If $k = 0$, the proof value of L is set to *failed*.

If $k = 1$, the control procedure replaces L with T_{i_1}.

If $k > 1$, an `OR` node is created with successors $T_{i_1}, T_{i_2}, \dots, T_{i_k}$; we denote the resulting tree by T'. The control procedure replaces the leaf L with T'.

After this step, the proof values of nodes of T might have changed. The control procedure *propagates* the proof values from leaves to root according to the following rules

$$\Pi(\mathtt{AND}, p_1, p_2, \dots, p_k) = \begin{cases} \mathit{proved} & \text{if } \forall i \in \overline{1,k},\ p_i = \mathit{proved} \\ \mathit{failed} & \text{if } \exists i \in \overline{1,k},\ p_i = \mathit{failed} \\ \mathit{pending} & \text{otherwise} \end{cases};$$

$$\Pi(\mathtt{OR}, p_1, p_2, \dots, p_k) = \begin{cases} \mathit{proved} & \text{if } \exists i \in \overline{1,k},\ p_i = \mathit{proved} \\ \mathit{failed} & \text{if } \forall i \in \overline{1,k},\ p_i = \mathit{failed} \\ \mathit{pending} & \text{otherwise} \end{cases},$$

where $p_1, p_2, \dots, p_n$ are the proof values of the immediate successors of an `AND` (respectively, `OR`) node.

While the proof value of the root of T is *pending* the next leaf is chosen and the control procedure continues building the deduction tree T.

The described procedure works fine when a basic prover can determine that a proof situation is a final proof situation. For the propositional case and some other cases this is true whereas for the metavariable method it is not.

The point is that acting in the described way, the proof search procedure can construct just a preproof. The proof is the result of substituting of metavariables with concrete terms. The substitution can be easily find after the preproof is constructed, but not knowing the substitution one cannot say if the leaf of a preproof will be transformed into a final proof situation.

We demonstrate this statement by the following example. Let us consider the deduction tree (Figure 1) for

$$P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x)). \tag{5}$$

The first two steps are evident—we introduce a new metavariable ξ and split the conjunction in the goal. On the left branch we get the proof situation $P(a), P(b), Q(c), Q(b) \vdash P(\xi)$. It can be transformed into a final proof situation by substitutions $\sigma_1 : \xi \mapsto a$ and $\sigma_2 : \xi \mapsto b$. At this point we do not know which substitution is correct, so we generate an `OR` node. Similarly on the right branch, the proof situation $P(a), P(b), Q(c), Q(b) \vdash Q(\xi)$ can be transformed into a final proof situation by $\sigma_3 : \xi \mapsto c$ and $\sigma_4 : \xi \mapsto b$.

For our example, the required substitution σ coincides with σ_2 and σ_4. Having computed, σ we see that the proof value of the leaves 1 and 3 should be *failed*, so the reductions marked by numbers 4 and 6 are useless.

The presented at Figure 1 deduction tree does not have nodes with the *pending* proof value. It corresponds to *breadth first search*. From the practical point of view *depth first* search is more attractive as the search procedure might find the desired branch before all possible alternatives will be examined.

At Figure 2 we present a deduction tree constructed by depth first search for the same initial proof situation. Nodes 2 and 3 have the *pending* proof values and are marked by "?", as the proof

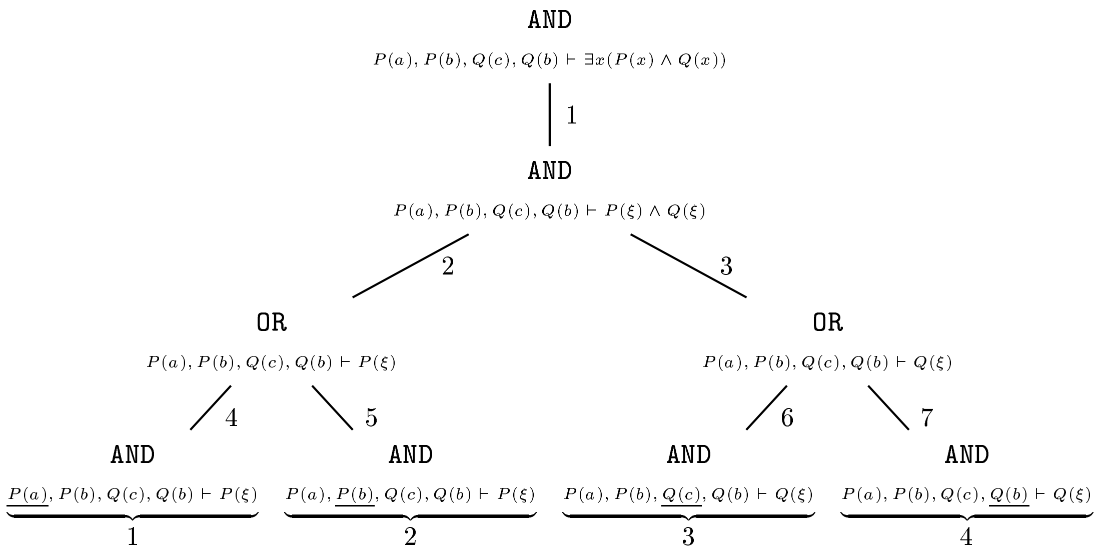

Figure 1: Deduction tree for $P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x))$.

search procedure did not yet examine them and no information is known till now. Indeed, the substitution $\xi \mapsto a$ makes the proof situation corresponding to the node 1 a final proof situation. Before we realize that this substitution is not correct, there is no need to proceed with the proof situation corresponding to the leaf 2. At the presented moment, the control procedure should determine that the built preproof is not finished and continue processing with nodes 2 and 3.

In the Section 6 we develop a procedure that constructs preproofs independently on the proof strategy.

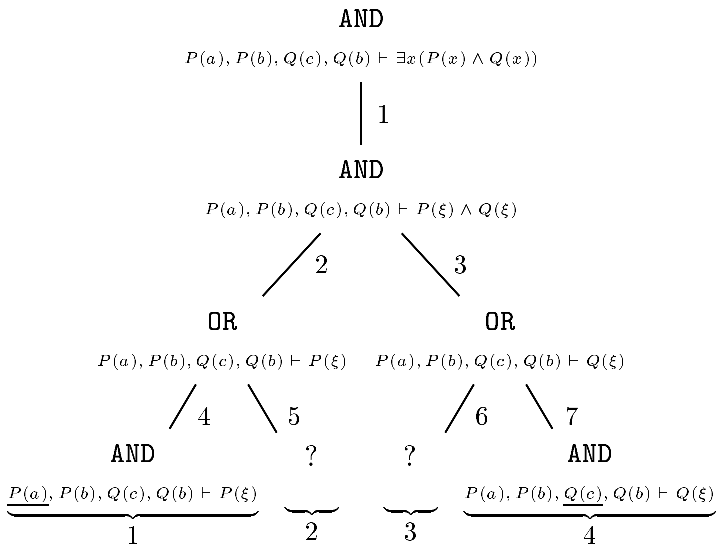

Figure 2: Deduction tree for $P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x))$ constructed by depth first search.

5 Basic prover for first-order logic

In this section we describe the basic prover that makes one step of proof search for first-order logic. We call it `PND` (Proof by Natural Deduction).

Note that the `PND` prover that is used in practice contains much more inference rules than we give in this section. All these rules are used to simplify proof search and improve readability of proofs. We describe below just a subset of all rules that is necessary for the proof search procedure be complete.

`PND` takes as parameters a proof situation $\Gamma \vdash G$ and the local context `LC`. Among other facts, `PND` keeps in the local context a set of restrictions of the form (4) and pairs of formulas.

At first, `PND` checks whether the given proof situation can be transformed to a final proof situation by substituting of terms in places of metavariables. This is true in the following cases.

- There exists a unifier for the goal G and one of assumptions A_i. Let C_1 be G and C_2 be A_i.
- There exist two assumptions A_i and A_j, $i < j$, such that A_i and $\neg A_j$ have a unifier. Let C_1 be A_i and C_2 be $\neg A_j$.
- There exist two assumptions A_i and A_j, $i < j$, such that $\neg A_j$ and A_i have a unifier. Let C_1 be $\neg A_i$ and C_2 be A_j.

If the pair (C_1, C_2) does not occur to the local proof context and a most general unifier σ for C_1 and C_2 satisfies the set of restrictions that is kept in the local proof context, the output of `PND` is the deduction tree consisting of an `OR` node with two successors. The first is the `AND` node with the *proved* proof value. The node is supplied with the unifier σ. The proof situation corresponding to the node is the proof situation passed to `PND`.

The second successor is the `AND` node with the *pending* proof value. The proof situation corresponding to the node is also the proof situation passed to `PND`. The pair (C_1, C_2) is added to the local context, so this proof situation will never again be considered as a final proof situation because of formulas C_1 and C_2.

The resulting deduction tree is presented at Figure 3. Informally, in this case the basic prover tries a substitution and keeps the possibility to go further if it is not correct.

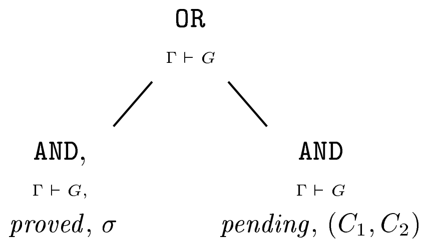

Figure 3: The given proof situation is a final proof situation.

If the given proof situation cannot be transformed to a final proof situation, `PND` tries to apply to it the following rules one-by-one in the order the rules are written. If one of them is applicable, `PND` returns the corresponding deduction tree.

- Rules that do not increase the number of branches in the deduction tree:

$$\frac{\Gamma, \neg G_2 \vdash G_1}{\Gamma \vdash G_1 \vee G_2}; \quad \frac{\Gamma, G_1 \vdash G_2}{\Gamma \vdash G_1 \implies G_2}; \quad \frac{\Gamma, \neg A \vee B, \Gamma' \vdash G}{\Gamma, A \implies B, \Gamma' \vdash G}; \quad \frac{\Gamma, A, B, \Gamma' \vdash G}{\Gamma, A \wedge B, \Gamma' \vdash G};$$

$$\frac{\Gamma, G \vdash \mathit{false}}{\Gamma \vdash \neg G}; \quad \frac{\Gamma, A, \Gamma' \vdash G}{\Gamma, \neg\neg A, \Gamma' \vdash G}; \quad \frac{\Gamma, \neg A \vee \neg B, \Gamma' \vdash G}{\Gamma, \neg(A \wedge B), \Gamma' \vdash G}; \quad \frac{\Gamma, \neg A \wedge \neg B, \Gamma' \vdash G}{\Gamma, \neg(A \vee B), \Gamma' \vdash G};$$

$$\frac{\Gamma, A \wedge \neg B, \Gamma' \vdash G}{\Gamma, \neg(A \implies B), \Gamma' \vdash G}; \quad \frac{\Gamma, \forall x_1, \ldots, x_n \neg A, \Gamma' \vdash G}{\Gamma, \neg(\exists x_1, \ldots, x_n A), \Gamma' \vdash G}; \quad \frac{\Gamma, \exists x_1, \ldots, x_n \neg A, \Gamma' \vdash G}{\Gamma, \neg(\forall x_1, \ldots, x_n A), \Gamma' \vdash G}.$$

  The local proof context is kept intact.

- Rules that do increase the number of branches in the deduction tree:

$$\frac{\Gamma, A, \Gamma' \vdash G; \quad \Gamma, B, \Gamma' \vdash G}{\Gamma, A \vee B, \Gamma' \vdash G}; \quad \frac{\Gamma \vdash G_1; \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \wedge G_2}.$$

  The local proof context is kept intact.

- Rules that introduce Skolem constants:

$$\frac{\Gamma \vdash [G]_{x_1 \to c_1, \ldots, x_n \to c_n}}{\Gamma \vdash \forall x_1, \ldots, x_n G}; \quad \frac{\Gamma, [A]_{x_1 \to c_1, \ldots, x_n \to c_n}, \Gamma' \vdash G}{\Gamma, \exists x_1, \ldots, x_n A, \Gamma' \vdash G},$$

  where $c_1, \ldots c_n$ are Skolem constants new to the deduction tree. For each metavariable ξ that occurs to the input proof situation we add to the local proof context the following restrictions

$$c_1 \notin \xi, \ldots, c_n \notin \xi.$$

  These restrictions guarantee that the Skolem constants do not occur to the conclusion of the application of the rule.

- Rules that introduce metavariables:

$$\frac{\Gamma \vdash [G]_{x_1 \to \xi_1, \ldots, x_n \to \xi_n} \vee \exists x_1, \ldots, x_n G}{\Gamma \vdash \exists x_1, \ldots, x_n G}; \quad \frac{\Gamma, [A]_{x_1 \to \xi_1, \ldots, x_n \to \xi_n} \wedge \forall x_1, \ldots, x_n A, \Gamma' \vdash G}{\Gamma, \forall x_1, \ldots, x_n A, \Gamma' \vdash G},$$

  where $\xi_1, \ldots, \xi_n$ are metavariables new to the deduction tree. For each quantified variable y such that x_j is in the range of the quantifier over y, we add to the local proof context the following restriction

$$y \notin \xi_j.$$

  This restriction guarantees that ξ_j is free for x_j.

If none of these rules is applicable, `PND` returns "not applicable".

6 Algorithm

Before we describe the general algorithm, let us consider a particular case. Suppose that we are given a deduction tree constructed by the procedure described in Section 4 and the only basic prover `PND`. Suppose that the deduction tree does not contain `OR` nodes (or, we can think that a branch has been chosen at each `OR` node). The deduction tree is finished, if there exist a substitution σ that maps each leaf to a final proof situation (recall that we call such a substitution a correct substitution).

Let L be a leaf of the deduction tree supplied with a substitution λ and let denote by R a set of restrictions of the form (4) that are kept in the local proof context of the leaf. It is not hard to see that λ is the least restrictive (in other words, the most general) substitution mapping the proof situation of L to a final proof situation. Since λ is a most general unifier, σ is a composition of λ and a certain substitution μ, $\sigma = \lambda\mu$.

Note that `PND` constructs the set of restrictions exactly as described in Section 3. Thus, R is a subset of the set of restrictions corresponding to the whole preproof. If σ satisfies R, λ also does.

So we see that the substitution and the set of restrictions that `PND` assigns to each leaf of the deduction tree are the minimal requirements that a correct substitution should satisfy. In fact, the substitution σ can be computed inductively going down-up along the tree.

If the substitution λ satisfies the set R, it is correct for the tree consisting of L.

Let us do one more step. Consider a node N of the deduction tree whose immediate successors are the leaves $L_1, L_2, \ldots, L_r$ and that does not have non-leaf successors. Let `PND` supplies L_i with a substitution λ_i; the local proof context of L_i contains a set of restrictions R_i. The reasoning similar to the reasoning above shows that the correct substitution σ_N for the subtree having N as the root is a composition of λ_i and a certain substitution μ_i for every $i: 1 \leq i \leq r$, that is,

$$\sigma_N = \lambda_1\mu_1 = \lambda_2\mu_2 = \cdots = \lambda_r\mu_r. \tag{6}$$

The substitution σ_N should satisfy each of the sets $R_1, \ldots, R_r$, that is, their union $\cup_i R_i$.

Note that if $\mu_1, \ldots, \mu_r$ satisfy (6), then for any substitution ν, substitutions $\mu_1\nu, \ldots, \mu_r\nu$ also satisfy (6). We are interested in most general substitutions $\mu_1, \ldots, \mu_r$ (it follows, the substitution σ be most general).

To do this, we represent a substitution as a set of equations as it is done in the Martelli-Montanary unification algorithm [19], $\xi_1 = T_1; \ldots; \xi_s = T_s$, where ξ_i are metavariables, T_i are terms. In fact, the term T_i is the value of the substitution on ξ_i. It is not hard to see that if we present all substitutions $\lambda_1, \ldots, \lambda_r$ by systems of equations and run the Martelli-Montanary procedure on the union of these systems, we get the equational presentation of σ_N. Since σ_N is a most general unifier, the substitutions $\mu_1, \ldots, \mu_r$ are also most general. If the system has no unifier, there are no substitutions $\mu_1, \ldots, \mu_r$ satisfying (6).

We compute a correct substitution by the following function.

Function `Simple_Combine`

Input: A list of pairs $\{(\lambda_1, R_1), (\lambda_2, R_2), \ldots, (\lambda_r, R_r)\}$.

Output: A pair (σ_N, R_N) or `NULL`.

Method:

- Present each substitution λ_i as a system of equations E_i.

- Find a most general unifier σ_N for $\cup_i E_i$.
- If σ_N exists and satisfies $\cup_i R_i$, then return $(\sigma_N, \cup_i R_i)$ else return `NULL`.

□

The output of the function is either `NULL`, that is, there is no correct substitution for the subtree, or a pair (σ_N, R_N). Thus, the nodes that are one level "upper" than leaves are now supplied with substitutions and sets of restrictions as well. Then we can apply the function `Simple_Combine` to the next level. If finally `Simple_Combine` computes a pair $(\sigma_{root}, R_{root})$ for the root of the deduction tree, the substitution σ_{root} is correct for the whole tree.

Consider now the case, when the deduction tree may contain `OR` nodes. Let N be such an `OR` node that all nodes below are `AND` nodes, that is, N is a "first from the bottom" `OR` node. Let us choose among all immediate successors of N nodes $N_{i_1}, \ldots, N_{i_q}$ having the *proved* proof value and consider corresponding subtrees $T_{i_1}, \ldots, T_{i_q}$. The trees T_i do not contain `OR` nodes, so for each of them we can construct a correct substitution λ_i. For the node N we generate not a single pair (σ_N, R_N), but a set of pairs $\{(\lambda_{i_1}, R_{i_1}), (\lambda_{i_2}, R_{i_2}), \ldots, (\lambda_{i_q}, R_{i_q})\}$. Each substitution, occurring to this construction is correct for the subtree having the `OR` node as the root.

The `OR` node may also have immediate successors having the *pending* proof value; no substitution is assigned to them. If a node has the *pending* proof value, constructing of the deduction tree can be continued somewhere under this node until the value of the node becomes *proved* or *failed*. It well may be that the deduction tree should be expanded in this way to finish the proof. So we keep pointers to the nodes having the *pending* proof value, to return to them when all other possibilities are exhausted.

Now we a ready to describe the general case. To each node we add a *system*—a set whose elements are either pairs of the form (σ_N, R_N) or pointers to nodes of the constructed deduction tree. Let functions `Pairs`() and `Pointers`() choose from a system its parts of the first and the second type respectively. A node is *proved* if the system assigned to it contains a pair. Indeed, a substitution occurring to the pair is correct for the corresponding subtree. Thus, systems play the key role in manipulating with proof values, in fact, we compute proof values and systems by means of one function `Compute_Value`. This function is a counterpart of the function Π, Section 4.

To describe the function `Compute_Value` we need two auxiliary functions `CombineOR` and `CombineAND` that compute a system associated with `OR` and `AND` nodes of the deduction tree.

For an `OR` node the system is the union of systems of its immediate successors.

Function `CombineOR`

Input: A list of systems $\{S_1, \ldots, S_r\}$.

Output: A system S.

Method:

- Return the union of $S_1, \ldots, S_r$.

□

If all immediate successors of an `AND` node are proved, a system is assigned to each of them. Let denote these systems by $S_1, \ldots, S_r$. Each substitution occurring to `Pairs`(S_i) is correct for the corresponding subtree. At first, `CombineAND` chooses an element s_i of `Pairs`(S_i) for all $i : 1 \leq i \leq r$, and passes $s_1, \ldots, s_r$ to the function `Simple_Combine`. From the description of `Simple_Combine` we see that if `Simple_Combine` returns not `NULL`, its output contains a correct substitution for the tree having the `AND` node as the root. `CombineAND` tries all possible choices of s_i for all i. In fact, the sequences $(s_1, \ldots, s_r)$ are elements of the Cartesian product of $\texttt{Pairs}(S_1), \ldots, \texttt{Pairs}(S_r)$.

If no system can be obtained in this way, the function returns the union of sets of pointers occurring to the input systems. The proof search procedure can expand further the deduction tree starting from these nodes, then new leaves with the *proved* proof value may appear.

If there is no pointers, there is no hope to prove the node, the function returns `NULL`.

Function `CombineAND`

Input: A list of systems $\{S_1, \ldots, S_r\}$.

Output: A system $\{(\sigma_1, R_1), \ldots, (\sigma_M, R_M), p_1, \ldots, p_N\}$ or a list of pointers $\{p_1, \ldots, p_N\}$ or `NULL`.

Method:

- Let $S = \emptyset$.
- Let $P = \texttt{Pointers}(S_1) \cup \cdots \cup \texttt{Pointers}(S_r)$.
- For all sequences $(s_1, s_2, \ldots, s_r) \in \texttt{Pairs}(S_1) \times \texttt{Pairs}(S_2) \times \cdots \times \texttt{Pairs}(S_r)$
  - if $\texttt{Simple\_Combine}(s_1, \ldots, s_r) \neq$ `NULL`, add the output of $\texttt{Simple\_Combine}(s_1, \ldots, s_r)$ to S.
- If $S = \emptyset$ and $P = \emptyset$, return `NULL`,
- else if $S = \emptyset$, return P,
- else return $S \cup P$.

□

The function `Compute_Value` combines the computation of proof values with the computation of systems. In addition, it builds the global list `NList` of nodes. Function `CombineAND` may return a set of pointers, these are pointers to nodes starting from which building of the deduction tree should be continued. All such nodes go to the list `NList`.

Function `Compute_Value`

Input: A node N having immediate successors $N_1, \ldots, N_t$.

Output: A proof value and a system assigned to N.

Side effect: Changes the global list `NList`.

Method:

- If N is an `AND` node
  - If the proof value of one of $N_1, \ldots, N_t$ is *failed*, set the proof value of N to *failed* and exit.
  - If the proof value of one of $N_1, \ldots, N_t$ is *pending*, set the proof value of N to *pending* and exit.
  - Let $S_1, \ldots, S_t$ be the systems associated with $N_1, \ldots, N_t$ respectively. Let $S_{\texttt{AND}} = \texttt{CombineAND}(S_1, \ldots, S_t)$.

* if $\texttt{Pairs}(S_{\texttt{AND}}) = \emptyset$ and $\texttt{Pointers}(S_{\texttt{AND}}) = \emptyset$, then set the proof value of N to *failed* and exit.
* if $\texttt{Pairs}(S_{\texttt{AND}}) = \emptyset$ and $\texttt{Pointers}(S_{\texttt{AND}}) \neq \emptyset$, then set the proof value of N to *pending*, add $\texttt{Pointers}(S_{\texttt{AND}})$ to `NList` and exit.
* if $\texttt{Pairs}(S_{\texttt{AND}}) \neq \emptyset$ and $\texttt{Pointers}(S_{\texttt{AND}}) \neq \emptyset$, then set the proof value of N to *proved*, associate with N the system $S_{\texttt{AND}}$ and exit.

- If N is an `OR` node
  - If the proof values of all of $N_1, \ldots, N_t$ is *failed*, set the proof value of N to *failed* and exit.
  - If the proof values of all of $N_1, \ldots, N_t$ is *pending*, set the proof value of N to *pending* and exit.
  - If the proof value of one of $N_1, \ldots, N_t$ is *proved*,
    * Let $N'_{i_1}, \ldots, N'_{i_j}$ be those of the nodes $N_1, \ldots, N_t$ who have the *proved* proof value. Let $S_{i_1}, \ldots, S_{i_j}$ be the systems associated with $N'_{i_1}, \ldots, N'_{i_j}$ respectively.
    * Let $N''_{k_1}, \ldots, N''_{k_l}$ be those of the nodes $N_1, \ldots, N_t$ who have the *pending* proof value. Let $p_{k_1}, \ldots p_{k_l}$ be the pointers to the nodes $N''_{k_1}, \ldots, N''_{k_l}$ respectively. Let T be the system $\{p_{k_1}, \ldots p_{k_l}\}$.
    * Set the proof value of N to *proved*, associate with it the system $\texttt{CombineOR}(S_{i_1}, \ldots, S_{i_j}, T)$ and exit.

□

Finally, we present the function `Proof_Search`. It differs from the procedure described in Section 4 in the following. First, it manipulates not just with proof values, but also with systems. The second is the way how the function chooses the leaf L having the *pending* proof value. It can be either a leaf such that no its predecessor has the *proved* proof value (as it was in Section 4), or a successor of one of the members of `NList`. The later case corresponds to the situation when there exist an `AND` node in the constructed part of the deduction tree that can be proved only if the tree be expanded.

Some basic provers (as `PND`) assign systems to nodes. For compatibility, if a basic prover assigns to a node just a proof value, we associate with this node the empty system $\mathcal{E} = \{\epsilon, \emptyset\}$, where ϵ in the empty substitution. The empty system can be combined with any system.

Function `Proof_Search`

Input: Initial proof situation $\Gamma \vdash G$.

Output: If $\Gamma \vdash G$ is valid, a deduction tree T presenting its proof.

Method:

- Construct the deduction tree T consisting of a single `AND` node corresponding to the initial proof situation.
- Let $\texttt{NList} = \emptyset$.
- While the proof value of the root of T is *pending* do

– Choose a leaf L of T having the *pending* proof value such that one of the following conditions is satisfied:
  1. no predecessor of L has the *proved* proof value;
  2. L is a successor of one of the nodes occurring to `NList`.

  The choice of L is determined by the search strategy.
– If no basic prover is applicable to L, set the proof value of L to *failed*.
– If only one basic prover $\mathcal{P}_i$ is applicable to L, replace L with the output of $\mathcal{P}_i$.
– If several basic provers $\mathcal{P}_{j_1}, \ldots, \mathcal{P}_{j_l}$ are applicable to L, form an `OR` node whose successors are generated by $\mathcal{P}_{j_1}, \ldots, \mathcal{P}_{j_l}$. Replace L with the newly formed tree.
– Propagate proof values from leaves to root applying inductively the function `Compute_Value`. During the propagation, the list `NList` may have changed.
– Exclude from the list `NList` all nodes that do not have successors with the pending proof value.

□

The described procedure may not terminate, in practice we run it space- and/or time-constrained.

We return to our example to illustrate the work of the procedure. At Figure 4 we present a deduction tree for $P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x))$ where we mention proof values and associated systems. At the picture, p_2 and p_3 are pointers to the leaves 2 and 3 respectively. At

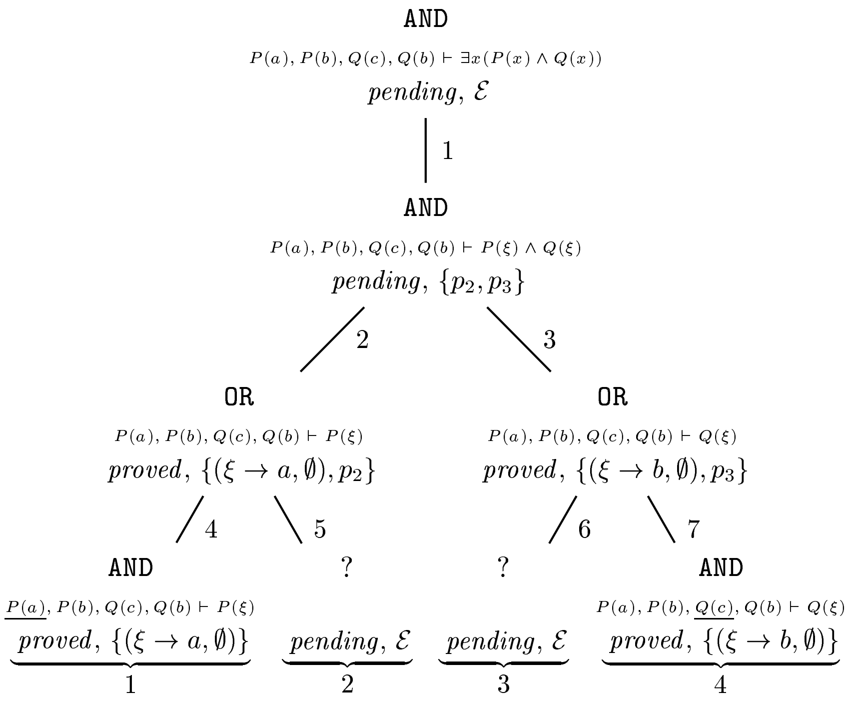

Figure 4: Deduction tree for $P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x))$ with proof values and systems.

the presented moment, the list `NList` contains nodes 2 and 3. Hence, the procedure chooses as L one of the leaves 2 and 3. Suppose that the procedure chooses the leaf 2. After the procedure expands the deduction tree, we get the deduction tree presented at Figure 5. This deduction tree is

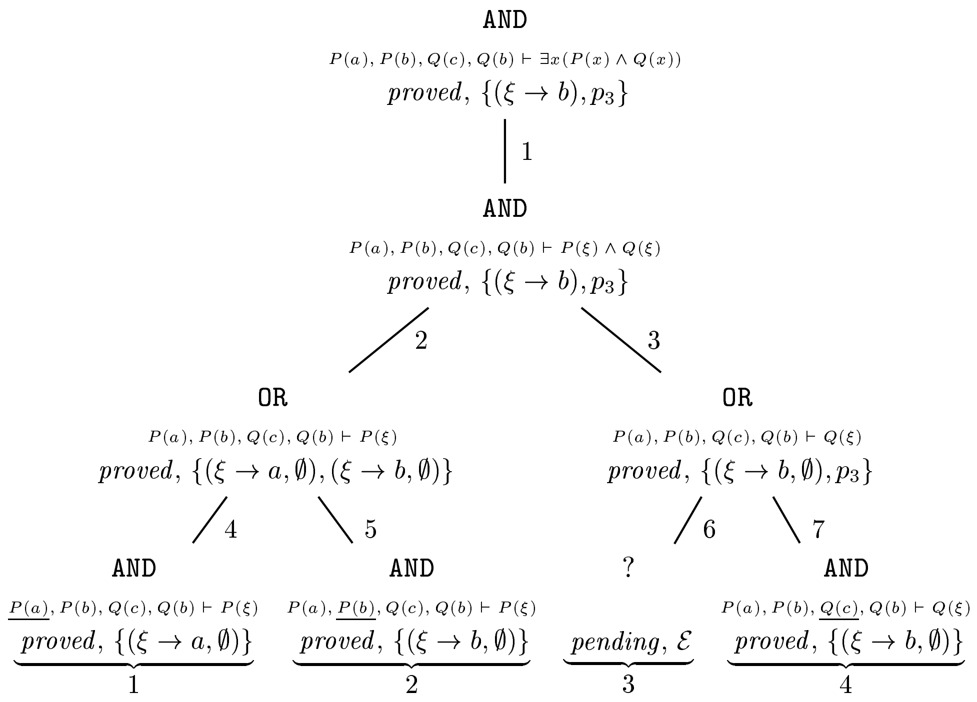

Figure 5: Finished deduction tree for $P(a), P(b), Q(c), Q(b) \vdash \exists x(P(x) \wedge Q(x))$.

finished, that is, it presents a valid proof of the initial proof situation. The system associated with the root of the tree contains a correct substitution $\xi \to b$. The pointer p_3 shows the place where the deduction tree can be further expanded.

7 Conclusion

The developed implementation of the metavariable method is efficient and does not depend on the search strategy. Note that using the same technique one can implement any method where the provability of a proof situation depends on extra data that has to be computed going down-up along a deduction tree.

We plan to further investigate different search strategies and implement heuristics which allow to speed-up the proof search.

References

[1] W. Bibel. *Automated Theorem Proving.* Friedr. Viewing & Sohn, second edition, 1986.

[2] K. Broda. The relationship between semantic tableau and resolution theorem proving. In *Proceedings of Workshop on Logic*, Debrecen, Hungary, 1980. Also published as technical report, Imperial College, Department of Computing, London, UK.

[3] F. M. Brown. Toward the automation of set theory and its logic. *Artificial Intelligence*, 10:281–316, 1978.

[4] B. Buchberger. *Mathematica*: Doing mathematics by computer? In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 1–29. Springer Wien—New York, 1993. Invited talk at DISCO 93, Gmunden, Austria.

[5] B. Buchberger. Symbolic computation: Computer algebra and logic. In *Frontiers in Combining Systems*, number 40 in Applied Logics Series, pages 193–220. Kluver Academic Publishers, 1996.

[6] B. Buchberger. *Mathematica* as a rewrite language. In T. Ida, editor, *The second Fuji International Workshop on Functional and Logic Programming.* Telos publishing, November 1996. Also published as RISC report 96-40.

[7] B. Buchberger and T. Jebelean. The predicate logic prover. In *Proceedings of the First International Theorema Workshop*, number 97-20 in RISC Linz report series. Research Institute for Symbolic Computation, June 1997.

[8] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the *Theorema* project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*, Maui, Hawaii, July 1997.

[9] B. Buchberger, T. Jebelean, and D. Vasaru. *Theorema*: A system for formal and scientific training in natural language presentation. Technical report 97-34, Research Institute for Symbolic Computation, 1997.

[10] B. Buchberger, T.Ida, and D.Vasaru, editors. *Proceedings of the First International Theorema Workshop*, Hagenberg, Austria, June 1997. Also published as RISC Technical Report No 97-20.

[11] B. Buchberger and E. Tomuta. Combining provers in the *t*heorema system. Technical report 98-02, Research Institute for Symbolic Computation, 1998.

[12] G. Cioni, A. Colagrossi, and A. Miola. A sequent calculus for automated reasoning in symbolic computation systems. *J. Symbolic Computation*, 19:175–199, 1995.

[13] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, second edition, 1996.

[14] J. H. Gallier. *Logic for Computer Science.* Harpet & Row, 1986.

[15] G. Gentzen. Untersuchungen über das logische schließ. *Mathematische Zeitschrift*, 39:176–210, 405–433, 1934.

[16] J. Girard, P. Taylor, and Y. Lafont. *Proofs and types.* Cambridge University Press, 1989.

[17] C. Kanger. *A simplified proof method for elementary logic*, pages 87–93. Computer Programming and Formal Systems / Studies in Logic and Foundations of Mathematics. North-Holland, 1963.

[18] J. Krajíček and P. Pudlák. The number of proof lines and size of proofs in first order logic. *Arch. Math. Logic*, 27:69–84, 1988.

[19] A. Martelli and U. Montanary. An efficient unification algorithm. *ACM transactions on Programming Languages and Systems*, 4(2):258–282, 1988.

[20] V. P. Orevkov. *Complexity of Proofs and Their Transformations in Axiomatic Theories.* Translations of Mathematical Monographs. American Mathematical Society, 1991.

[21] D. Prawitz. An improved proof procedure, 1960. Theoria 26.

[22] E. Tomuta. *An Architecture for Combining Provers and Applications in the Theorema System.* PhD dissertation, Research Institute for Symbolic Computation, July 1998.

[23] H. Wang. Toward mechanical mathematics. *IBM J. of Research and Develpment*, 4(1), 1960.