AUTOMATED NATURAL DEDUCTION IN THINKER

Francis Jeffry Pelletier
University of Alberta
Edmonton, Alberta, Canada

INTRODUCTION

Although resolution-based inference is perhaps the industry standard in automated theorem proving, there have always been systems that employed a different format. For example, the Logic Theorist of 1957 produced proofs by using an axiomatic system, and the proofs it generated would be considered legitimate axiomatic proofs; Wang's systems of the late 1950's employed a Gentzen-sequent proof strategy; Beth's systems written about the same time employed his semantic tableaux method; and Prawitz's systems of again about the same time are often said to employ a natural deduction format. [See Newell, *et al* (1957), Beth (1958), Wang (1960), and Prawitz *et al* (1960)]. Like sequent proof systems and tableaux proof systems, natural deduction systems retain the "natural form" of the logic in question, rather than converting to some normal form. Natural deduction methods typically are distinguished from sequent systems by having only "rules on the right" and from tableaux systems by having the notion of a *subproof*–an embedded sequence of lines that represent the proof of some subproblem.[1] Additionally there is a difference in the ways that proofs in these systems are presented. Proofs in tableaux systems are typically presented as trees with (sets of) formulas at the nodes; proofs in sequent systems are typically presented as trees

[1] Within natural deduction systems, it is common to distinguish between Quine-systems and the original Gentzen formulation. The difference is reflected in the account given to rules involving the existential quantifier. In Gentzen's formulation, an existential quantifier is "eliminated" by introduction of a subproof that assumes an instance of the formula using an "arbitrary name." The last line of this subproof can be "exported" to the encompassing proof if it obeys certain restrictions on variables. In a Quine-system an existential quantifier is "eliminated" directly, by replacing the variable of quantification by a suitably unused variable, and the result remains in the same subproof level. It is a consequence of this difference that Quine-systems do not obey a condition on (partial) proofs that other formulations of natural deduction do, namely that each line of a proof is a semantic consequence of all the assumptions that are active at this point in the proof. Kalish (1974) argues that this is a difference of some importance.

with entire proofs at the nodes; and natural deduction proofs are typically presented as Fitch diagrams that have some graphical method of marking off subproofs.

A *direct method* of theorem proving is to construct proofs within one of these proof theories[2], by which I mean both that the result generated would be recognized as a proof according to the definition of 'proof' and also (in the case of automated theorem proving) that the "machine internal" strategies and methods are applications of what it is legal to do within the proof theory. In Whitehead & Russell (1910), for example, this would amount to finding substitution instances of formulas for propositional variables in the axioms, and applying Modus Ponens. Were one directly constructing proofs in Smullyan's (1968) tableaux system, the output should be a list of subformulas of the original formula, each with a "sign" (indicating whether they are "true" or "false"), and arranged in such a way that the "dependencies amongst the decompositions" generated by the machine reflect what is a genuine proof as characterized by the definition of proof in Smullyan's proof system. Furthermore, the machine-internal representation of the problem should involve this decomposition and dependency formulation, and actually make use of it in determining whether the original argument was valid or not. A "direct" propositional resolution system ought to have an internal representation of clauses each as a representation of a disjunction of literals and ought to use this representation in some way so as to generate resolvants. The output should be a listing of the clauses with an ordering of which formulas gave rise to the null clause by resolution.

Of course, one should distinguish between "logic moves" and "computation moves" in this regard. Although saying that some mechanical system is direct for Whitehead & Russell might require that the proof mechanism be restricted to finding substitution instances and to applying MP, this does not rule out any of the various ways such substitution instances are stored or accessed for future reference, nor does it rule out any of the various methods by which one might choose which substitution instance ought to be applied. It *does* rule out finding that a formula is provable by

[2] A (very) *indirect method* would be to prove something in another system and then use some metatheoretic result about (say) how the two methods are both complete for the semantics, and conclude therefore that having proved it in one proof system establishes *the existence of a proof* in the original system. Somewhat less indirect would be to have machine-internal procedures that make use of logical techniques that are not legitimized by the original proof theory.

using a tableaux system and converting that proof into a proof that employs substitution into Whitehead & Russell's axioms. The distinction between logic vs. computation moves is, naturally, rather fuzzy; but it has a solid intuitive force. And if the goal is to find proofs within a specific proof theory, as opposed simply to determining whether some formula is a theorem, then one will wish to pay attention to whether a theorem proving system is direct for the proof theory under consideration. (Further reflection on this issue can be found in Pelletier 1991).

In this sense, the natural deduction theorem proving system described below, THINKER, is a direct theorem proving system for first order logic with identity of Kalish & Montague (1964); for, its internal representation and method of constructing a proof–everything THINKER does internally–is a legitimate Kalish & Montague proof-step; and its output is straightforwardly a proof in Kalish & Montague's system. Of course, the *choice* of when to do what is *not* a matter of the proof system but is instead a matter of the computational strategy. For example, choosing to add a certain formula to the goals or to engage in a blind search, is not dictated or prohibited by the logic…it is rather *allowed* by the logic and hence is simply outside the compass of the direct-indirect distinction. It is the ability of a device (or human, for that matter) to construct proofs which follow the definition of proof in a given system that makes such a device legitimately be said to employ a "direct theorem proving method." Certainly it is possible to develop a logical system in which not only are specified as part of the logic what we have here called the rules of inference but also such things as when to do "backwards subgoaling" or "forward inferencing"–and in general the various algorithmic properties of proof search. Direct theorem proving in such a logic must take place in accordance with these further decrees of what the logic is.[3]

The THINKER system was developed in the early-to-late 1980's: see my tech reports (1982, 1987). Since that time a lot of the effort has gone into extending the basic system to handle both direct methods in modal logic and also indirect "translation" methods, as well as towards the task

[3] This approach of incorporating the various proof strategies into the definition of the logic under consideration might have the advantage of being able to more clearly prove logical properties (such as completeness) of the proof search algorithm than the more traditional account of a logic. (See Sieg *et al* 1992, 1996 for such an approach). But this is not at odds with the notion of direct theorem proving; indeed, it is an instantiation of the advantages of such an approach.

of “verbalizing” proofs (Edgar & Pelletier, 1993). The initial motivation was a reaction to resolution systems of the time, both to the “unnaturalness” of such systems and to what seemed to be their shortcomings…the descriptions of problems solvable by the systems of the time seemed to be very modest indeed. I thought that many of the apparent wild-goose chases that such systems appeared to engage in was due to the use of a logical system that obscured the logical form of problems, and that this could be overcome to a large measure simply by employing some form of natural deduction. Indeed, I saw resolution provers as taking a problem that was presented in one logical system, converting it to an entirely different problem–one that is not even logically equivalent–and then trying to solve that problem in order to use its solution as a guarantee that there is a solution to the original problem. In short, I saw it as embodying the worst of indirect methods: both computationally and psychologically implausible. In these regards I was echoing the sentiments of Bledsoe (1971, 1977), although the actual systems described both in these articles and in his (1983) seemed to be much less a natural deduction framework than merely some sort of pre-processor to a resolution system. I therefore aimed at developing a more “genuine” natural deduction system. I was also attracted to the idea of trying to simulate the way people actually proved elementary logic problems, and in this regard was informed by my having taught elementary and intermediate logic to students for whom a large part of any such course was to learn the heuristics I taught about how to start, to develop, and to complete natural deduction proofs. It seemed to me that my better students could outperform the resolution-based provers described in the literature of the time, and I was interested in the cognitive reasons that might be behind this. My hypothesis was that the natural deduction format in fact was cognitively significant: that the mere fact of the general format allowed naive humans to attain greater insight into the structure of logic problems than could be had in the “unstructured” resolution format even with much more massive memory facilities. Even such strategies as “if you don't see a way to work directly towards the to-be-proved goal, then just arbitrarily apply rules of inference to a few current lines and see what turns up” are immensely helpful for students; and as we will see shortly, they also allow for the proof of many moderately difficult logic problems. About this same time, mid-to-late 1980’s, John

Pollock started developing his Oscar system (see, e.g., Pollock 1992a for summary), which was also a natural deduction-based automated system that now also includes a very interesting form of default reasoning (see Pollock 1992b). And at this time, numerous systems for proof-checking in the natural deduction systems of various textbooks were being developed, and these in turn generally had the ability to construct proofs.

The discussion of the natural deduction system, THINKER, naturally divides into three parts: a part describing the logic and what is legal in the system, a part about overall proof construction strategies, and a part concerning implementation issues and lower level data structures. The first of these concerns what is the underlying logical system/proof method, and is what one would investigate to see whether the system adequately embodies first order logic with identity. The second of these concerns what are the relevant heuristics and strategies employed by the program in constructing proofs, and is what one would investigate to see whether the implementation can, in the abstract, adequately capture the theorems of first order logic with identity.[4] It is here that one would look for "the brains" of the implementation. The third part concerns issues about how the information that is required by the proof heuristics and strategies can be efficiently made available to the program. It is here that one would look for "the guts" of the implementation, including clever programming details. Many of the details (especially concerning the implementation issues) have only been available in Pelletier (1982), which was always difficult to obtain and is long out of print. Other details, especially concerning the treatment of identity, are only available in Pelletier (1987), which is again difficult to obtain. I will present sufficient explanation of the system and the implementation so as to allow an understanding of the methods by which the (program-produced) examples presented in Appendix II were generated.

The Underlying Logical System

[4] In the abstract, I say; of course in the concrete it can't. No implementation can do this, if for no reason other than the finitude of implementations.

THINKER generates and presents proofs in the Fitch-style natural deduction system of logic described in Kalish & Montague (1964) and (the updated) Kalish, Montague, Mar (1980)–henceforth this system is called KM. The only aspect of KM that is not represented in THINKER is the use of arbitrary function symbols.[5] The only terms in THINKER are individual constants (0-place function symbols) and variables. (Of course, function symbols are theoretically dispensable: an n-place function symbol can be replaced (in context) by a n+1-place predicate symbol together with an axiom saying that the n+1st place is unique. It is not clear, however, how difficult it would be to add to THINKER either arbitrary function symbols or this way of defining functions away.) The proofs that are generated look very much like those which would be produced by a good student -- not surprisingly, since the underlying strategy was constructed so as to mimic such a student. As is typical in natural deduction systems, formulas are left in their "natural form" and not converted to any normal form. There are therefore a wide variety of Rules of Inference (which give the circumstances under which a new formula may be generated and placed into the proof; these circumstances have to do with what types of formulas have already been generated and are in the proof). With the retention of the natural form there needs to be Rules describing what can be done with each different type of formula. Writing a natural deduction automated theorem proving system is in large measure a matter of organizing the application of all these Rules so as to efficiently generate proofs. The Rules of Inference in KM are the following.[6] Each rule (except for REFL) has some preconditions in terms of formulas which must already be in the proof and which must be *antecedent* (a technical term explained below). These preconditions are stated to the left of the '==>'. When these preconditions are met, the formula to the right of the '==>' *may* be

[5] But because THINKER retains existential quantifiers, there is a sense in which it has *some* functions, namely the skolem functions. But they receive no special symbol; merely the relative placement of existential and universal quantifiers represents them.

[6] Actually, these are the Rules of Inference embodied in THINKER. Except for the treatment of identity they include all the Kalish & Montague rules, but in addition contain some so-called "derived rules of inference"--such as Quantifier Negation. A derived rule of inference is admissible into the system in the sense that its introduction does not give rise to any new theorems. It can be shown that the rules of Quantifier Negation can be admitted into Kalish & Montague in this sense. The treatment of identity in THINKER follows more the treatment given in Kalish, Montague, and Mar (1980) rather than that given in Kalish & Montague (1964).

introduced into the proof (along with an *annotation* --a justification in terms of the Rule of Inference employed and the locations [line numbers in the proof] of the preconditions). Some Rules take more than one form, as indicated here; the name of the Rule is given in the table, and its abbreviation (which is found in the proofs presented in Appendix II) is indicated in bold.

RULE	NAME	RULE	NAME
$\Phi ==> \Phi$	**R** (**R**epetition)	$\Phi ==> (\Phi v \Psi)$	**ADD** (**Add**ition)
$\Phi ==> \neg\neg\Phi$	**DN** (**D**ouble **N**egation)	$\Phi ==> (\Psi v \Phi)$	**ADD**
$\neg\neg\Phi ==> \Phi$	**DN**	$\neg(\forall\alpha)\Phi ==> (\exists\alpha)\neg\Phi$	**QN** (**Q**uantifier **N**egation)
$(\Phi\&\Psi) ==> \Phi$	**S** (**S**implification)	$\neg(\exists\alpha)\Phi ==> (\forall\alpha)\neg\Phi$	**QN**
$(\Phi\&\Psi) ==> \Psi$	**S**	$(\forall\alpha)\neg\Phi ==> \neg(\exists\alpha)\Phi$	**QN**
$(\Phi\rightarrow\Psi),\Phi ==> \Psi$	**MP** (**M**odus **P**onens)	$(\exists\alpha)\neg\Phi ==> \neg(\forall\alpha)\Phi$	**QN**
$(\Phi\rightarrow\Psi),\neg\Psi ==> \neg\Phi$	**MT** (**M**odus **T**ollens)	$(\forall\alpha)\Phi ==> \Phi'$	**UI** (**U**niversal **I**nstantiation)
$\Phi,\Psi ==> (\Phi\&\Psi)$	**ADJ** (**Adj**unction)	$(\exists\alpha)\Phi ==> \Phi'$	**EI** (**E**xistential **I**nstantiation)
$(\Phi v\Psi),\neg\Phi ==> \Psi$	**MTP**(**M**odus **T**ollendo	$\Phi' ==> (\exists\alpha)\Phi$	**EG** (**E**xistential **G**eneralization)
$(\Phi v\Psi),\neg\Psi ==> \Phi$	**MTP** **P**onens)	$\Phi\alpha,\alpha=\beta ==> \Phi\beta$	**LL** (**L**eibniz's **L**aw)
$(\Phi\rightarrow\Psi),(\Psi\rightarrow\Phi)==>(\Phi\leftrightarrow\Psi)$	**CB** (**C**ond. to **B**icond.)	$\Phi\alpha,\neg\Phi\beta ==> \neg\alpha=\beta$	**NEGID** (**Neg**ated **Id**entity)
$(\Phi\leftrightarrow\Psi) ==> (\Phi\rightarrow\Psi)$	**BC** (**B**icond. to **C**ond.)	$==> \alpha=\alpha$	**REFL** (**Refl**exivity of Identity)
$(\Phi\leftrightarrow\Psi) ==> (\Psi\rightarrow\Phi)$	**BC**		

In the rules UI, EI, and EG, Φ' is the result of replacing all free occurrences of α in Φ (that is, the ones that are bound by the quantifier phrase in $(\forall\alpha)\Phi$ or $(\exists\alpha)\Phi$) with some term (constant or variable) in such a way that if it is a variable it does not become bound by any quantifier in Φ'. Furthermore, in the case of EI, this new term must be a variable which is entirely new to the proof as thus far constructed. In the rules LL and NEGID, the relationship between $\Phi\alpha$ and $\Phi\beta$ is that some free occurrences of α in $\Phi\alpha$ can be replaced by β and these occurrences of β will be free in $\Phi\beta$. The fact that the rule REFL has no preconditions means that a self-identity statement can be introduced anywhere in the proof.

An argument in general has premises and a conclusion. In KM, premises can be entered into a proof at any time (with the annotation PREM). KM (and THINKER) allows free variables to appear in premises and conclusion. The semantic interpretation of such free variables is as if they were universally quantified. What makes natural deduction systems distinctive is the idea of *subproofs* -- "smaller" or "simpler" subproblems which, if proved, can be "put together" to constitute a proof of the main problem. Quite obviously, a major factor in natural deduction theorem proving (whether automated or not) is the timely selection of appropriate subproofs to be attempted. In KM, one indicates that one is about to attempt a subproof of formula Φ by writing "show Φ". (This is also called "setting a (sub)goal", and the line in the proof which records this is called a show-line). Since the main conclusion to be proved, C, is itself considered a subproblem, the first line of a proof will be "show C". One is allowed to write "show Φ" for any Φ at any further stage of the proof. Intuitively, one is always "working on" the most recently set subgoal -- although one can of course set a further sub-subgoal (and then "work on" that newly-set sub-subgoal). The formula following the "show" on one of these lines which initiates a subproof is not really a part of the proof proper (until it has been proved). The technical term for this is that it is *not antecedent*, and the effect of being not antecedent is that this formula is unavailable for use in the Rules of Inference.

Setting a subgoal allows (but does not require) one to make an *assumption*. The form of the assumption depends on the form of the formula whose proof is being attempted, and these assumptions can only be made on the line immediately following a show line. (They are annotated ASSUME). There are three types of assumptions allowed:

show $(\Phi\rightarrow\Psi)$	show $\neg\Phi$	show Φ
Φ ASSUME	Φ ASSUME	$\neg\Phi$ ASSUME

That is, one is always allowed to assume the explicit negation of a formula to be proved; if the formula to be proved is itself a negation, one can assume the unnegated formula; and if the formula to be proved is a conditional, one can assume its antecedent. An assumption, of course, is an antecedent line (in the technical sense), until that subproof is completed.

The final concept required here is that of *(sub)proof completion* -- the conditions under which a (sub)goal can be considered to have been proved. KM has the following methods of subproof completion:

(1) If the last portion of the proof as thus far constructed is of the form

$$\begin{array}{l} \text{show } \Phi \\ \quad X_1 \\ \quad \ldots \\ \quad \ldots \\ \quad X_n \end{array}$$

where amongst $X_1 \ldots X_n$ there is (a) no "uncanceled show" and (b) either Φ occurs "unboxed" or else both Θ and ¬Θ occur "unboxed" for some formula Θ, then this part can be changed to

$$\begin{array}{l} * \text{ show } \Phi \\ \begin{array}{|l} X_1 \\ \ldots \\ \ldots \\ X_n \end{array} \end{array}$$

$X_1 \ldots X_n$ are now *boxed* and thus are no longer antecedent, and the "show" is *canceled* (indicated by prefixing a *-sign) so that Φ is now antecedent (the boxed lines constitute a proof of Φ, given the lines antecedent to this show line).

(2) If the last portion of the proof as thus far constructed is of the form

$$\begin{array}{l} \text{show } (\Phi \rightarrow \Psi) \\ \quad X_1 \\ \quad \ldots \\ \quad \ldots \\ \\ \quad X_n \end{array}$$

and amongst $X_1 \ldots X_n$ there is (a) no "uncanceled show" and (b) Ψ occurs "unboxed", then this part can be changed to

$$\begin{array}{l} * \text{ show } (\Phi \rightarrow \Psi) \\ \begin{array}{|l} X_1 \\ \ldots \\ \ldots \\ X_n \end{array} \end{array}$$

$X_1 \ldots X_n$ are now *boxed* and not antecedent; the show line is *canceled* and (Φ→Ψ) is antecedent.

(3) If the last part of the proof as thus far constructed is of the form

$$\begin{array}{l} \text{show } (\forall\alpha)\Phi\alpha \\ \quad X_1 \\ \quad \ldots \\ \quad \ldots \\ \quad X_n \end{array}$$

where amongst $X_1 \ldots X_n$ there is (a) no "uncanceled show" and (b) Φα occurs "unboxed" and (c) α does not occur free in any line antecedent to this show line, then this part can be changed to

* show $(\forall\alpha)\Phi\alpha$
| X_1
| …
| …
| X_n

$X_1 \ldots X_n$ are now *boxed* and thus are no longer antecedent, and the "show" is *canceled* so $(\forall\alpha)\Phi\alpha$ is antecedent.

The * before the "show" indicates that this goal has been proved; in effect, it *cancels* the "show" and establishes the formula on that line as an antecedent formula (in the technical sense), so that it can now be used in the Rules of Inference. The vertical line on the left of the formulas indicates that these formulas were used in establishing the goal, and are no longer "valid". That is, they are no longer antecedent in the technical sense -- the reason being that they may have "depended upon" an assumption made in the course of that subproof and this assumption is no longer in force now that the goal has been proved. Kalish & Montague call this *boxing* -- they drew a box around these formulas -- and this is a term we will continue to use despite the lack of a box. (We will also call these "scope lines").

Two things here should be noted. First, when a subgoal is proved, *all* lines after the show line are boxed -- even if the "reason for the canceling" is not the last line of the proof as thus far constructed. (This is required for soundness). Secondly, although it is common and expected that the method of proof completion will "match" the type of assumption made -- e.g., assuming the antecedent of a conditional should allow one to cancel because of generating the consequent of the conditional -- this is not required. With or without such a requirement on proof completion, exactly the same arguments are valid.

A proof of Φ from a set of premises Γ is a sequence of lines of formulas and scope lines constructed in accordance with the above, in which Φ occurs unboxed and in which there are no uncanceled show lines. We present here four simple examples to show what proofs look like in the KM. (In each case the formula to be proved (on line 1) is a theorem, so there are no premises).

```
1.  * show (P→Q)→(¬Q→¬P)
2.  |      (P→Q)                ASSUME
3.  |      *show (¬Q→¬P)
4.  |     |  ¬Q                 ASSUME
5.  |     |  ¬P                 2,4 MT
```

```
1.  * show (P v ¬¬¬P)
2.  |     ¬(P v ¬¬¬P)           ASSUME
3.  |   * show ¬P
4.  |   |   P                   ASSUME
5.  |   |   (P v ¬¬¬P)          4,ADD
6.  |   |  ¬(P v ¬¬¬P)          2,R
7.  |   ¬¬¬P           3,DN
8.  |    (P v ¬¬¬P)             7,ADD
```

```
1.  *show (∀x)(Fx→Gx)→((∀y)Fy→(∀z)Gz)
2.  |  (∀x)(Fx→Gx)          ASSUME
3.  |  *show ((∀y)Fy → (∀z)Gz)
4.  |  |        (∀y)Fy                 ASSUME
5.  |  |      *show (∀z)Gz
6.  |  |     |     Fz→Gz               2,UI
7.  |  |     |     Fz                  4,UI
8.  |  |     |     Gz         6,7 MP
```

```
1.  *show (Faaa & a=b)→Fbab
2.  |  Faaa & a=b                      ASSUME
3.  |  Faaa                            2,S
4.  |  a=b                             2,S
5.     Fbab                            2,3LL
```

The KM system, and natural deduction systems generally, have the property that any (legally permissible) line can be entered into the proof at any time and this will not affect the detection of the validity of an argument. That is, even if some line is logically superfluous, so long as it is logically legitimate to be entered, then it can be entered into the on-going proof and be retained in the final proof, and this on-going proof (with the superfluous line) can still be made into a proof of the theorem on the first line of the initial proof. No backtracking of the form which requires deleting already-entered lines is necessary. This may not seem a very striking result until one notices that it is true even for superfluous show lines.[7] Such extraneous subproofs, ones that are "useless" from the point of view of proving a more encompassing show line, even ones that just randomly add an arbitrary show line, can always be boxed, if the outermost encompassing show line is truly provable.

The THINKER programs have a postprocessor (described in Pelletier 1987) which eliminates logically unnecessary lines from a generated proof. The number of lines eliminated varies greatly with the type of problem. In simple proofs (of the sort found in elementary logic books) it tends to be about two lines for every 10 or 15 generated. But some special problems are much more likely

[7] It is also true for adding superfluous lines with free variables, even though it might seem that this would involve violating some existential instantiation or universal generalization restriction later in the proof.

to "fool" THINKER into generating a lot of unnecessary lines. Pelletier & Rudnicki (1986) describe a problem which made THINKER generate 1808 lines of which only 245 were logically required.

The Brains: Heuristics and Strategies

Besides the data structure which contains the actual proof (PROOFMATRIX) and which changes as each new line is added to a proof under construction, there are two important other data structures relevant to the overall proof strategy, corresponding to the two structurally distinct parts of a KM proof. There is a GOALSTACK and an ANTELINES table. Whenever the proof strategies dictate that a new subgoal be added to the proof, it is entered into GOALSTACK, which is a stack because in KM one is always “working on” the most recently-added goal. One does this “work” by adding more and more lines to the antecedent-lines table, ANTELINES, until a certain configuration of the proof is reached…one which would allow the most recently-set goal to be canceled. When that configuration is attained, the most recently-set goal is canceled (becomes proved) and it is deleted from the GOALSTACK and added to ANTELINES. Simultaneously, all entries into ANTELINES which were made after that goal had been initially added to GOALSTACK must be deleted from ANTELINES (they have become "boxed" and are no longer antecedent lines in the proof). When all this happens, the actual proof that is being stored in PROOFMATRIX needs to become updated with indications of "boxing and canceling". This procedure is continued until there are no more members of GOALSTACK, at which time the proof has been completed and the contents of PROOFMATRIX are printed out. If an invalid argument is entered, or if THINKER’s strategies are exhausted and there is nothing else it can do, then there will continue to be an entry on the GOALSTACK and THINKER will ask the user for assistance. At this point the user can inspect the proof as thus far constructed and can choose to take one of three courses of action: (i) enter a new antecedent line, and have THINKER continue construction of the proof using this new line, (ii) enter a new show line and have THINKER continue the proof by making this be a new, most-recent goal, or (iii) quit.

So when an argument to be proved is entered, the premises (if any) are entered into ANTELINES and the conclusion is put on the GOALSTACK. In the normal course of events, the premises do not immediately yield a proof of the goal. Whenever THINKER cannot immediately prove a goal (by methods mentioned below), it has a choice based on the main connective of the formula to be proved: it can either make an assumption or else it can set up one or more subsidiary goals. (THINKER makes sure that any proposed new goal is not identical with one already active. There are no proofs that require the same goal to be simultaneously active at two different points in the proof…although it *can* happen that the same goal needs to be set again at some place after it had already been proved, for it might have been boxed away.) If THINKER is allowed to set a new goal, it uses the following "splitting" strategies: If the main connective is ↔, it sets itself the two subgoals → and ← to be proved independently (and sequentially). When they are proved, and hence become antecedent, THINKER uses the rule CB ("conditionals-to-biconditional") to generate the ↔ formula, and this allows the canceling of the original ↔ goal. Similarly, if the goal's main connective is &, it sets each conjunct as an independent subgoal and after proving them it uses the rule ADJ ("adjunction") to establish the & formula, which in turn allows the cancellation of the original &-goal. If the formula has a universal quantifier as its main connective and the variable of quantification is not free in any antecedent line, then it sets the unquantified formula as a goal. If it proves this, then since the variable (not being free in antecedent lines) was chosen "arbitrarily", the original universally quantified formula goal is canceled and becomes antecedent.

Besides these splitting heuristics, there are other ways for THINKER to set subgoals. But before discussing them let's look at the other option open to THINKER -- the making of assumptions. If the goal formula has any connective other than the foregoing, or if the above subgoals are already active, or if the formula is atomic, then (subject to two provisos to come) THINKER will make an assumption. The assumption depends entirely on the form of the just-added goal, and is added to ANTELINES. Subject to the provisos, this is done as follows: (a) if the goal formula is a conditional, the next line will be the antecedent of that goal, (b) if the goal formula has a negation as its main connective, the next line will be the unnegated formula, (c) in all other cases the next formula is the negation of the goal. In all these cases the PROOFMATRIX contains the annotation "ASSUME". (The two exceptions to this assumption-making are: (i) lines that are already antecedent are not entered again into ANTELINES, so if a proposed assumption already occurs somewhere the assumption will not be made; and (ii) since a conditional goal can be canceled if its consequent occurs in the next embedded scope, before its antecedent is added as an assumption there is a quick check made by the function SIMPLEPROOF (see below) to determine whether this consequent could just be added to the proof without the addition of the assumption.)

In a natural deduction (sub)proof, lines are being added by application of the rules of inference until the right type of line is encountered, and this line then allows us to complete the subproof (cancel the last show line). For any particular show line, there are only a very limited number of lines which could be added that would cancel it. So, it is easy to check whether the most recently-added ANTELINE could generate such a line in one step. This function is called ONESTEP(ϕ), and it investigates whether Φ can generate, in one proof inference, a line that will allow the cancellation of the most recent goal. (If the function succeeds, then it adds this new line and then cancels the most recent goal.) The success of this operation depends on what the most recent goal is, what Φ is, and what is currently in ANTELINES. If, for example, the most recent goal is a conditional, say $(\Theta \rightarrow \Psi)$, then it could be canceled if its consequent were in the proof. So, among other things, ONESTEP(Φ) should look for a conditional $(\Phi \rightarrow \Psi)$ in ANTELINES, so that it can do a MP inference, thereby add Ψ to the proof, and box and cancel the goal. Each different type of goal calls for a search of different types of formulas in ANTELINES (including finding a member of ANTELINES that contradicts Φ, since this is also a legitimate manner of canceling the most recent goal), so there is obviously a premium on being able to find just the right type of formulas to search for. In the example given, we would wish to search the conditionals that occur in ANTELINES to see whether any of them have Φ as an antecedent and Ψ as a consequent. We would want to search the disjunctions in ANTELINES to see whether any of them are $(\neg\Phi \vee \Psi)$. And generally we need to determine whether there is *any* line in ANTELINES which could use Φ to generate Ψ in one step. Whatever the formula is that is currently on the top of GOALSTACK, there are only a small number of ways that it can be canceled; and if we know that in the attempt to cancel it we are going to be using formula Φ, then there are only a very limited number of members of ANTELINES that could be relevant. They just need to be found. In the next section (about data structures) we will see how THINKER manages this type of information, but for now we just note that whenever a new line is added to a proof being constructed, ONESTEP is called with that new formula as an argument. This includes the special case of when the new line "added" is the result of canceling the current show line. ONESTEP is called with the canceled show line formula as argument, to see if this allows us to cancel the next most-recent show line. This general use of ONESTEP turns out to be a *very* powerful heuristic: always look to see if the most recently generated line will immediately prove the most recent goal in one step.

Another function is SIMPLEPROOF(Φ), which attempts to find a one-step proof of Φ. This is different from ONESTEP(Φ), which attempts to *use* Φ to find a one-step proof of the most recent goal. There are three pieces of information that are usually relevant when drawing a "non-blind" inference: the formula you desire to prove, and the two formulas you intend to use to generate it (since most inference rules require two antecedent lines). ONESTEP(Φ) is provided with two of the three pieces of information and looks into ANTELINES to determine whether, for any of the rules of inference, there is a third formula which can be used. SIMPLEPROOF(Φ) instead has only one of the pieces of information: that we are trying to generate a one-step proof of Φ. So, it needs to find two entries in ANTELINES for any of the rules of inference. So SIMPLEPROOF is a much more difficult function to compute, and its use is accordingly restricted. It is used only when it is known that if Φ could successfully be added to the proof, then there would be an immediate way to cancel the most recent goal. For example, it is SIMPLEPROOF(Ψ) that is called immediately after a goal of the form (Φ→Ψ) is set, for if it is successful then that goal can be canceled. (As indicated above).

FORWARD is a "blind" procedure which attempts to apply the various rules of inference more or less blindly to lines that are in ANTELINES. It is made up of a set of procedures that are called "FINDxx", where 'xx' is the name of one of the inference rules (such as DN or UI or MP, etc.…standing for "double negation" or "universal instantiation" or "modus ponens", etc.). It is not completely blind, however. For one thing it only applies "simplifying rules" such as &-elimination, and does not apply complexity-introducing rules such as ∨-introduction. For another thing, the way it deals with existentially and universally quantified formulas in ANTELINES is rather careful (see below). And for a third thing, each time FORWARD adds a new line "blindly", ONESTEP is called with that new line as argument. If ONESTEP succeeds, then the FORWARD procedure is exited. The FORWARD procedure could apply to a formula, and the resulting formula might have that same rule be reapplied. The proof algorithm described in Appendix I details the flow of control.

Natural deduction systems have a very general problem in dealing with formulas in a proof that have an existential quantifier embedded inside a universal quantifier. But even for the less difficult case where there are just a number of existentially quantified formulas, one wants them instantiated only once under any subproof level. And that includes the case where there are further embedded subproofs: one wishes to instantiate them as early as possible (so that universally quantified formulas can then be instantiated to those values) and then not re-instantiate them ever, unless the level at which they were instantiated becomes boxed away.

A standard strategy for dealing with universal quantifiers is to instantiate them to every name or ground term or free variable[8] that is currently active in the proof. But if an existential quantifier is embedded within the universal quantifier, it cannot be eliminated until *after* the universal quantifier. Typically (in Quine systems), the existential quantifier is eliminated by instantiating it to a *new* variable (it can't be instantiated to an old one, since then it would not be "arbitrary"). But having the new variable in the proof permits the strategy for universal quantifiers to come into play again: it should be re-instantiated, this time to the newly-introduced variable. And this re-introduces an existentially quantified formula into the proof which needs to be instantiated to a new variable, etc. As can be seen, some method of dealing with the threatening infinite "loop" of instantiations is required. Before stating how THINKER deals with this, I wish to emphasize that there are valid arguments which cannot be proved to be so unless at least *some* circles through this "loop" are

[8] In those Quine-style systems that allow free variables to be the value of existential instantiation.

made. That is, one cannot simply say that “no universally quantified formula shall be instantiated to a variable introduced by an existential formula which was in turn introduced by an instantiation of that very universally quantified formula,” for sometimes one *must* do this to detect valid arguments. There are furthermore valid arguments that come about from *pairs* (or triples, etc.) of formulas of this general form (with an existential embedded within a universal quantifier). To see this latter issue, consider a blind strategy that instantiates each of the two universally quantified formulas to some term; then the two resulting existentially quantified formulas would each be instantiated to new variables. But this would make the original universally quantified formulas become salient again, and require that they be instantiated to these new variables. And even if we managed to keep track of which of these new existentially-introduced variables was due to which of the universally quantified formulas as an ancestor, and we were to block the new universal quantifier instantiation to that variable, still the *other* universally quantified formula would be eligible to be instantiated to that new variable. And that could give rise to an infinite chain of "crisscross" instantiations. Yet once again, there are arguments which require that at least *some* of the crisscross instantiations be performed in order to show them valid, so we cannot merely outlaw all crisscross instantiations.

THINKER’s solution to this problem is to maintain a list of “prohibited variables” and a list of “prohibited ancestors”. The prohibited variables are those which arose from an existential instantiation of a formula which in turn had a universally quantified formula as an ancestor. The prohibited ancestor is this universally quantified formula. In the normal course of doing the FORWARD inferencing, prohibited variables and ancestors may arise during existential instantiation, and they are marked as such when they are generated by the FINDEI procedure. At any pass through the entire proof strategy, no member of the set of prohibited ancestors is permitted to be instantiated to any of the prohibited variables. But after all other strategies are pursued and do not succeed in canceling the current goal, then another instantiation round is permitted to take place (this is governed by a flag, UIPROHIB, which keeps track of whether all other strategies have been tried), followed once again by all the other available strategies. This is continued until there are no more variables that the strategy can use. As remarked above, this re-instantiation *must* happen in some cases to detect valid arguments. THINKER has a finite store of variables it can use, so ultimately this will quit and THINKER will go on to other strategies or request help from the user. There are 20 such variables by default, although more (or less) can be requested by the user when running the program. But whatever number *n* is specified, it is finite; and therefore any proof which requires more variables (such as one asserting that the existence of $n+1$ individuals follows from premises that claim that *n* objects exist) will be misclassified in terms of its validity.

Universal Instantiation is therefore implemented by having two separate procedures. One difference between them has to do with which universally quantified formulas they apply to, and the other to do with which variables these formulas are instantiated to. The procedure FINDUIPROHIB uses the set of prohibited ancestors; and it instantiates them *only* to the set of prohibited variables. This happens once on each application of the entire set of heuristics. The other procedure, FINDUI, can apply at many times during any one application of the heuristics. It takes the following form: For each universally quantified formula in ANTELINES, (i) instantiate it to every constant and free variable that is active in the proof (plus premises)[9], except do not instantiate prohibited ancestors to variables on the prohibited list, (ii) if there are no such active terms and there is nothing on the prohibited list of variables, then instantiate it to the variable of quantification of the most recent active universally quantified show line, and (iii) if none of these are applicable, then pick some arbitrary variable. Of course, there will be valid arguments that cannot be detected because of the finite number of variables available…but what else can be expected in the case of first order logic? A more explicit account of the FORWARD procedure is given in Appendix I.

[9] Although this sounds like a very wasteful and computationally expensive way to proceed, experience with THINKER has not borne out this feeling. Perhaps it is because ONESTEP checks as each new instance is entered whether it will generate a one-step proof of the most recent goal, that many proofs are stopped before all the possible instantiations are entered. And because of the post-processing proof condensor, the user is not bombarded with a huge number of superfluous instantiations. Pollock (1997) reports the same result from his experience with the Oscar system. One could instead employ a strategy of instantiation to a most general unifier akin to Sieg & Kauffmann (1993), but although this might have some nice metatheoretic properties (as guaranteeing that the appropriate instance is described in the minimal space), it is not clear that the other overhead costs (e.g., having to check for unification "on the fly") really is any practical savings. Such claims at least need to be empirically tested.

As suggested by remarks above, to apply a rule of inference when one does not have any specific conclusion in mind[10] (as is done by the FORWARD procedure) requires the discovery of two lines in ANTELINES which are each instantiations of the preconditions of some rule of inference. The next section (on data structures) outlines a general method which allows this to be done much more efficiently than just looking through the ANTELINES for a pair of lines that happen to have a rule of inference applicable to them.

TRYNEGS is a strategy that is used when more direct approaches fail. It searches ANTELINES for an occurrence of the negation of a conditional, the negation of a biconditional, and the negation of a disjunction.[11] If there is one such, and if the IND flag is on (indicating that an indirect proof is underway), then THINKER will set the unnegated formula as a new goal. (Actually, in the case of a disjunction, just one of the disjuncts is set as a goal.) The idea is that if the unnegated formula can be proved then it will contradict the negation which was found in ANTELINES, and this contradiction will allow the cancellation of the most recently-set goal.

CHAINING is another "blind forward deduction" technique that THINKER uses when other strategies fail. It searches ANTELINES for a conditional for which the consequent of that conditional is not also in ANTELINES. If it finds one of these, then it will set the antecedent of the conditional as a new goal. If it can prove this antecedent, then it can do a modus ponens which it could not do before, and so FORWARD can be called again to do new inferences. A similar strategy involves looking for a conditional the negation of whose antecedent is not also in ANTELINES. If it finds one such it will set the negation of the consequent as a new goal; and if this can be proved, then there is a new modus tollens which can be performed. And finally, there is a similar strategy involving disjunctive formulas in ANTELINES where neither disjunct also occurs in ANTELINES. In this case

[10] Of course, there is always *some* specific conclusion in mind, namely, the most-recently set goal. But here I mean that there is no way to generate a proof of this goal by ONESTEP and the lines that are in the proof at the present time, and there is no directly-applicable heuristic which will set some new goal.

[11] It is not useful to apply this strategy to the negation of atomic formulas because the effect then would be just to set the unnegated atom as a goal and then to assume its negation. But since the negation is already in the proof this will not generate any new lines.

the negation of one of the disjuncts is set as a new goal. If this can be proved then there is a modus tollendo ponens that can be performed which couldn't before.

Here's a problem in the propositional logic to illustrate the operation of the various components of this proof strategy. The problem is to show the equivalence between disjunction and the conditional (with a negated antecedent). The problem is not so simple as might be expected from the fact that resolution-style provers eliminate it in the conversion to clausal form; and it is not especially simple in KM, despite the fact that in other natural deduction provers this equivalence is given as a primitive rule. In fact, elementary logic students using systems that do not embody this (and related equivalences) as primitive, will find it challenging to produce a proof of it.

The problem is to prove $((P\vee Q)\leftrightarrow(\neg P\rightarrow Q))$ to be a theorem. THINKER's proof procedure puts the problem on the goal stack, and recognizes it as a biconditional problem. It therefore sets itself the subgoals of showing each of the conditionals. So it sets one of them as the next subgoal and recursively calls the proof procedure on that show line. When it finishes this subproof it will then set the other conditional as a goal to be proved and recursively call the proof procedure on *that* show line. Since the first of these subgoals is a conditional, it adds the formula's antecedent into ANTELINES with the annotation "ASSUME". (Actually, as indicated in the discussion above, SIMPLEPROOF is first called to see whether the consequent of this conditional could be proved in one step. It can't, of course, since there is nothing in ANTELINES yet.) After making that assumption the proof looks like this:

1.   show $((P\vee Q)\leftrightarrow(\neg P\rightarrow Q))$
2.   show $((P\vee Q)\rightarrow(\neg P\rightarrow Q))$
3. $(P\vee Q)$ ASSUME

The insertion of $(P\vee Q)$ into ANTELINES gave rise to ONESTEP($P\vee Q$), which tries to prove the most recent goal in onestep using $P\vee Q$. This fails, as does SIMPLEPROOF($\neg P\rightarrow Q$), which tried to prove $\neg P\rightarrow Q$ in one step. So the consequent of the conditional on line 2 now is set as a new goal. But since it is itself a conditional, its antecedent is added as an assumption. So now the proof looks like:

1.   show $((P\vee Q)\leftrightarrow(\neg P\rightarrow Q))$

```
2.        show ((P∨Q)→(¬P→Q))
3.             (P∨Q)              ASSUME
4.             show ¬P→Q
5.                 ¬P             ASSUME
```

Of course, when line 5 was added, ONESTEP(¬P) was called. This succeeds in finding a one step inference that will allow the cancellation of the most recent goal (line 4). It does this by applying the rule MTP to lines 3 and 5, adding Q to the proof, then canceling line 4 and boxing lines 5 and 6. But now that line 4 is in ANTELINES, it is subject to ONESTEP; and ONESTEP(¬P→Q) succeeds because the very line 4 itself is a reason to cancel line 2. So, line 2 is canceled and the lines 3 through 6 are boxed. The proof now looks like this:

```
1.    show  ((P∨Q)↔(¬P→Q))
2.        * show ((P∨Q)→(¬P→Q))
3.         |     (P∨Q)            ASSUME
4.         |   * show ¬P→Q
5.         |    |   ¬P            ASSUME
6.         |    |   Q             3, 5 MTP
```

Since line 2 is proved, THINKER is ready to set the other conditional as a goal. And it will assume the antecedent of this conditional, and (since ONESTEP and SIMPLEPROOF fail) will set the consequent as a new goal and then assume its negation. So now the proof looks like this:

```
1.    show  ((P∨Q)↔(¬P→Q))
2.        * show ((P∨Q)→(¬P→Q))
3.         |     (P∨Q)            ASSUME
4.         |   * show (¬P→Q)
5.         |    |   ¬P            ASSUME
6.         |    |   Q             3, 5 MTP
7.         show ((¬P→Q)→(P∨Q))
8.             (¬P→Q)      ASSUME
9.             show  (P∨Q)
10.                ¬(P∨Q)         ASSUME
```

FORWARD inference has failed: there are no inferences that can be made with the three antecedent lines (lines 2, 8, 10). TRYNEGS notices that line 10 is the negation of a disjunction and that neither of the disjuncts is in ANTELINES, so it sets one of the disjuncts as a goal. It then assumes the negation of this goal, and the proof looks like this:

```
1.    show  ((P∨Q)↔(¬P→Q))
2.        * show ((P∨Q)→(¬P→Q))
3.         |     (P∨Q)            ASSUME
```

```
4.        |   * show (¬P→Q)
5.        |   |  ¬P               ASSUME
6.        |   |  Q                3, 5 MTP
7.        show ((¬P→Q)→(P∨Q))
8.            (¬P→Q)      ASSUME
9.            show  (P∨Q)
10.              ¬(P∨Q)           ASSUME
11.              show   P
12.                    ¬P ASSUME
```

Although ONESTEP does not succeed with the new line 12, this new line's presence allows the FORWARD procedure to continue. Lines 8 and 12 will generate a Q by modus ponens, and ONESTEP(Q) succeeds because doing one application of ∨-addition, this contradicts line 10. So, the result of the addition rule is entered into the proof, and line 10 is repeated so that it occurs in the same scope as the line it contradicts. This justifies canceling line 11.

```
1.       show  ((P∨Q)↔(¬P→Q))
2.         * show ((P∨Q)→(¬P→Q))
3.        |    (P∨Q)              ASSUME
4.        |   * show (¬P→Q)
5.        |   |  ¬P               ASSUME
6.        |   |  Q                3, 5 MTP
7.         show ((¬P→Q)→(P∨Q))
8.            (¬P→Q)      ASSUME
9.              show  (P∨Q)
10.                 ¬(P∨Q)        ASSUME
11.                 * show   P
12.                 |  ¬P ASSUME
13.                 |  Q          8, 12 MP
14.                 |  (P∨Q)      13, ADD
15.                 |  ¬(P∨Q)     10, R
```

But now ONESTEP(P), using the just-canceled line 11, will succeed, since by an ∨-addition it will contradict line 10, and thereby cancel line 9. So this addition is performed and added to the proof and line 9 is canceled (with all successive lines boxed). But now that line 9 is in ANTELINES, doing a ONESTEP on it will justify the cancellation of line 7. And finally, now that lines 2 and 7 are both in ANTELINES, they will be used to introduce the biconditional, which justifies the cancellation of line 1. And the final proof looks like this:

```
1.     * show  ((P∨Q)↔(¬P→Q))
2.     |  * show ((P∨Q)→(¬P→Q))
3.     |  |    (P∨Q)              ASSUME
4.     |  |   * show (¬P→Q)
```

```
5.    | | | ¬P                   ASSUME
6.    | | | Q                    3, 5 MTP
7.    | * show ((¬P→Q)→(P∨Q))
8.    | |   (¬P→Q)      ASSUME
9.    | |   * show  (P∨Q)
10.   | |   |   ¬(P∨Q)           ASSUME
11.   | |   |   * show   P
12.   | |   |   | ¬P ASSUME
13.   | |   |   | Q              8, 12 MP
14.   | |   |   | (P∨Q)          13, ADD
15.   | |   |   | ¬(P∨Q)         10, R
16.   | |   |   (P∨Q)            11, ADD
17.   | ((P∨Q)↔(¬P→Q))           2, 7 CB
```

The reader might consider some of the proofs in Appendix II, keeping in mind the heuristics involved here and the flow of control (indicated in the flow diagram, also in the Appendix I).

The Guts: Data Structures and Low-Level Routines

As can be concluded from the preceding discussion of THINKER's proof strategies, there are various pieces of information that need to be readily available. The point of this section is to describe some of the more important data structures that facilitate this. THINKER was originally written in the early 1980's, and that version was written in Spitbol (a dialect of Snobol4). It was later (mid-to-late 1980's) rewritten in C, with an eye to increasing portability. The original choice of Spitbol was due to certain "philosophical views" about logic's being "really" a matter of pattern matching. And in fact the original set of Spitbol programs, running on an Amdahl mainframe computer, still (in my mind) contains the most interesting set of data structures (and operated at least as fast as the C version does on Sparc stations). In fact, the conversion to C attempted to mirror all these data items, but was unsuccessful at various points (which I will indicate shortly). Here I will describe the original Spitbol vision.

Spitbol is a string-oriented language. The basic command in the language is: "does string S match pattern X? --if yes then branch to line n, else branch to line m." Formulas in THINKER are stored as strings, rather than trees or linked lists of pointers, etc. Spitbol provides a wide range of functions that can deal with these strings. THINKER defines a SPLIT function, which succeeds with any non-atomic formula, storing the formula's main connective (propositional connective or quantifier) into a variable OP, if the main connective is a quantifier it stores the variable of quantification into VAR, and it stores the subformula(s) into LOP (for negations and quantified formulas) and also in ROP for binary connectives. Also defined are patterns which will quickly determine the locations within a formula of all the variables bound by some particular quantifier. (In KM there is no requirement that all quantifiers use distinct variables, nor is there any prohibition against free variables; so the task of finding occurrences bound by a quantifier is not entirely trivial.) In this way the structural properties of a formula can be determined extremely quickly. As indicated in the last section, we need to have a list of all the constants and free variables active in the proof at any given time. And we need to have global variables that maintain what the current line number is of the proof as thus far constructed (CURLINE), and also what the current depth of embedding of subproofs is (CURLEVEL). (Recall that boxing and canceling will affect this latter variable).

The proof as it is being constructed--the formulas being entered, the line numbers of these formulas, their annotation, the scope indications, etc.--is maintained in a PROOFMATRIX. Once a proof line is entered here, it never is removed. (But facts about whether a subproof is or isn’t currently proved can change, naturally). But all the action happens with two other data structures: GOALSTACK and ANTELINES. The former maintains the stack of “show lines” (and the value of CURLEVEL); the latter contains the list of currently active antecedent lines. As was described in the last section, when a subproof is completed, the lines added to the PROOFMATRIX (and also to ANTELINES) after the setting of the most recent goal must now be boxed--i.e., a scope line is added to them in PROOFMATRIX and they must be deleted from ANTELINES. So, we needed to remember the CURLINE value when this last show line was added, and we need to box all lines between then and the current value of CURLINE. And all these must be deleted from ANTELINES--a somewhat more difficult task because they are not stored sequentially in ANTELINES (see the next paragraph). Further, the subgoal which is proved must be popped from the GOALSTACK and then added to ANTELINES.

One of the useful primitive data types in Spitbol is the Table, which makes use of built-in hash functions on strings to index storage locations. THINKER often needs to know the answer to a question such as: “Does formula X occur in the antecedent lines at this stage of the proof?”, and the information about whether it does, and where in the proof it is, is stored in the Table at the location indexed by the formula X. Thus finding the answer to this question can be done in constant time, regardless of the size of the proof. (Spitbol has dynamic Table sizing, so that if too many collisions occur, the Table size is increased. This is one of the features of Spitbol that we were unable to recreate in C.) Items are “removed” from Tables (such as the ANTELINES Table) by keeping the formula as a valid index to the Table, but changing the value of the entry of the table to indicate that it is no longer an antecedent line.

Not only are the strings representing the formulas used as indices to the ANTELINES table, but so also are TEMPLATEs. A TEMPLATE is a schematic representation of a formula (in exceptional cases, the same TEMPLATE can represent more than one formula in a proof)–schematic in that some feature of the formula has been omitted and replaced by ‘@’. Thus, $F(x,@)$ is a string which represents the class of actual formulas that have x standing in the F relation to something…without saying what that something is. $(@\rightarrow P)$ represents the class of formulas that have ‘$\rightarrow$’ as main connective and P as consequent. THINKER uses such TEMPLATEs like this: whenever a formula--say, $((A\&B)\rightarrow C)$ -- is added to the proof as an antecedent line (hence, into ANTELINES), THINKER also stores various templates it can form from it. For this formula it constructs the TEMPLATEs $((A\&B)\rightarrow @)$ and $(@\rightarrow C)$, thereby remembering that $(A\&B)$ is the antecedent of some conditional statement and that C is the consequent of some conditional statement. These are strings, and can therefore be used to index the ANTELINES table. Their value, however, is not the same as an ordinary formula’s entry; it is instead a list of all the specific formulas that it is a TEMPLATE for. These specific formulas can then be used as indices into ANTELINES to discover where in the proof they are. Thus it is a constant time matter to discover whether there is any formula in the proof with such-and-so structural feature (at least for the features that have been deemed important enough to represent as TEMPLATEs). THINKER keeps TEMPLATEs for the following types of formulas that have been entered into ANTELINES:

1. For every formula, there is a template generated by replacing all occurrences of exactly one free term (constant or free variable) by @. (*P(x,y)* therefore has two TEMPLATEs: *P(@,y)* and *P(x,@)*, but there is no TEMPLATE *P(@,@)* generated by this formula. It would have to come from *P(x,x)*.)
2. Where Q is a quantifier, formulas of the form $(Q\alpha)\Phi\alpha$ generate the TEMPLATE $(Q@)\Phi@$, where the replacement of α by @ in $\Phi\alpha$ observes scope requirements, of course. (A formula such as $(\exists x)F(x,y)$ would therefore generate two TEMPLATEs: $(\exists @)F(@,y)$ and $(\exists x)F(x,@)$, where the second TEMPLATE comes by means of the method mentioned in (1).)
3. A formula whose main connective is binary has the form $(\Phi\bullet\Psi)$, and it generates the two TEMPLATEs, $(\Phi\bullet@)$ and $(@\bullet\Psi)$.
4. No other formulas generate TEMPLATEs.

With these TEMPLATEs, the sorts of information required by various of the search methods mentioned in the previous section can easily be found. Generally speaking, the effort required for search is constant; the space required for storage of this type of information approximately triples. (Typically, a new ANTELINE will generate an entry for the formula itself and two further TEMPLATEs).

The final data type to be mentioned are the RINGs. Each type of connective and quantifier has its own RING. Thus (for example) a formula whose main connective is a conditional is added to the circular linked-list called →RING. (Only actual formulas from ANTELINEs, not the TEMPLATES they generate nor their subformulas, are added to the RINGs). When a line ceases to be an ANTELINE because it has been boxed, then that formula is deleted from the appropriate RING. The RINGs are used by a set of functions called FINDxx, where 'xx' is the name of some rule of inference. (This was mentioned in the last section as being a part of the "blind" FORWARD procedure). If the FORWARD procedure has called FINDDN (perform double negations), then the ¬RING (which contains all formulas in ANTELINES that start with a ¬) is traversed to locate those formulas which have an immediately embedded ¬. It eliminates these two negation signs, adds the result to ANTELINES (thereby also constructing the appropriate TEMPLATES and adding this new formula to the relevant RING, etc.), and calls ONESTEP on this newly-added formula. This ¬RING is traversed until there are no more formulas left in the ¬RING, and at that time FORWARD calls upon some other FIND procedure. The ¬RING is used by the FINDQN (quantifier negation) procedure as well as the FINDDN procedure. The FINDMP and FINDMT procedures use the →RING; the FINDMTP procedure uses the ∨RING; the FINDBC procedure uses the ↔RING; the FINDEI procedure uses the ∃RING; and the FINDUI procedure uses the ∀RING.

Additionally, the ¬RING is used by the TRYNEGS procedure discussed in the previous section. In this procedure, we are searching for negations of biconditionals, of conditionals, and of disjunctions with an eye to perhaps setting some other formula as a goal. So it will traverse the ¬RING, looking at the main connective of the immediately embedded, unnegated formula. When it discovers that this immediately embedded formula is a conditional (for example), it puts this unnegated formula on GOALSTACK.

The CHAINING strategies also use the RINGs. This strategy will look (for example) in the →RING to find a conditional, and then probe into ANTELINES using the consequent of that conditional as an index, to determine whether this consequent is already in ANTELINES. If it isn't then CHAINING sets the antecedent of this conditional as a new goal on GOALSTACK. The strategy also looks into the ∨RING to find a disjunction, and then probes into ANTELINES to determine whether or not one of the disjuncts is already there. If neither of them is there then: if neither is an active goal it will set one of them as a goal on GOALSTACK and if one is but the other isn't already a goal, it is the latter which will be set as a goal.

Thus applying these CHAINING strategies is pretty much a constant-time matter, depending only on how big the relevant RING is–that is, depending only on how many conditionals (or whatever) are currently active in ANTELINES, and not on how many other formulas there are in ANTELINES. The cost for this efficiency in search is a matter of memory: yet another copy of the formula is to be stored on the relevant RING, over and above the one in ANTELINES, the one in the

PROOFMATRIX, and the various TEMPLATES that are constructed from the formula and stored in ANTELINES.

Identity

Now that the overall proof procedure and the various data structures have been explained, it is possible to understand the issues involved in adding identity to THINKER. The identity rules employed by THINKER follow those of Kalish, Montague, Mar (1980) rather than those of the original Kalish & Montague (1964). In this development of the logic of identity, only the two rules REFL and LL (listed above) are employed, but I found it helpful to have NEGID also available.

The presence of REFL in the system suggests some simple alterations in various areas of the proof strategy. For example, if we are trying to prove "x=x", we should just recognize this as an instance of REFL. Or if we are trying to prove "$(\exists x)x=y$" we should recognize that "y=y" could be added as a line by REFL and this formula can have the rule EG applied to it, thereby yielding the desired formula. Furthermore, if we have a line such as "$(x=x\longrightarrow\phi)$" in the proof, we should recognize that "x=x" can be added by REFL and then modus ponens could be used to yield ϕ. I call these type of inferences "special cases".

Recognition of special cases needs to be added to ONESTEP, to SIMPLEPROOF, and to FINDxx. In particular we should add as special cases of FIND the following:

> In FINDMP, if the formula $(\alpha=\alpha\longrightarrow\phi)$ is encountered in the $\longrightarrow$RING, add ϕ to ANTELINES.
> In FINDMT, if the formula $(\phi\longrightarrow\alpha\neq\alpha)$ is encountered in the $\longrightarrow$RING, add $\sim\phi$ to ANTELINES.
> In FINDMTP, if the formula $(\alpha\neq\alpha \vee \phi)$ is encountered in the $\vee$-RING, add ϕ to ANTELINES.

With regard to SIMPLEPROOF(ϕ), recall that we are trying to add ϕ to the proof in one step. The special cases are (where α is some term and x is some variable):

ϕ is $\alpha=\alpha$, ϕ is $(\exists x)x=x$, ϕ is $(\exists x)x=\alpha$

A call to SIMPLEPROOF always specifies the argument ϕ, which we are trying to add to the proof. Should it get called with one of these three types of argument, then it will just add the formula to ANTELINES with an appropriate annotation.

Finally, with regard to ONESTEP(ϕ), recall that we are given two pieces of information: the formula ϕ (typically the most recently formula entered into the proof) and the most recent subgoal Ψ. The function succeeds if Ψ can be proved from ϕ in one step. Here the special cases are:

ϕ is $\alpha\neq\alpha$, Ψ is $\alpha=\alpha$, Ψ is $(\exists x)x=x$, Ψ is $(\exists x)x=\alpha$

If any of these four things are happening when ONESTEP is called, then the relevant lines are added to the proof and Ψ is marked as proved.

In addition to the special cases, identity is used "normally" in each of ONESTEP, SIMPLEPROOF, and FINDxx. The first two are straightforward. If ONESTEP(ϕ) is called with current goal Ψ, then the three following cases might arise ('$F\alpha$' is some sentence that has α free.).

1. ϕ is $\alpha=\beta$, Ψ is $F\alpha$ (or equivalently, Ψ is $F\beta$)
2. ϕ is $F\alpha$, Ψ is $F\beta$
3. ϕ is $F\alpha$, Ψ is $\alpha\neq\beta$ (or equivalently, ϕ is $F\beta$)

In case (1) we wish to use the identity $\alpha=\beta$ to prove $F\alpha$, so we should search ANTELINES to see whether $F\beta$ is in it. If so, then we should enter $F\beta$ into the proof and annotate it appropriately. Case (2) is similar, except that we are here searching for the identity $\alpha=\beta$. In case (3) we are trying to prove a non-identity using $F\alpha$, so we should search for $\neg F\beta$ and use NEGID. (If $F\alpha$ is itself a negation, then we should in addition look for the unnegated version of this formula with β replacing α).

In SIMPLEPROOF(ϕ) we are trying to add ϕ to the proof in one step. These two cases are relevant to identity:

ϕ is $F\alpha$, ϕ is $\alpha\neq\beta$

In the former case we wish to search ANTELINES for two formulas: $F\beta$ and $\alpha=\beta$. If we have them, then $F\alpha$ can be added by LL. In the latter case we search ANTELINEs for any formula that has α free and also for the negation of that same formula except that it has some occurrence(s) of β where the former one had α. If we find such a formula, we add $\alpha\neq\beta$ to ANTELINES by the rule NEGID.

This leaves only a description of how identity is handled generally in the FINDxx area of the proof procedure. (We've already seen how the various "special cases" are added to FINDMP, FINDMT, and FINDMTP). In essence there are two new FIND procedures: FINDLL and FINDNEGID.

The former is a function that searches ANTELINES for an identity, and finding one, it searches ANTELINES for a formula that contains one of the terms of the identity free and performs a substitution in accordance with LL. The latter function searches the ~RING for negated formulas with a free term α, and when it finds one it then tries to find an unnegated formula in ANTELINES just like the first except for having some other term β free in it instead of α. If it finds such a pair it enters $\alpha\neq\beta$ into the proof by NEGID.

Identity in automated theorem provers can give rise to a number of problems of proof development and of search, causing even logically simple proofs to involve a "combinatorial explosion" in the checking of such things as "is there a formula just like $F\alpha$ except for having some other term β free in it instead of α?" and the like. Here is a simple difficulty that must be overcome in some way by every automatic theorem prover that is going to handle identity. If THINKER is given the two formulas

$(a=b\longrightarrow Fc)$, $b=a$

as antecedent lines, it cannot immediately perform MP, since the second formula is not string-identical with the left-hand-side of the first. In THINKER the proof strategy would have to decide first to set itself the task of proving that from these lines we can prove $a=b$ (or alternatively, prove $(b=a\longrightarrow Fc)$). Not only is it very difficult to know when this should be done, but also if there are numerous lines with embedded identities, the attempts to prove such things raises the questions of which order to attempt the subproofs and what to do if one of them cannot be proved. A related problem in THINKER, as well as in other systems, manifest itself as the question: "How do we know that $a=b$ and $b\neq a$ are contradictory?" Different theorem provers have different ways to avoid these problems, all of which amount to saying that "order of stating an identity does not matter". THINKER's solution is along these lines also. It is:

For every identity statement, write it in a fixed order (lexicographic order)

I call this "normalizing identities." It means that, for example, $b=a$ will never appear in a proof. It can be entered (as a premise to an argument) but the program will store it (and always print it) as $a=b$. This solves the two problems mentioned here: In the displayed argument, $b=a$ never appears but is instead written as *a=b* and the MP is performed. In the case of contradictions, *b≠a* is represented as *a≠b* and the contradiction is noticed immediately.

Another problem with identities is this. Suppose we have already these two lines in ANTELINES:

Fa and *a=b*

Obviously we can infer *Fb* by LL. But *should* we? After all, any information in *Fb* is already in *Fa* (so long as we remember that a=b). The only reason to want *Fb* if it was explicitly required to justify the finishing of a subproof or explicitly required for application of some other rule of inference. If we *do* introduce *Fb* into the proof we now have two formulas to keep track of (in ANTELINES, in the TEMPLATEs, in the RINGs, etc.) and do inferences with respect to. Related to this is the problem of "order of substitutions" with LL. Consider a proof that has these three formulas:

(Fa⟶Gx), a=b, Fb

from which we wish to infer *Gx*. We could use the second formula to do LL with the first formula (yielding *(Fb⟶Gx)*) and then an MP with the third formula to yield *Gx*. Or, we could use the second formula to do an LL with the third (yielding *Fa*) and then an MP with the first formula to yield *Gx*. But if we do both then we will have both of

(Fb⟶Gx), Fa

in the proof (and in ANTELINES, and TEMPLATEs, and in the RINGs). We would like a way that allows us not to do all possible substitutions, but somehow restricts them to some smaller subset that is still adequate (i.e., complete).

THINKER's solution is to maintain identities as equivalence classes. Each equivalence class is given a "representative name", namely the lexically lowest of the terms which are in that class. Except for when the current goal can be proved in one step of LL done in a different way, whenever an LL is to be performed, it "aims at" this representative name. For example, if the proof contained the identities *a=b* and *b=c* this would be stored as an equivalence class whose name was *a*. If a formula had a free *b* or *c* in it, and an LL was to be performed, the result would always be represented with *a* in it (unless it was explicitly required to use one of the other names, as for instance to complete a subproof). When a new identity is added to a proof, THINKER computes the (identity) transitive closure of the two terms. Again for example, if $a=b$ and $b=c$ were already in the proof (with no other identities), THINKER would create [a], the transitive closure of these identities with the representative name *a*. If, somehow, the lines $d=e$ and $e=f$ got into the proof, THINKER would create [d], the equivalence class with the representative name *d*. If now the identity $b=f$ (for example) were added to the proof, THINKER would merge the two earlier equivalence classes into one and give it the representative name *a*. Given an equivalence class, and given a formula that mentions one of the equivalent terms, FINDLL will generate a new formula which replaces that term by the equivalence class's representative name. This would solve the two problems mentioned here: in the second problem, the LL which is chosen would be with the second and third formulas, and the other possible LL would never be performed. In the first problem *Fa* and $a=b$ are not subject to LL, since *a* is lower than *b*.

When a subgoal has been proved, THINKER may have to destroy an existing equivalence class and re-establish earlier ones. If, in the example of the last paragraph, [a] and [d] were established before the setting of a subgoal, and the formula $b=f$ is added after this setting, then the merging of the two equivalence classes to yield the new [a] is no longer valid after this subgoal is proved. We must at that time revert to our earlier [a] and [d]. THINKER keeps track of this by maintaining a list of valid equivalence classes at teach level of subgoals.

So the two new additions to the FINDxx section of the proof procedure are:

FINDLL For each equivalence class with more than one member, find every formula in ANTELINES that contains a lexically greater member of that class, replace these lexically greater names by their representative name, and add the new formula to ANTELINES.

FINDNEGID For each formula $\neg F\alpha$ in the ¬RING, and for each other "active" term β, replace occurrences of α by β and see if $\phi\beta$ is in ANTELINES. If it is, add $\alpha\neq\beta$ to the proof.

Clearly these new procedures, especially FINDNEGID, involve quite a bit of additional computation. Since the replacement and search is so time-consuming it was decided to start each proof with a pre-scan to determine whether there were any occurrences of '=' in premises or conclusion. If there aren't any, then none of the identity rules will even be attempted. (This never misses any valid arguments. If there are no identities in the premises and conclusion of a valid argument, then there is a proof for that argument that never mentions identity in any of its lines.)

A Concluding Unscientific Remark

There has been no theoretical analysis of THINKER's proof search algorithm, although as I mentioned above it cannot be complete due to the finitude of the various types of representations it employs. But it is not clear whether, given unlimited resources, every valid formula would receive a proof. Some method along the lines of Sieg and his colleagues (1992, 1993, 1996), as mentioned above in footnote 3, seems to be called for in this regard. On the other hand, an informal look at the proof algorithm in Appendix I shows that it must be sound, since every formula entered into the proof is justified by some rule of inference.

Despite the lack of theoretical investigation into THINKER's completeness, I find it surprising just how many arguments of a moderately high degree of difficulty (at least to judge from elementary logic books) can be proved with just this simple strategy.[12] Of course, proponents of natural deduction systems claim that this is due to the retention of the natural form, the separation of distinct methods for dealing with all the different types of formulas that one might need to prove, the ability (in a natural deduction format) to recognize easily what the relevant subproblems are, and

[12] It proves all the problems without function symbols in Pelletier (1986/7). It is difficult to assess THINKER's completeness because it can arrive at step J (of the algorithm given in the Appendix) in various different ways, and even in those cases where the problem is logically valid, THINKER might have got there because it has run out of variables. Of course, running out of variables is a "logically uninteresting" reason for the algorithm to quit. But although it is difficult to believe that any of these elementary problems just require more variables, there has never been any analysis of whether or not this is true. There have also been some problems that get timed out by the operating system after 20 seconds, but have not reached step J.

the ability to be able to prove these subproblems and then use the conclusions that these subproofs generated in the proof of the main conclusion. The problem with formats such as resolution, proponents of natural deduction claim, is that in the conversion to a normal form all information is lost about what the relevant subproblems are…or indeed even what the original conclusion was.

There has certainly been much intellectual effort (and money and public relations) expended on resolution-style proving over the last 30 years. I would wish to encourage those involved in that effort also to use some of their expertise gained over these decades in the exploration of natural deduction automated theorem proving. Maybe this would take the field of automated reasoning would out of its present state of “being in the doldrums.”

Appendix I: THINKER's Proof Strategy

This Appendix gives an informal flow chart explanation of the proof strategies used by THINKER. For an explanation of the meaning of various variable names and for how certain of the underlying procedures work (such as ONESTEP and SIMPLEPROOF, including how they are augmented by identity "special cases"), see the main text. Note that whenever a formula is added to ANTELINES all the TEMPLATES are added, the RINGs are updated, the various lists of constants and free variables are updated, and the identity-equivalence classes are revised. The reverse (deletion) is done whenever a line is removed from ANTELINES (when boxing occurs). The relevant functions are ADDANTE(Φ) and DELANTE(Φ).

The PROOF procedure is called recursively by PROOF, and in addition there are various branchings (which are not recursive) within the procedure. If one of the recursive calls fails then the higher-level parent process from which it was called does not succeed (there is no backtracking that would attempt to take a different direction through the parent process), and the entire proof fails. (In the algorithm this results in a branch to step J). For example, if the formula to be proved is a biconditional, and if it cannot be proved by SIMPLEPROOF, then step (D.a) will be invoked. And assuming that the left-to-right conditional is not already a goal, the PROOF procedure will be called recursively to try to prove this conditional. If this recursive call fails, then this embedded call will branch to step J, stopping the entire proof. (If the user enters some lines to be used, then these lines will apply to the recursively embedded call). As a somewhat more intricate example, consider step (G.d). Here we have located the negation of a disjunction in ANTELINES; if the left disjunct is not already a goal then PROOF will be called recursively in an attempt to prove it, by step (G.d.i.a). In this recursive call, it is possible that we reach step (G.d) and once again find this negation of a disjunction in ANTELINES. But this time the left disjunct *is* a goal, and so step (G.d.i) will be bypassed and control handed to step (G.d.ii), which will recursively call PROOF on the right disjunct (unless it already is a goal) by step (G.d.ii.a). If this most-deeply embedded recursive call succeeds, then control is returned to the point from which it was called, (G.d.ii.a), and after a few steps more of section (G.d.ii) this embedded recursive call successfully completes. And control is returned to the calling instance from step (G.d.i.a), and in a few steps more of section (G.d.i) this recursive call succeeds.

Now for the flow of control. First the various variables are initialized: ANTELINES and GOALSTACK are empty, CURLEVEL is 0, CURLINE is 1, and the formula to be proved is read in (as are the premises, if there are any) with identities being normalized. The PROOF procedure, PROOF(ϕ,n) attempts to prove Φ, whose line number in PROOFMATRIX is n.

```
A.  Increment CURLINE and CURLEVEL by 1.
B.  Add $\Phi$ to GOALSTACK and "show $\Phi$" to PROOFMATRIX
C.  If SIMPLEPROOF($\Phi$) then goto K {SIMPLEPROOF has added $\Phi$ to PROOFMATRIX, increments CURLINE; step K
        wraps up subproof}
D.  {splitting heuristics}
    a.    If $\Phi = (\Psi \leftrightarrow \Theta)$ then
          i)    if $(\Psi \rightarrow \Theta) \notin$ GOALSTACK then
                a.  PROOF($(\Psi \rightarrow \Theta)$,CURLINE)
                b.  if ONESTEP($(\Psi \rightarrow \Theta)$), go to K
                c.  if SIMPLEPROOF($(\Theta \rightarrow \Psi)$), go to K
          ii)   if $(\Theta \rightarrow \Psi) \notin$ GOALSTACK then PROOF($(\Theta \rightarrow \Psi)$,CURLINE)
          iii)  SIMPLEPROOF($(\Psi \leftrightarrow \Theta)$); goto K.  {guaranteed to succeed, due to i) and ii)}
    b.    Else if $\Phi = (\Psi \& \Theta)$ then
          i)    if $\Psi \notin$ GOALSTACK then
                a.  PROOF($\Psi$,CURLINE)
                b.  if ONESTEP($\Psi$), go to K
                c.  if SIMPLEPROOF($\Theta$), go to K
          ii)   if $\Theta \notin$ GOALSTACK then PROOF($\Theta$,CURLINE)
          iii)  SIMPLEPROOF($(\Psi \& \Theta)$); goto K.  {guaranteed to succeed, due to i) and ii)}
    c.    Else if $\Phi = (\forall\alpha)\Psi$ and $\neg$FREE($\alpha$) and $\Psi \notin$ GOALSTACK then
          PROOF($\Psi$, CURLINE); go to K
E.  {Assumptions.  $\Phi$ is the line being proved}
    a.    if $\Phi = \neg\Psi$ and $\Psi \notin$ ANTELINES then
          i)    add "$\Psi$    ASSUME" to PROOFMATRIX; increment CURLINE by 1
          ii)   ADDANTE($\Psi$)
          iii)  set  IND := TRUE {turn on the indirect proof flag}
    b.    else if $\Phi = (\Psi \rightarrow \Theta)$ and $\Psi \notin$ ANTELINES then
          i)    if SIMPLEPROOF($\Theta$) then goto K  {the consequent was proved; wrap up subproof}
          ii)   add "$\Psi$    ASSUME" to PROOFMATRIX; increment CURLINE by 1
          iii)  ADDANTE($\Psi$)
          iv)   if ONESTEP($\Psi$) then goto K  {the consequent was proved; wrap up subproof}
        v)    if SIMPLEPROOF($\Theta$) then goto K  {the consequent was proved; wrap up subproof}
        vi)   PROOF($\Theta$,CURLINE)
        vii)  goto K  {since (vi) has provided a proof of consequent, we wrap up this subproof}
    c.    else if $\neg\Phi \notin$ ANTELINES then
        i)    add "$\neg\Phi$    ASSUME" to PROOFMATRIX; increment CURLINE by 1
        ii)   ADDANTE($\neg\Phi$)
        iii)  set  IND := TRUE {turn on the indirect proof flag}
F. {Forward "blind" inference. These are all "simplifying inferences," as described in the text. The calls to ONESTEP
   occur as each line is added to the proof, and if it succeeds we wrap up the current subproof level, dropping out of
   this FORWARD procedure.}
    a.    OLDCUR := CURLINE
    b.    if FINDQN then if ONESTEP then goto K {wrap up subproof} else goto Fb {more QNs}
    c.    if FINDDN then if ONESTEP then goto K {wrap up subproof} else goto Fc {more DNs}
    d.    if FINDBC then if ONESTEP then goto K {wrap up subproof} else goto Fd {more BCs}
    e.    if FINDS then if ONESTEP then goto K {wrap up subproof} else goto Fe {more Ss}
    f.    if FINDMP then if ONESTEP then goto K {wrap up subproof} else goto Ff {more MPs}
    g.    if FINDMT then if ONESTEP then goto K {wrap up subproof} else goto Fg {more MTs}
    h.    if FINDMTP then if ONESTEP then goto K {wrap up subproof} else goto Fh {more MTPs}
    i.    if FINDLL then if ONESTEP then goto K {wrap up subproof} else goto Fi {more LLs}
    j.    if FINDNEGID then if ONESTEP then goto K {wrap up subproof} else goto Fj {more NEGIDs}
    k.    if OLDCUR $\neq$ CURLINE then goto Fa {if steps b–j added new lines, they may participate in further
           forward inferences}
    l.    FINDALLEI {Existentially instantiate all lines that can be instantiated; mark as being EIed under
          CURLEVEL; do not EI them under the same or more embedded CURLEVEL. If an EI had an ancestor which
          was universally quantified, then mark that ancestor as a "prohibited ancestor" and the variable of
          instantiation as a "prohibited variable"}
    m.    if FINDUI then if ONESTEP then goto K {wrap up subproof} else goto Fm {more UIs}{FINDUI
          instantiates to all constants/variables active at this stage of the proof, except it does not instantiate
          "prohibited ancestors" to "prohibited variables"}
    n.    if OLDCUR $\neq$ CURLINE then goto Fa {the two ways to exit the FORWARD procedure are: if some
          ONESTEP succeeds or no more rules can be applied to any ANTELINES…no new lines have been added}
G.    {TRYNEG}  If IND then
    a.    if $\neg(\Psi\rightarrow\Theta) \in$ ANTELINES and $(\Psi\rightarrow\Theta) \notin$ GOALSTACK then
        i)    PROOF($(\Psi\rightarrow\Theta)$,CURLINE)
        ii)   ONESTEP($\Psi\rightarrow\Theta$); goto K {ONESTEP guaranteed to succeed, due to contradiction}
    b.    if $\neg(\Psi\leftrightarrow\Theta) \in$ ANTELINES and $(\Psi\leftrightarrow\Theta) \notin$ GOALSTACK then
        i)    PROOF($(\Psi\leftrightarrow\Theta)$,CURLINE)
        ii)   ONESTEP($\Psi\leftrightarrow\Theta$); goto K {ONESTEP guaranteed to succeed, due to contradiction}
    c.    if $\neg(\Psi\&\Theta) \in$ ANTELINES and $(\Psi\&\Theta) \notin$ GOALSTACK then
        i)    PROOF($(\Psi\&\Theta)$,CURLINE)
        ii)   ONESTEP($\Psi\&\Theta$); goto K {ONESTEP guaranteed to succeed, due to contradiction}
    d.    if $\neg(\Psi \vee \Theta) \in$ ANTELINES then
          i)    if $\Psi \notin$ GOALSTACK then
                a) PROOF($\Psi$,CURLINE)
                b) add " $(\Psi \vee \Theta)$  ADD"  to PROOFMATRIX
                c) add $(\Psi \vee \Theta)$ to ANTELINES; increment CURLINE by 1
                d) ONESTEP($\Psi \vee \Theta$); goto K {ONESTEP guaranteed to succeed, due to contradiction}
          ii)   else if $\Theta \notin$ GOALSTACK then
                a) PROOF($\Theta$,CURLINE)
                b) add " $(\Psi \vee \Theta)$  ADD"  to PROOFMATRIX
                c) add $(\Psi \vee \Theta)$ to ANTELINES; increment CURLINE by 1
                d) ONESTEP($\Psi \vee \Theta$); goto K {ONESTEP guaranteed to succeed, due to contradiction}
H. {chaining}
    a.    If $(\Psi \rightarrow \Theta) \in$ ANTELINES and $\Psi \notin$ ANTELINES and $\Theta \notin$ GOALSTACK then
                PROOF($\Psi$,CURLINE), goto F {to try more FORWARD inference}
    b.    If $(\Psi \vee \Theta) \in$ ANTELINES then
          i)    if $\Psi \notin$ ANTELINES and $\neg\Theta \notin$ GOALSTACK then PROOF($\neg\Theta$,CURLINE), goto F
          ii)   else if $\Theta \notin$ ANTELINES and $\neg\Psi \notin$ GOALSTACK then PROOF($\neg\Psi$,CURLINE), goto F
I.  {Prohibited universal instantiations}
    a.    OLDCUR := CURLINE
    b.    FINDUIPROHIB
    c.    if OLDCUR $\neq$ CURLINE then goto F {new lines in proof, so try FORWARD again}
                else goto J {ask for help}
J.  {Help}
    a. print out the proof as so far completed, ask for help
    b. read input
    c. if input starts with "stop" then stop
      else if input starts with "show $\Psi$" then
          PROOF($\Psi$,CURLINE) goto F {new line has been proved, so try FORWARD again}
      else
          i)    add $\Psi$  to PROOFMATRIX with "told so" as annotation
          ii)   increment CURLINE by 1
          iii)  goto F {try FORWARD inference with new line}
K.  {Wrap up subproof. Recall that n was CURLINE when the most recent call to PROVE was made}
    a.    For i:=n+1 until CURLINE do
          i)    DELANTE(PROOFMATRIX(i))  {delete from ANTELINES what's being boxed }
          ii)   add scope line to PROOFMATRIX for these lines
    b.    ADDANTE(PROOFMATRIX(n))  {add the formula just proved to ANTELINES}
    c.    add "*" to PROOFMATRIX(n)
    d.    POPGOAL {remove the most recent goal from GOALSTACK}; CURLEVEL := CURLEVEL - 1
    f.    If CURLEVEL = 0 then printout PROOFMATRIX and stop
```

Appendix II: THINKER's Proofs of Some Theorems

Some proofs in set theory can be formulated directly within first order logic by allowing the binary predicate 'Fxy' to stand for "x is a member of y." Then Russell's paradox can be put "there is no set which contains exactly those sets that are not members of themselves," that is, as

$$\neg(\exists y)(\forall x)(Fxy \leftrightarrow \neg Fxx)$$

Since "the Russell set" cannot exist, it follows that if there is a set of all things whose members *are* members of themselves ("the anti-Russell set"), then not every set can have a complement, i.e.,

$$(\exists y)(\forall x)(Fxy \leftrightarrow Fxx) \rightarrow \neg(\forall x)(\exists y)(\forall z)(Fzy \leftrightarrow \neg Fzx)$$

Zermelo-Frankel set theory replaces the unrestricted comprehension axiom (that every property determines a set) with a restricted version: given a set z, there is a set all of whose members are drawn from z and which satisfy some property. Now, if there were a universal set, then the Russell set could be formed, *per impossible*. So, given the restricted comprehension axiom, there is no universal set:

$$(\forall z)(\exists y)(\forall x)(Fxy \leftrightarrow (Fxz \& \neg Fxx)) \rightarrow \neg(\exists z)(\forall x)Fxz$$

Finally, call a set x "circular" if it is a member of a set z which in turn is a member of x. Intuitively, all the sets are non-circular, but if we could pick out the class of the non-circular sets we could thereby pick out the universal set. Hence there can be no class consisting of exactly the non-circular sets:

$$\neg(\exists y)(\forall x)(Fxy \leftrightarrow \neg(\exists z)(Fxz \& Fzx))$$

THINKER's proofs of these theorems are on the following pages.

Bibliography

Beth, F. (1958) "On Machines which Prove Theorems", in Siekmann & Wrightson pp. 79-90.

Bledsoe, W. (1971) "Splitting and Reduction Heuristics in Automatic Theorem Proving" *Artificial Intelligence* **2:** 55-78.

Bledsoe, W. (1977) "Non-Resolution Theorem Proving" *Artificial Intelligence* **9:** 1-35.

Bledsoe, W. (1983) *The UT Natural-Deduction Prover*. Technical Report, Dept. Computer Science, Univ. Texas.

Edgar, A. & F.J. Pelletier (1993) "Natural Language Generation from Natural Deduction Proofs" *Proceedings of PACLING.* (Vancouver: Simon Fraser Univ.), pp. 269-278

Kalish, D. (1974) "Review of I. Copi *Symbolic Logic*". *Journal of Symbolic Logic* : 177-178.

Kalish, D. & Montague, R. (1964) *Logic* (Hartcourt, Brace, Janovich).

Kalish, D., Montague, R., and Mar, G. (1980) *Logic* (Hartcourt, Brace, Janovich).

Newell, A., Shaw, J., Simon, H. (1957) "Empirical Explorations with the Logic Theory Machine", in Siekmann & Wrightson pp. 49-73.

Pelletier, F.J. (1982) *Completely Non-Clausal, Completely Heuristic-Driven, Automatic Theorem Proving*. Tech. Report TR82-7, Dept. Computing Science, Univ. Alberta.

Pelletier, F.J. (1986/7) "75 Graduated Problems for Testing Automated Theorem Provers" *Journal of Automated Reasoning* **3:** 191-216, and "Errata for 75 Problems" *Journal of Automated Reasoning* **4:** 235-236.

Pelletier, F.J. (1987) *Further Developments in THINKER, an Automated Theorem Prover*. Tech. Report TR-ARP-16/87 Automated Reasoning Project, Australia National University.

Pelletier, F.J. (1991) "The Philosophy of Automated Theorem Proving" *Proceedings of IJCAI-91* (Morgan Kaufmann) pp. 1039-1045.

Pelletier, F.J. (1993) "Automated Modal Logic Theorem Proving in THINKER" Technical Report TR 93-14, Dept. Computing Science, Univ. Alberta.

Pelletier, F.J. (1993) "Identity in Modal Logic Theorem Proving" *Studia Logica*.

Pelletier, F.J. & Rudnicki, P. (1986) "Non-Obviousness" Newsletter of the Association for Automated Reasoning, No. 6, pp. 4-6.

Pollock, J. (1992a) “Interest-Driven Suppositional Reasoning” *Journal of Automated Reasoning* **6:** 419-462.

Pollock, J. (1992b) “How to Reason Defeasibly” *Artificial Intelligence* **57:** 1-42.

Pollock, J. (1997) “Skolemization and Unification in Natural Deduction”, unpublished paper available at http://www.u.arizona.edu/~pollock/ .

Prawitz, D., Prawitz, H. & Voghera, N. (1960) "A Mechanical Proof Procedure and Its Realization in an Electronic Computer" in Siekmann & Wrightson pp. 202-228.

Sieg, W. & J. Byrnes (1996) “Normal Natural Deduction Proofs (in Classical Logic)” Technical Report PHIL-74, Dept. Philosophy, Carnegie Mellon Univ.

Sieg, W. & B. Kauffmann (1993) “Unification for Quantified Formulae”. Technical Report PHIL-46, Dept. Philosophy, Carnegie Mellon Univ.

Sieg, W. & R. Scheines (1992) “Searching for Proofs (in Sentential Logic)” in L. Burkholder (ed.) *Philosophy and the Computer* (Westview: Boulder), pp. 137-159.

Siekmann, J. & Wrightson, G. (1983) *The Automation of Reasoning, Vol. 1*. Springer-Verlag.

Smullyan, R. (1968) *First Order Logic*. (Springer-Verlag).

Wang, H. (1960) "Toward Mechanical Mathematics". Reprinted in Siekmann & Wrightson 1983, pp. 244-264.

Whitehead, A. & Russell, B. (1910) *Principia Mathematica, Vol I.* Cambridge Univ. Press.