A Look at TPS

Dale A. Miller
Department of Mathematics
Carnegie-Mellon University
Pittsburgh, PA 15213

Eve Longini Cohen
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Peter B. Andrews
Department of Mathematics
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Certain aspects of the theorem proving system *TPS* are described. Type theory with λ-abstraction has been chosen as the logical language of *TPS* so that statements from many fields of mathematics and other disciplines can be expressed in terms accessible to the system.

Considerable effort has been devoted to making *TPS* a useful research tool with which interaction is efficient and convenient. Numerous special characters are available on the terminals and printer used by *TPS*, and many of the traditional notations of mathematics and logic can be used in interactions with the system. When constructing a proof interactively, the user needs to specify only essential information, and can often construct needed wffs from others already present with the aid of a flexible editor for wffs.

TPS constructs proofs in natural deduction style and as p-acceptable matings. Proofs in the latter style can be automatically converted to the former. *TPS* can be used in a mixture of interactive and automatic modes, so that human input need be given only at critical points in the proof.

The implementations of the algorithms which search for matings and perform higher order unification are described, along with some associated search heuristics. One heuristic used in the search for matings, of special value for dealing with wffs containing equality, considers in close sequence vertical paths on which mating processes are likely to interact. A very useful heuristic for pruning unification trees involves deleting nodes subsumed by other nodes. It is shown how the unification procedure deals with unification problems which keep growing as the search for a mating progresses.

It has been found that although unification of literals is more complicated for higher order logic than for first order logic, this is not a major source of difficulty.

An example is given to show how *TPS* constructs a proof.

This work is supported by NSF Grants MCS78-01462 and MCS81-02870.

§1 Introduction

In this paper we shall describe certain features of an automated theorem-proving system called *TPS* which is under development at Carnegie-Mellon University, and which implements the ideas in [1] and [2], with which we assume familiarity.

§2 Long term goals and design philosophy

There is abundant evidence of the need for new intellectual tools to aid mankind in dealing with the complexities of the modern world. Among these tools must surely be some designed to facilitate logical reasoning. Since the construction of proofs is one of the best understood aspects of logic, work on automated theorem proving provides a natural starting point for research on the more general problem of automating logical reasoning.

Naturally, one must expect that the early stages of work on automating reasoning will produce systems which can be successful only on fairly simple problems. Nevertheless, in designing such a system it is helpful to keep in mind the features and capabilities which one would like it to have in the long run, and to provide an adequate framework for the long term developments which are anticipated. One of the important decisions which must be made at an early stage concerns the language of logic to be used.

As one envisions the sophisticated question-answering systems of the future, it seems clear that scientific knowledge at various levels of abstraction will gradually be incorporated into them. Since mathematics is the language of science, artificial intelligence systems which use logic in a sophisticated way will surely come to be regarded as severely limited if they cannot handle mathematics on a conceptual as well as a computational level. Even if mathematics is not used explicitly, the sort of logical abilities needed for mathematics are vital for many sophisticated applications of logic. For this reason, one of the objectives in the development of *TPS* is to obtain a system in which mathematics can be expressed naturally and in which mathematical reasoning can be (partially) automated.

Of course, mathematics provides an ideal environment for research on logical reasoning. Mathematical statements are customarily expressed in ways which make their logical structure very clear, and mathematical arguments are customarily carried out with an attention to logical detail unparalleled in any other field. The theorems of mathematics exhibit an enormous variety of subject matter and complexity. Thus mathematics provides a wealth of ready-made examples which can be used in the study of ways to

increase the efficiency of a computer's deductive apparatus.

In order to provide for a wide variety of possible future applications, *TPS* needs a universal language of logic in which virtually any mathematical statement can be expressed. For reasons which we shall discuss only briefly here, a formulation of type theory with λ-notation due to Church [3] has been chosen as the logical language used by *TPS*. Mathematical statements can be translated rather directly into this language, and many sets and functions have names in the language, so no axioms concerning set existence (except an Axiom of Infinity and the Axiom of Choice, where appropriate) are needed. Of course, to use type theory one must assign a type to each mathematical entity, but mathematicians naturally do make intuitive distinctions between different types of mathematical entities. Indeed, type symbols provide important syntactic clues which enable an automated theorem-prover automatically to eliminate from consideration expressions which would be permitted in languages such as axiomatic set theory, but not in type theory, and which a working mathematician would reject almost immediately as meaningless. This is particularly valuable for the unification algorithm, where many inappropriate substitutions are avoided via type considerations.

Actually, it has been found that unification algorithms for type theory ([4], [6]), which combine λ-conversion with substitution, are powerful tools which enable one to find proofs of certain intricate theorems (such as Cantor's Theorem for Sets) which are hard to prove automatically by other means. The availability of these algorithms constitutes a strong argument for the use of type theory.

The basic logical approach to theorem proving underlying the design of *TPS* is discussed in [1] and [2], where the formal discussion is limited to the problem of proving theorems of first order logic. While *TPS* is in principle logically complete in automatic mode only for proving theorems of first order logic at present, it has a number of facilities for handling wffs of higher order logic, such as the interactive construction of proofs in natural deduction style, and unification. While it may seem premature to be concerned with automating type theory when there is still so much to be done to develop systems which can handle first order logic satisfactorily, the use of type theory has proved to be very advantageous even at the current stage of research, since there are numerous theorems, such as Cantor's Theorem, which can be expressed in type theory much more naturally than in first order logic even though their proofs involve essentially only the techniques of first order logic combined with higher order unification.

TPS is both a system under development and a research tool. The program is written in

LISP and uses about 18,000 lines of code. In order to maximize its usefulness, we have tried to make it serve those who use it, rather than asking them to adapt to its requirements. Thus, we have devoted considerable effort to making the computer accept input convenient for people to provide, and provide output convenient for people to use. Much of the traditional notation of mathematics and logic can be used in communicating with *TPS*. In order to achieve this, *TPS* had to be given the ability to handle the many special symbols which are used in mathematics and logic. This required a sizable amount of hardware and software support.

§3 Support for special characters

TPS is written in CMU LISP, a descendant of UCI LISP. The system uses Concept 100 terminals and a Xerox Dover printer, both of which support multiple character sets. Two auxiliary sets of 128 characters were developed by us for the terminals. These sets contain Greek, script and boldface letters, plus many mathematical and logical symbols. Some characters are available as subscripts and superscripts. The Dover is capable of printing all these characters and many more. In order to use these characters, several improvements to the standard LISP input and output facilities have been made. For example, an input facility called RdC was added to LISP so that special characters could be entered and translated to a representation suitable for LISP.

The Concept terminals can also define windows, read and write the cursor position, and insert and delete both characters and lines. These capabilities permitted us to write two text editors for use in *TPS*. The *scratch pad* editor is a very simple editor relying on the terminal's local screen editing. A second, more powerful editor called VIDI is a screen-type editor, much like EMACS, which supports special characters, and permits such text to be stored and retrieved from disk storage.

TPS can also produce hard copy images of text containing special characters by preparing a manuscript file for the document compiler SCRIBE [7]. This typesetting language permits access to numerous character sets and many formating features. The result of compiling such files is a second file which can be printed on the Dover. Files can be made which permit the user's interaction with *TPS* to be redisplayed on the terminal screen.

§4 The TPS interface with the user

TPS has two representations for wffs. The internal representation is a linked list whose structure mirrors the structure of the wffs found in [3]. This representation is very convenient for computing, but it does not resemble conventional notation of mathematical logic because of its syntax and lack of special characters. The *TPS* user deals with an external representation of wffs which is much more convenient to read and type.

The external representation uses both brackets and Church's dot convention to specify scopes of connectives. Function symbols and predicates can be declared to be infix instead of simply prefix, which permits a more natural rendering of such symbols as $\subseteq$, $+$, and $\equiv$. Symbols are handled by first assigning each character a name and then giving each symbol a list of character names, called its *face*. For example, the quantifier $\exists_1$ has the internal name EXISTS1, and this has the face (EXISTS SUB1), for the two characters used to represent this symbol. *TPS* can both print this symbol with the correct characters and find its occurrences within text typed by the user. Text containing special characters can be input by both RdC and the VIDI editor. The internal name of a symbol is used if a device is being used on which special characters are not available.

The fact that wffs are represented within higher order logic proves quite useful, even when we restrict our attention to first order wffs. Abbreviations can be defined very conveniently by using λ-abstraction. For example, SUBSET is defined by the wff $[\lambda P_{o\alpha}\ \lambda Q_{o\alpha}\ \forall x_{\alpha}\ .\ Px \supset Qx]$. Thus, instantiating SUBSET is done simply by substituting this term for the occurrences of $\subseteq$ in the wff, and then doing λ-contractions of the expression. Such definitions can be stored in libraries as polymorphic type expressions. Their actual type is determined from the context. Individual users of *TPS* can have their own libraries.

A structure editor is available which permits the user to edit the internal representation of a wff. In this editor the user can issue commands to move to any subexpression of the original wff and change it by placing it into various normal forms, changing bound variables, extracting subformulas, instantiating abbreviations, etc. Although the user is editing the internal representation, it is the external representation which is printed out to the user at each level of the editing. This editor also has a convenient way of using the two text editors on subformulas.

The user can specify a wff in many different ways. The user can refer to other wffs or their subformulas, or the result of processing other wffs. For example, all of the following ASSERT commands will place into the current proof outline a line numbered 4 containing the wff specified by the second argument:

```
>assert 4 "FORALL X(OA) . [POWERSET . X INTERSECT Y(OA)] SUBSET
                . POWERSET Y"

>assert 4 rdc
RdC>∀X_oα . [P . X ∩ Y_oα] ⊆. P Y

>assert 4  pad

>assert 4 thm1.

>assert 4 (neg (skolem thm1))

>assert 4 (loc 2)
```

In the first command, the wff is entered as a string without special characters. In the second command, the user requests the RdC prompt and then enters the wff. The third command places the cursor in the scratch pad editor and the user then either edits a wff already present in the pad or types another one. When the contents of the pad is the desired wff, pressing the appropriate key on the Concept will send the pad's contents to the wff parser. In the fourth and fifth commands a previously defined wff called THM1 is used. The last command calls the LOC function, which is much like the structure editor but is simply used to locate a subformula of a given wff; in this case, it returns a subexpression of the wff in line 2 of the proof.

TPS can print wffs in several different fashions. For example, they can be pretty-printed so that the arguments of infix operators are indented appropriately. Also, *TPS* can produce two-dimensional diagrams of wffs in nnf such as are found in [2]. When these are too big to fit on one sheet of paper, they are laid out so that several sheets can be pasted together to make one large diagram.

The user of *TPS* can make many choices about such matters as the way wffs are displayed, the amount of information that is printed out during the search for a proof, the search heuristics and parameters that are used, and the degree to which *TPS* functions in automatic or interactive mode. These choices are made by changing the values of certain variables called flags. A command called REVIEW provides help for the user who wishes to recall what flags are available, what they mean, and what their current values are.

TPS also has help facilities to remind the user of the correct format for various commands, the commands available in the editor, etc.

The user can cause a *status line* to be displayed on the terminal screen. This shows which lines of the proof are sponsored by each planned line, and is automatically updated as the proof is constructed.

§5 Organization of the proof process

Proofs in *TPS* can be constructed in natural deduction format or as *p-acceptable matings* (see [2]), and both styles of proof can be constructed automatically or interactively. When constructing natural deduction proofs, *TPS* builds and progressively improves *proof outlines* (see [1]), which are fragments of proofs in which certain lines called *planned lines* are not yet justified. The transformation rules described in [1] are implemented in *TPS* as commands which can be used interactively to fill out the proof. Since the context often suggests how these commands should be applied, by setting appropriate flags and letting *TPS* compute default arguments for commands, the user can let this process proceed automatically until decisions are required. When input is needed, wffs can be specified with minimal effort as described above, and *TPS* checks for errors and allows corrections.

Instead of proceeding interactively, one can send the wff to be proved in a planned line to the mating program. When a p-acceptable mating is found, it is converted into a *plan* and used to construct a natural deduction proof of the planned line as described in [1]. This process is illustrated with an example in §8. Even when a plan is being used, the user can intervene to make decisions concerning the order in which rules are to be applied and thus control the structure of the proof.

Thus, the user can construct some parts of the proof interactively or semi-automatically and others automatically. Plans as well as proofs can be constructed interactively.

The mating program can also be run as an independent entity.

All of the rules of inference of natural deduction mentioned in §2 of [1] are available as *deducing rules* (see [1]). In addition, there is an ASSERT command which can be used to insert into a proof a theorem which the interactive user obtains from a library of theorems or simply asserts. Deducing rules are used to construct proofs down from the top, while *planning rules* essentially provide for working backwards to build a proof up from the bottom. By combining these rules *TPS* or the interactive user can work both forward and backward. This facilitates the construction of the proof and the implementation of heuristics to control its style and structure. Of course, some of the deducing rules (such as Universal Generalization) are used only in interactive mode, since in automatic mode *TPS* inserts the same lines into the proof by the use of appropriate planning rules.

At least one transformation rule is available to deal with every possible form of an active or planned line which is not a literal. Thus, a number of special forms of Rule P (see [1]) are available, and are invoked when appropriate. Examples of these are to infer *A* and *B*

from $[A \wedge B]$, to infer A from $\sim\sim A$, and to push in negations. At present, deduced lines of the form $[A \supset B]$ are replaced by $[\sim A \vee B]$, and disjuctions are broken into cases using the rule P-Cases of [1], though this sometimes yields inelegant proofs. When all quantifiers and definitions have been eliminated from a planned line and the active lines which it sponsors, *TPS* invokes the unrestricted Rule P to infer the planned line, thus providing the essential link between the top and bottom parts of the proof of the planned line. It is anticipated that more elegant proofs will be constructed in automatic mode as we learn to make more sophisticated use of the information in plans, especially the matings.

§6 Mating search

The mating program first processes the wff to be proved by removing all abbreviations, negating, miniscoping (when appropriate), skolemizing, and transforming to negation normal form. The final wff contains only universal quantifiers, conjunctions, disjunctions, and negations with atomic scopes. *TPS* then attempts to derive a contradiction from this wff by searching for a p-acceptable mating for each top-level disjunct in turn. As in [2], we call any set of pairs of literals a *potential mating*. A vertical path is *fixed* if it contains a pair of mated literals. A potential mating is *complete* if each vertical path is fixed, otherwise it is *partial*. A *p-acceptable mating* is a complete potential mating for which a substitution exists which makes mated literals complementary. Thus, a contradiction is derived, and by Herbrand's theorem the original wff is valid.

The first step in the mating search is to create a connection graph, which stores information useful in constructing the mating. As in a strictly first order connection graph, an arc is directed from literal A to literal B whenever $\sim A$ and B are unifiable, *i.e.*, whenever A and B are *c-unifiable*. Type theory adds additional complications to the construction of the graph:

- It is not *a priori* obvious which pairs of literals can be c-unified, or which is to be the "negative" literal. Because the substitution term may contain $\sim$, it may be possible to c-unify A and B when either, neither, or both of the literals is a negation, or to c-unify the literals in both orientations. Of course, a path can be fixed by mating the literals in either orientation, but different orientations produce different substitutions.

- Unification in type theory is in general undecidable, so the algorithm may not terminate. We deal with this problem by limiting the depth of unification search at the time the graph is created.

- Even when a unifier exists, there may be no most general unifier of two wffs in

type theory. In this case there will be branching in the unification tree. However, during the construction of the connection graph we pursue only those parts of the unification problem requiring no branching.

Each arc in the connection graph is labeled with a partial c-unifier of its two literals.

The next part of the process is to choose a vertical path in the wff containing no mated pair, choose from the connection graph a pair of literals which fixes that path, add it to the mating under construction, partially unify (still allowing no branching in the unification tree) the set of disagreement pairs representing the partial mating, and then iterate the above process until a complete potential mating is constructed, backtracking as required by failures of mating or unification. The same partial mating may arise several times in the search process, but *TPS* considers it only once. When a completed mating is obtained, the full power of unification is allowed, occurring in parallel with the construction of a new mating. After all potential matings of a wff have been considered, *TPS* replicates certain quantifiers and their scopes, and seeks a mating for the enlarged wff. Thus it alternates between the construction of a single mating, and unification work on that partial mating and on any number of previously constructed completed matings, which may be from any number of replication levels. The user can control the relative effort devoted to the mating and unification procedures.

The search for a mating may cause certain quantifiers to be duplicated inappropriately, so after a p-acceptable mating has been found, *TPS* simplifies the replication scheme if possible, and makes appropriate adjustments of the substitutions and mating. Such simplification avoids certain ambiguities which might otherwise arise when Skolem functions are eliminated from substitution terms in the process of constructing a plan from the mating. It also eliminates redundancies which would otherwise occur in proofs constructed from the mating.

We have tried a number of different heuristics for the choice of an unfixed path and a pair to fix it. *TPS* tries to fix first the paths where the choices are most constrained. The default procedure for choosing paths is to focus on those with the smallest number of matable pairs with respect to the current partial mating.

Heuristics are also available to deal with certain special situations, such as those in which equality occurs. Equality is defined as $[\lambda x_\alpha \lambda y_\alpha \forall Q_{o\alpha} . Qx \supset Qy]$, and a positive equality statement occurs as $\forall Q[\sim QX \vee QY]$. When one mates one of these literals, one constrains the possible substitutions for Q and limits the possible mates for the other literal. This motivates the following heuristic.

Suppose that the wff for which one is seeking a mating contains a subformula $[K \vee L]$, where K and L are conjunctions of literals, that $\boldsymbol{P}$ is a path which passes through K, that A and B are literals on $\boldsymbol{P}$, with A in K but B not in K, and that the mating process has tentatively decided to mate A with B. Of course, there is another path $\boldsymbol{P}'$ which is like $\boldsymbol{P}$ but which passes through L instead of K. $\boldsymbol{P}'$ must also be given a pair of mated literals, and we might as well assume that at least one of these is to be in L, since otherwise the mated pair which fixes $\boldsymbol{P}'$ also fixes $\boldsymbol{P}$, and the mating of A with B is unnecessary. If each literal in L has a variable in common with A or with B, then the substitution required to mate A with B may constrain the possible ways of fixing $\boldsymbol{P}'$. Indeed, if this constraint is so strong that there is no way to fix $\boldsymbol{P}'$, then one should not mate A with B. This leads to the following heuristic: having fixed $\boldsymbol{P}$ by mating A with B, consider next all variant paths $\boldsymbol{P}'$ which satisfy the given description, and see if they can each be fixed. This may require repeated calls on this heuristic. In this way, one may be able to reject bad partial matings sooner rather than later, which is the essence of good heuristic search.

§7 Unification search

The unification scheme used by *TPS* is that described by Gérard Huet in [4], with only minor modifications. Each mating is represented by a list of pairs of wffs which require unification. We call each such pair a *disagreement pair* (*dpair*), and the whole list a *unification node*. Each wff of type theory is of the form $[\lambda w^1,...,\lambda w^n . hE^1,...,E^m]$, where n, m $\geq$ 0. We call h the *head* of the wff, $E^1,...,E^m$ its *arguments*, and $\{w^1,...,w^n\}$ its *binding*. We call a wff *rigid* if its head is either a constant or a member of its binding. Non-rigid wffs are said to be *flexible*. The head of a rigid wff cannot be altered by substitution, so if two rigid wffs are to be unified, the heads must be identical (modulo alphabetic changes of bound variables). Huet's unification procedure employs two alternating subroutines, `Simpl` and `Match`. `Simpl` reduces a node by breaking up all compatible rigid-rigid dpairs into the subproblems represented by their arguments, or returns failure upon finding non-compatible rigid-rigid dpairs. `Match` suggests a set of substitutions for the flexible head of a flexible-rigid dpair. If that set contains more than one substitution, it necessitates branching in the unification tree to create one node for each possibility. The non-unit sets of substitutions are called *branching* sets. Since we are only interested in the existence of a unifying substitution, a node consisting only of flexible-flexible dpairs is considered to be a terminal success node.

`SIMPLIFY` is a procedure based on Huet's `Simpl` with the following additions: 1) we

include only one dpair from each equivalence class with respect to alphabetic changes of bound variables; 2) we recognize the trivial unifiable fixed point problem $\langle x, E\rangle$ where x does not occur free in E, with the substitution of the most general unifier $x \rightarrow E$; 3) we recognize some non-unifiable fixed point problems, including those suggested by Huet in 3.7.3 of [5]. MATCH is precisely Huet's algorithm of the same name, offering a choice of either the η-rule or non-η-rule substitutions.

7.1. Search heuristics for full unification

Once we have a complete description of the unification problem, *i.e.*, the potential mating is complete, we search for a unifier with the full force of Huet's algorithm. For each variable which is the flexible head of a flexible-rigid dpair, one substitution set is computed by applying MATCH to the first flexible-rigid dpair in which that variable occurs as a flexible head. Of course, such substitutions can be stored between iterations of the algorithm to avoid recomputing. All substitution sets which reduce to singletons after SIMPLIFication are applied immediately. If the search must branch because all substitution sets have multiple members, we choose a set causing the least branching. Unapplied branching substitution sets are not discarded, but passed on to each descendant. Each node also has associated with it a list of compound substitutions for the variables in the original mated pairs.

Surprisingly often, one encounters a unification node **M** which subsumes another node **N**, *i.e.*, **M** is a subset of **N** (modulo alphabetic changes of bound and free variables). In this case any unifying substitution for **N**, with the appropriate alphabetic transformation, will also unify **M**; thus, since we are interested only in the existence of a unifier, it is sufficient to consider **M**. We detect this relationship between nodes quite efficiently using a hash table on dpairs. When **N** is a newly-created node there is no question as to the right course of action: we simply eliminate **N** from further consideration. Of course, if **N** is the sole descendant of **M** this terminates attempts to find a unifier for **M**. However, when **N**, a proper superset of **M**, already exists in the unification tree and possesses descendants, it may well be that the leaves of the tree under **N** represent more progress toward the ultimate unification than does node **M**. It is not clear that the best action is to delete **N** and all its descendants (except, possibly, **M**), though that is our current strategy.

7.2. Search heuristics for unification on partial matings

We also perform unification on nodes representing partial matings. In order not to have multiple nodes to which a dpair representing a newly mated pair must be added, we allow no branching during unification until the mating is complete. Since the mating is incomplete, so also is the unification problem, so a node containing only flexible-flexible dpairs is not a terminal success node. Also, since the partial mating might well be a subset of an already completed mating though it in no sense subsumes the complete mating, the subsumption procedure described in the previous paragraph is applicable only when the subsuming node represents a complete mating.

In building up a mating, one wants to be sure that the partial unifier for mating a new pair of literals is consistent with the substitutions associated with the partial mating to which the new pair is to be added, and one certainly wants to make use of the work towards unification that one performed in producing the connection graph. When a pair of literals is added to the mating, therefore, we want to merge the substitutions labeling the arc between them in the connection graph with the substitutions already associated with the mating. Fortunately, the substitutions produced by Huet's algorithm are very well behaved. Given two substitutions $(v \rightarrow T^1)$ and $(v \rightarrow T^2)$ for the same variable v, the substitutions are compatible iff the dpair $\langle T^1, T^2 \rangle$ does not fail under SIMPLIFication, since T^1 and T^2 contain only new and distinct variables.

7.3. Experience with higher-order unification

Because the higher order unification algorithm constructs a search tree with potentially infinite branches, it might be expected to place an unreasonable computational burden on a theorem prover. However, our experience suggests that this is not the case when the algorithm is implemented as described above. The difficulties we have encountered in trying to prove various theorems have generally been attributable more to the complexity of the search for a mating than to the complexity of the unification process.

§8 An example

We shall now demonstrate several features of *TPS* by having it prove THM87:

```
∃w ∀j ∃q .[j ∈ .[P a .h j ] ∪ .P w .k j ] ⊃ .j ∈ .P w q
```

Here ∈ is an abbreviation for [λX λP .P X] and ∪ is an abbreviation for [λP λQ λX .[P X] ∨ .Q X]. This theorem is neither interesting nor challenging, but permits us to briefly illustrate various features of *TPS* in the space available. Here h and k are function symbols and [h j] denotes the value of h on the argument j. For this example, we have suppressed the printing of type symbols. A dot denotes a left bracket whose mate is as far to the right as is consistent with the pairing of brackets already present. Except for the comments in italics, the remaining text of this section is essentially what the user sees when *TPS* processes this theorem.

The PLAN *command initiates the proof process by creating a proof outline whose only line is the theorem to be proved.*

```
*PLAN THM87
(100)          ⊢  ∃w ∀j ∃q .   [j ∈ .[P a .h j ] ∪ .P w .k j ]
                                ⊃ .j ∈ .P w q                    PLAN1
```

At this stage the justification PLAN1 *for line 100 is just an empty label. TPS chooses 100 to be the number of the last line in the proof. This choice is easily changed, and if a given choice does not leave enough room for the complete proof, all the lines can easily be renumbered.*

The mating search program now attacks the theorem in line 100. TPS takes the wff in line 100, instantiates all definitions, places it in negation normal form, skolemizes it, and attaches names to the literals. The result is represented as a vertical path diagram:

```
∀w q
 |          LIT3                    LIT4                  |
 | P a [h .jA w ][jA w] V P w [k .jA w ].jA w              |
 |                                                        |
 |                    LIT6                                |
 |               .P w q .jA w                            |
```

Here disjunctions are displayed horizonally and conjunctions are displayed vertically so that this diagram represents the wff:

```
∀w∀q .[[P a [h .jA w].jA w] V .P w [k .jA w].jA w] ∧ ~ .P w q .jA w
```

TPS now searches for an acceptable mating for this wff.

```
Path with no mates: (LIT3 LIT6)
   try this arc: (LIT3 LIT6)

Partial Mating 0:
```

```
   (LIT3)-(LIT6)

Path with no mates: (LIT4 LIT6)
Path with no mates: (LIT3 LIT6)
No proof on this level
```

TPS has found that there is no acceptable mating for this particular amplification of the theorem. Hence, it must duplicate some quantifiers. It was proved in [2] that duplicating outer quantifiers is a complete although not sophisticated method to duplicate quantifiers. In the diagram below, variables and literals are given a suffix to show which copy of the orginal variable or literal they represent.

```
Duplicate outer quantifiers.

| ∀w↑1 q↑1                                                                  |
|    |          LIT3↑1                              LIT4↑1                  |
|    | P a [h .jA w↑1 ][jA w↑1] V P w↑1 [k .jA w↑1 ].jA w↑1                  |
|    |                                                                       |
|    |                          LIT6↑1                                       |
|    |                    ¬ .P w↑1 q↑1 .jA w↑1                               |
|                                                                           |
|  ∀w↑2 q↑2                                                                 |
|    |          LIT3↑2                              LIT4↑2                  |
|    | P a [h .jA w↑2 ][jA w↑2] V P w↑2 [k .jA w↑2 ].jA w↑2                  |
|    |                                                                       |
|    |                          LIT6↑2                                       |
|    |                    ¬ .P w↑2 q↑2 .jA w↑2                               |
```

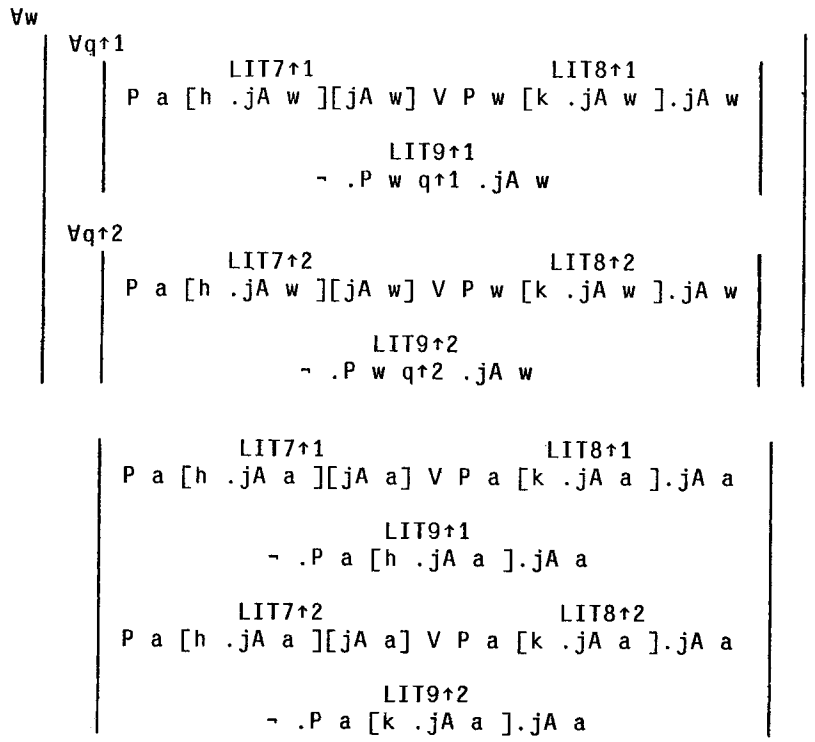
In this expanded wff, TPS quickly finds an acceptable mating.

```
Path with no mates: (LIT3↑1 LIT6↑1 LIT3↑2 LIT6↑2)
   try this arc: (LIT3↑2 LIT6↑1)

Partial Mating 0:
   (LIT3↑2)-(LIT6↑1)

Path with no mates: (LIT3↑1 LIT6↑1 LIT4↑2 LIT6↑2)
   try this arc: (LIT4↑2 LIT6↑2)

Partial Mating 0:
   (LIT3↑2)-(LIT6↑1), (LIT4↑2)-(LIT6↑2)

q²  ⇐k .jA a     q¹  ⇐h .jA a     w²  ⇐a          w¹  ⇐a
```

At this point, an acceptable mating and its associated substitution have been found.

```
The replication has been simplified.
Old: ( w   2 )   New: ( q   2 )
The substitution has also been changed.
Old:  q²  ⇐k .jA a     q¹  ⇐h .jA a     w²  ⇐a     w¹  ⇐a

New:  w  ⇐a     q¹  ⇐h .jA a     q²  ⇐k .jA a
```

TPS now displays the new replicated wff and the result of instantiating its quantifiers using the new substitution.



An acceptable mating is:
$(LIT8^2\ \ LIT9^2)$ $(LIT9^1\ \ LIT7^2)$ $(LIT8^1\ \ LIT9^2)$ $(LIT7^1\ \ LIT9^1)$

Now PLAN1 *must be constructed by removing skolem functions from the substitution terms and computing ancestory information for variables and atoms as mentioned in [1].*

PLAN1 is:

$$\exists w\ \forall j\ .\quad [\exists q^1\ .[[P\ a\ [h\ j\ ]\ j\ ]\ \vee\ .P\ w\ [k\ j\ ]\ j\ ]\ \supset\ .P\ w\ q^1\ j\ ]$$
$$\qquad ATM3^1 \qquad\qquad ATM4^1 \qquad\qquad ATM6^1$$
$$\vee\ .\exists q^2\ .[[P\ a\ [h\ j\ ]\ j\ ]\ \vee\ .P\ w\ [k\ j\ ]\ j\ ]\ \supset\ .P\ w\ q^2\ j$$
$$\qquad ATM3^2 \qquad\qquad ATM4^2 \qquad\qquad ATM6^2$$

The substitution is: $w \Leftarrow a$ $q^1 \Leftarrow [h\ j]$ $q^2 \Leftarrow [k\ j]$

The mating is:
$(ATM6^2\ \ ATM4^2)$ $(ATM3^2\ \ ATM6^1)$ $(ATM6^2\ \ ATM4^1)$ $(ATM6^1\ \ ATM3^1)$

The replication scheme is: (q 2)

TPS now proceeds to build the proof outline from this plan.

```
>P-EXISTS a 100
(99)      ⊢  ∀j ∃q .[j ∈ .[P a .h j ] ∪ .P a .k j ] ⊃ .j ∈ .P a q
                                                              PLAN2
```

(100) $\vdash \exists w\ \forall j\ \exists q\ .\ [j \in .[P\ a\ .h\ j\ ] \cup .P\ w\ .k\ j\ ] \supset .j \in .P\ w\ q$

∃G: a 99

The Plan for line 99, PLAN2, is:

$\forall j\ .\ [\exists q^1\ .[[P\ a\ [h\ j\ ]\ j\ ] \vee .P\ a\ [k\ j\ ]\ j\ ] \supset .P\ a\ q^1\ j\ ]$
ATM3[1] ATM7 ATM8
$\vee\ .\exists q^2\ .[[P\ a\ [h\ j\ ]\ j\ ] \vee .P\ a\ [k\ j\ ]\ j\ ] \supset .P\ a\ q^2\ j$
ATM3[2] ATM9 ATM10

The substitution is: $q^1 \Leftarrow [h\ j]$ $q^2 \Leftarrow [k\ j]$

The mating is:
(ATM10 ATM9) (ATM3[2] ATM8) (ATM10 ATM7) (ATM8 ATM3[1])

The replication scheme is: (q 2)

>P-ALL 99
(98) $\vdash \exists q\ .[j \in .[P\ a\ .h\ j\ ] \cup .P\ a\ .k\ j\ ] \supset .j \in .P\ a\ q$

PLAN3
(99) $\vdash \forall j\ \exists q\ .[j \in .[P\ a\ .h\ j\ ] \cup .P\ a\ .k\ j\ ] \supset .j \in .P\ a\ q$

∀G: 98

PLAN3 *has been omitted since it differs only slightly from* PLAN2. *The reader may wish to construct it. TPS now recognizes that the present plan is existentially complex (see [1]), so the proof must now proceed in an indirect fashion. The symbol* ⊥ *is used to denote falsehood.*

>P-INDIRECT 98
(1) 1 $\vdash \sim .\exists q\ .[j \in .[P\ a\ .h\ j\ ] \cup .P\ a\ .k\ j\ ] \supset .j \in .P\ a\ q$

Hyp
(97) 1 $\vdash \bot$ PLAN4

(98) $\vdash \exists q\ .[j \in .[P\ a\ .h\ j\ ] \cup .P\ a\ .k\ j\ ] \supset .j \in .P\ a\ q$

Indirect: 97

The Plan for line 97, PLAN4, is:

$[\sim .\ [\exists q^1\ .[[P\ a\ [h\ j\ ]\ j\ ] \vee .P\ a\ [k\ j\ ]\ j\ ] \supset .P\ a\ q^1\ j\ ]$
ATM3[1] ATM7 ATM8
$\vee\ .\exists q^2\ .[[P\ a\ [h\ j\ ]\ j\ ] \vee .P\ a\ [k\ j\ ]\ j\ ] \supset .P\ a\ q^2\ j\ ]$
ATM3[2] ATM9 ATM10
$\supset \bot$

The substitution is: $q^1 \Leftarrow [h\ j]$ $q^2 \Leftarrow [k\ j]$

The mating is:
(ATM10 ATM9) (ATM3[2] ATM8) (ATM10 ATM7) (ATM8 ATM3[1])

The replication scheme is: (q 2)

>D-NEG 1

```
(2)   1   ⊢  ∀q .[j ∈ .[P a .h j ] ∪ .P a .k j ] ∧  ~ .j ∈ .P a q
                                                                  Neg: 1
>D-ALL 2 [h j]
(3)   1   ⊢  [j ∈ .[P a .h j ] ∪ .P a .k j ] ∧ .~ .j ∈ .P a .h j
                                                                  ∀I: 2
>D-CONJ 3
(4)   1   ⊢  j ∈ .[P a .h j ] ∪ .P a .k j                  Conj: 3
(5)   1   ⊢  ~ .j ∈ .P a .h j                              Conj: 3

>D-DEF1 5
(6)   1   ⊢  ~ .P a [h j ] j                               Def: 5

>D-ALL 2 [k j]
(7)   1   ⊢  [j ∈ .[P a .h j ] ∪ .P a .k j ] ∧ .~ .j ∈ .P a .k j
                                                                  ∀I: 2
>D-DEF1 4
(8)   1   ⊢  [[P a .h j ] ∪ [P a .k j ]] j                 Def: 4

>D-DEF1 8
(9)   1   ⊢  [P a [h j ] j ] ∨ .P a [k j ] j               Def: 8

>D-CONJ 7
(10)  1   ⊢  j ∈ .[P a .h j ] ∪ .P a .k j                  Conj: 7
(11)  1   ⊢  ~ .j ∈ .P a .k j                              Conj: 7

>D-DEF1 11
(12)  1   ⊢  ~ .P a [k j ] j                               Def: 11

>D-DEF1 10
(13)  1   ⊢  [[P a .h j ] ∪ [P a .k j ]] j                 Def: 10

>D-DEF1 13
(14)  1   ⊢  [P a [h j ] j ] ∨ .P a [k j ] j               Def: 13

>RULEP 97 (14 9 6 12)
(97)  1   ⊢  ⊥                                  RuleP: 14 9 6 12
```

This completes the process of constructing the proof, which is displayed below:

```
(1)   1   ⊢  ~ .∃q .[j ∈ .[P a .h j ] ∪ .P a .k j ] ⊃ .j ∈ .P a q
                                                                  Hyp
(2)   1   ⊢  ∀q .[j ∈ .[P a .h j ] ∪ .P a .k j ] ∧ .~ .j ∈ .P a q
                                                                  Neg: 1
(3)   1   ⊢  [j ∈ .[P a .h j ] ∪ .P a .k j ] ∧ .~ .j ∈ .P a .h j
                                                                  ∀I: 2
(4)   1   ⊢  j ∈ .[P a .h j ] ∪ .P a .k j                  Conj: 3

(5)   1   ⊢  ~ .j ∈ .P a .h j                              Conj: 3

(6)   1   ⊢  ~ .P a [h j ] j                               Def: 5
```

(7) 1 ⊢ [j ∈ .[P a .h j] ∪ .P a .k j] ∧ .~ .j ∈ .P a .k j
∀I: 2

(8) 1 ⊢ [[P a .h j] ∪ [P a .k j]] j Def: 4

(9) 1 ⊢ [P a [h j] j] ∨ .P a [k j] j Def: 8

(10) 1 ⊢ j ∈ .[P a .h j] ∪ .P a .k j Conj: 7

(11) 1 ⊢ ~ .j ∈ .P a .k j Conj: 7

(12) 1 ⊢ ~ .P a [k j] j Def: 11

(13) 1 ⊢ [[P a .h j] ∪ [P a .k j]] j Def: 10

(14) 1 ⊢ [P a [h j] j] ∨ .P a [k j] j Def: 13

(97) 1 ⊢ ⊥ RuleP: 14 9 6 12

(98) ⊢ ∃q .[j ∈ .[P a .h j] ∪ .P a .k j] ⊃ .j ∈ .P a q
Indirect: 97

(99) ⊢ ∀j ∃q .[j ∈ .[P a .h j] ∪ .P a .k j] ⊃ .j ∈ .P a q
∀G: 98

(100) ⊢ ∃w ∀j ∃q . [j ∈ .[P a .h j] ∪ .P w .k j]
⊃ .j ∈ .P w q ∃G: a 99

§9 Sample theorems

We here present some examples of theorems which have been proved automatically by *TPS*. Other examples were presented in Appendix B of [2]. First we give several definitions which are used in the theorems below.

[# $F_{\alpha\beta}$] $D_{o\beta}$ is the image of the set $D_{o\beta}$ under the function $F_{\alpha\beta}$. # is defined as:

$$\lambda F_{\alpha\beta}\ \lambda D_{o\beta}\ \lambda Y_{\alpha}\ \exists X_{\beta}\ .[D\ X\ ] \wedge .Y = .F\ X$$

$\subseteq$ (set inclusion) is defined as:

$$\lambda P_{o\alpha}\ \lambda Q_{o\alpha}\ \forall X_{\alpha}\ .[P\ X\ ] \supset .Q\ X$$

= is defined as:

$$\lambda X_{\alpha}\ \lambda Y_{\alpha}\ \forall Q_{o\alpha}\ .\ [Q\ X\ ] \supset .\ Q\ Y$$

THM30A

$$[U_{o\beta} \subseteq V_{o\beta}\ ] \supset .[\#\ F_{\alpha\beta}\ U\ ] \subseteq .\#\ F\ V$$

THM47A

$$\forall X_\iota\ \forall Y_\iota\ .[X = Y\ ] \supset .\forall R_{o\iota\iota}\ .[\forall Z_\iota\ .R\ Z\ Z\ ] \supset .R\ X\ Y$$

THM62

$$[[\forall U_\iota\ .P_{o\iota\iota}\ A_\iota\ U\ ] \vee .\forall V_\iota\ .P\ V\ B_\iota\ ] \supset .\exists X_\iota\ .P\ X\ X$$

THM76

$$[\forall P_{o\iota}\ .[P\ Y_\iota\ ] \supset .P\ X_\iota\ ] \supset .\forall R_{o\iota}\ .[R\ X\ ] \supset .R\ Y$$

THM82

$$\begin{aligned}
&[\quad [\forall x_\iota\ \exists y_\iota\ .F_{o\iota\iota}\ x\ y\ ] \\
&\quad \wedge\ [\exists x\ \forall e_\iota\ \exists n_\iota\ \forall w_\iota\ .[S_{o\iota\iota}\ n\ w\ ] \supset .D_{o\iota\iota\iota}\ w\ x\ e\ ] \\
&\quad \wedge\ .\forall \varepsilon_\iota\ \exists \delta_\iota\ \forall x1_\iota\ \forall x2_\iota\ .\quad [D\ x1\ x2\ \delta\ ] \\
&\qquad\qquad \supset .\forall y1_\iota\ \forall y2_\iota\ .\quad [[F\ x1\ y1\ ] \wedge\ .F\ x2\ y2] \\
&\qquad\qquad\qquad \supset .D\ y1\ y2\ \varepsilon\ ] \\
&\supset .\exists y\ \forall e\ \exists m_\iota\ \forall w\ .[S\ m\ w\ ] \supset .\forall z_\iota\ .[F\ w\ z\ ] \supset .D\ z\ y\ e
\end{aligned}$$

THM83

$$\begin{aligned}
&[\forall X_\iota\ \exists Y_\iota\ .[P_{o\iota}\ X\ ] \supset .\forall Z_\iota\ .[R_{o\iota\iota}\ X\ Y\ ] \wedge\ .P\ Z\ ] \\
&\supset .\exists U_\iota\ \forall V_\iota\ .[P\ V\ ] \supset .R\ W_\iota\ U
\end{aligned}$$

§10 Acknowledgments

We would like to acknowledge the valuable assistance provided by Frank Pfenning, Edward Pervin, and Harry Porta.

References

1. Peter B. Andrews, "Transforming Matings into Natural Deduction Proofs," in *5th Conference on Automated Deduction, Les Arcs, France*, edited by W. Bibel and R. Kowalski, Lecture Notes in Computer Science, No. 87, Springer-Verlag, 1980, 281-292.

2. Peter B. Andrews, *Theorem Proving via General Matings*, Journal of the Association for Computing Machinery **28** (1981), 193-214.

3. Alonzo Church, *A Formulation of the Simple Theory of Types*, Journal of Symbolic Logic **5** (1940), 56-68.

4. Gérard P. Huet, *A Unification Algorithm for Typed λ-Calculus*, Theoretical Computer Science **1** (1975), 27-57.

5. Gérard Huet, *Resolution d'Équations dans les Languages d'Ordre 1,2,..., ω*, Thèse de Doctorat D'État, Université Paris VII, 1976.

6. D. C. Jensen and T. Pietrzykowski, *Mechanizing ω-Order Type Theory Through Unification*, Theoretical Computer Science **3** (1976), 123-171.

7. Brian K. Reid and Janet H. Walker, *SCRIBE Introductory User's Manual*, Third edition, UNILOGIC, Ltd., 160 N. Craig St., Pittsburgh, PA 15213, 1980.