Journal of Automated Reasoning **4** (1988) 191–213.

First-Order Modal Tableaux

MELVIN FITTING
Department of Mathematics and Computer Science: Herbert H. Lehman College (CUNY), Bronx, NY 10468; Department of Computer Science: Department of Philosophy: The Graduate School and University Center (CUNY), 33 West 42 Street, NY 10036, U.S.A.
Bitnet MLFLC@CUNYVM

(Received: 6 March 1987)

Abstract. We describe simple semantic tableau based theorem provers for four standard modal logics, in both propositional and first-order versions. These theorem provers are easy to implement in Prolog, have a behavior that is straightforward to understand, and provide natural places for the incorporation of heuristics.

Key words. Resolution, proof theory, modal logics, tableaux, nonclassical logics.

1. Introduction

Resolution is almost an industry standard for automated theorem proving in logic. But, it is primarily a mechanism of classical logic. When it has been applied outside that setting as in, say, [1] or [7], it has required considerable coercion, partly because normal form theorems are not generally available. As a lesser known alternative to resolution, Smullyan's formulation of semantic tableaux dates from about the same time period [15, 16] and also lends itself well to automation. It is a direct descendant of Beth's tableau system [3] which, in turn, is a descendant of the Gentzen sequent calculus [8]. Resolution methods are often thought of as being based on Herbrand's Theorem [9]. It is well-known that Gentzen's and Herbrand's ideas were closely related. But, even with this more-or-less common ancestry, tableau and resolution systems have distinctly different flavors to them. And unlike with resolution, over the years the basic first-order tableau system has developed natural extensions covering many non-classical logics [5]. Interest in non-classical theorem proving is growing and for this reason, as well as for its intrinsic interest, tableau methods deserve to be more widely known.

In this paper we describe tableau based theorem provers for four modal logics, in both propositional and first order versions. The underlying theory may be found in [5], where the tableau systems considered here are discussed in more detail. It turns out that only mild modifications to the abstractly presented systems suffice to make them easily programmable. Prolog implementations are especially perspicuous, because the backtracking mechanism of Prolog is exactly what is needed, and so the Prolog programs amount to little more than definitions of tableau proof. We begin with classical logic, for the sake of background and to make the paper more self-contained. We do assume readers are familiar with the general ideas of Kripke models [11, 12] and the basic notions of modal logic. Nothing beyong the elementary will be needed however.

2. Classical Propositional Tableaux

For those who are not familiar with it, we briefly sketch the semantic tableau system for classical propositional logic. The primary reference for this is [16]. We differ from that only by putting negations into a separate category, to avoid redundancy.

We assume propositional formulas are defined as usual, allowing as connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or) and $\supset$ (implies). Although various subsets of these constitute complete sets of connectives, there is no particular reason to restrict formulas, since Smullyan's system of 'uniform notation', presented below, allows a general treatment. The biconditional, however, does not fit well into the schema used, and is best treated as an abbreviation. Smullyan incorporates truth values directly into the syntax of tableaux. This is not necessary, and [16] also presents an alternate tableau system without this feature, but it is a device which has pedagogical advantages, and which makes parsing somewhat easier. Thus *signed* formulas are formulas prefixed with either 'T' or 'F', intuitively indicating that the following formula is true or false; for example, $F(X \supset Y)$ or $T(\neg X \supset (X \wedge Y))$. We will refer to a signed formula as *true* when we mean the unsigned part has the sign as its truth value. Signed non-atomic formulas are grouped into three categories: those acting conjunctively (called α *formulas*), those acting disjunctively (called β *formulas*), and those whose principal connective is negation (called *negative formulas*). Each *negative* has one *component*, each α has two *components*, denoted α_1 and α_2 respectively. Similarly for the β case. The categories and components are defined in the following tables.

negative	*positive*
$T\neg X$	FX
$F\neg X$	TX

α	α_1	α_2
$TX \wedge Y$	TX	TY
$FX \vee Y$	FX	FY
$FX \supset Y$	TX	FY

β	β_1	β_2
$TX \vee Y$	TX	TY
$FX \wedge Y$	FX	FY
$TX \supset Y$	FX	TY

Fig. 1.

The intuition is, under any truth-functional valuation, a *negative* is true iff the corresponding *positive* is true; an α is true iff both α_1 and α_2 are true; a β is true iff at least one of β_1 or β_2 is true.

We have followed Smullyan in making $\wedge$, $\vee$ and $\supset$ strictly binary. However this can be generalized in obvious ways. For example, here is a generalized α table. The tableau rules given below also have straightforward extensions, which we do not discuss here.

α	α_1	$\alpha_2 \ldots \alpha_n$
$TX_1 \wedge X_2 \wedge \ldots \wedge X_n$	TX_1	$TX_2 \ldots TX_n$
$FX_1 \vee X_2 \vee \ldots \vee X_n$	FX_1	$FX_2 \ldots FX_n$
$FX_1 \supset X_2 \supset \ldots \supset X_n$	TX_1	$TX_2 \ldots FX_n$

Fig. 2.

Just as in resolution theorem proving, tableau proofs are *refutation* arguments. Tableau proofs are labeled trees, with signed formulas used as labels, and which meet certain closure conditions. More specifically, a *proof* of a formula X begins with the one-branch, one-node tree labeled FX (essentially postulating that X could be false under some truth-functional assignment). Then the tree grows using the following *branch extension rules*, corresponding to the intuitions mentioned earlier.

negative	α	β
positive	α_1	$\beta_1 \mid \beta_2$
	α_2	

Fig. 3.

Thus, if a signed formula of type α occurs on a branch, the branch may be lengthened by adding α_1 and α_2 to the end. Similarly for negatives. Likewise if a signed formula of type β occurs, the branch is split and β_1 is added to one fork and β_2 to the other. Applying these branch extension rules is what corresponds to the conversion to *clause form* in resolution theorem proving, but it is possible for a tableau proof attempt to be successful before the construction has been carried out to the atomic level. In effect, full conversion to clause form is not always necessary.

A *branch* is called *closed* if it contains TP and FP for some formula P. A *tableau* is *closed* if every branch is closed. A closed tableau beginning with FX is a *proof* of X. The intuition is, the assumption that X could be false has led to an impossible situation, hence X must be true no matter what. It is shown in [16] that X is provable using tableaux iff X is a tautology.

Figure 4 shows an example of a tableau proof, for the tautology $((P \supset Q) \vee (P \supset R)) \supset (P \supset (Q \vee R))$. We have numbered signed formula occurrences in order to discuss the tableau; they are not part of the proof.

$F \quad ((P \supset Q) \vee (P \supset R))$

$\quad \supset (P \supset (Q \vee R))$ (1)

$T \quad (P \supset Q) \vee (P \supset R)$ (2)

$F \quad P \supset (Q \vee R)$ (3)

$T \quad P$ (4)

$F \quad Q \vee R$ (5)

$F \quad Q$ (6)

$F \quad R$ (7)

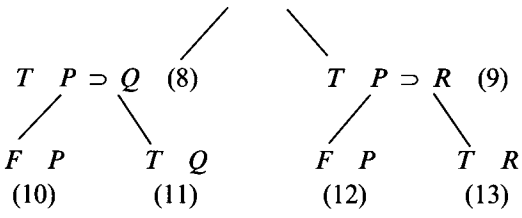

Fig. 4.

In this tableau entry (1) is of the form FX where X is the formula we wish to prove. Then entries (2) and (3) come from (1) by the α rule; (4) and (5) come from (3) by the α rule; (6) and (7) come from (5) by the α rule again. Next (8) and (9) are from (2) by the β rule; (10) and (11) are from (8) by the β rule; (12) and (13) are from (9) by the β rule. Finally, considering the branches from left to right, branch closure is by: (4) and (10); (6) and (11): (4) and (12); (7) and (13).

There are several remarks to be made, of both theoretical and practical interest.

As stated, it is allowed to apply a branch extension rule to a given formula occurrence on a branch more than once. But in fact this is never necessary. Thus, formulas can be removed from branches once they have been used, ensuring termination of any tableau construction.

The branch extension rules are inherently non-deterministic. We may choose which branch, and which signed statement on that branch to work with next. Suppose we call a deterministic ordering of branch extension rule applications for tableau construction *fair* if, eventually, each non-atomic signed formula occurrence on each branch has the appropriate rule applied to it. Any fair deterministic ordering must produce a proof of a formula that is a tautology, though some fair orderings will do so more quickly than others. It is here that heuristics begin to be applicable. In the tableau of Figure 4, for instance, we always worked with α formulas before β formulas. This is generally a good strategy.

As defined, a branch is closed if it contains a contradiction, TP and FP for some P. Any fair deterministic ordering of branch extension rule applications must

not only produce a proof of each tautology, but must, if carried out as far as possible, produce a proof in which each tableau branch contains an *atomic* level contradiction. This means that one can intermingle tests for closure with branch extension rule applications, or one can wait until no more branch extension rules apply, then test for closure. If we wait to test for closure, we may miss discovering that we had a closed tableau at a considerably earlier stage. On the other hand, frequent testing for closure is expensive. Again, this is a reasonable place for the application of heuristics. In the propositional tableau implementation in Prolog, presented in Appendix 1, we have adopted the following strategy. We test for closure before each branch extension rule application. On each branch we deal with negations first, then conjunctions, then disjunctions, and finally modalities. And within each of these classes we work with topmost formulas on branches first. The frequent testing for closure can be changed to a test delayed as long as possible by simply moving the Prolog clause that tests for contradictions to the end position in the program. Likewise the order in which branch extension rules are applied can be changed by changing the ordering of the corresponding Prolog clauses.

It follows from the remarks above that semantic tableaux provide a decision procedure for classical propositional logic. Use any fair ordering of rule applications, remove a signed formula occurrence from a branch once a branch extension rule has been applied to it, carry out branch extension rule applications until only atomic level signed formulas are left, and follow this by a test for closure.

3. Modal Propositional Tableaux

We sketch tableau systems for four standard propositional modal logics, and then consider implementation issues. A fuller discussion of these tableau systems can be found in [5]. The logics we consider, as characterized by the corresponding conditions placed on the accessibility relation of their Kripke model theory, are: K (no conditions); $K4$ (transitivity); T (reflexivity); and $S4$ (transitivity and reflexivity). For these logics the Kripke model theory does not require symmetry of the accessibility relation. This is important for the style of tableau we use here. Of course, this leaves out well-known logics like $S5$, but there are different styles of semantic tableaux suitable for such logics; we shall say more about this later.

Syntactically, the set of formulas is enlarged by adding the operators $\Box$ (necessity) and $\Diamond$ (possibility). Two new classifications of signed formulas are created, *necessaries*, or ν formulas, and *possibles*, or π formulas. These, and their single *components*, are defined as follows.

ν	ν_0
$T\Box X$	TX
$F\Diamond X$	FX

π	π_0
$T\Diamond X$	TX
$F\Box X$	FX

Fig. 5.

Again the intuition is simple. A ν formula is true at a possible world Γ iff ν_0 is true at all worlds accessible from Γ. A π formula is true at a possible world Γ iff π_0 is true at some world accessible from Γ.

Next we define an operation * on sets of signed formulas. The intuition is, if S is a set of signed formulas, and the members of S are true at a possible world Γ, then the members of S^* should be true at any world accessible from Γ. For example, suppose the logic is K, and $S = \{T\Box X, T\Diamond Y, F\Diamond Z, TP \wedge Q\}$. Let Γ be a possible world of some Kripke K model, with all members of S true at Γ, and let Δ be an arbitrary world, accessible from Γ. It is easy to see that, since $T\Box X$ is true at Γ, then TX must be true at Δ. Likewise, since $F\Diamond Z$ is true at Γ, FZ must be true at Δ. But in general we can say nothing about the behavior of Y or $P \wedge Q$ at Δ because examples of K models can easily be produced that have quite arbitrary behavior on these formulas. Thus all we can say for sure is that all the members of $\{TX, FZ\}$ must be true at Δ, and it is this set we take for S^* in this case. In general, the definition of * is logic dependent, and is given in the following chart.

Logic	S^*
K, T	$\{\nu_0 \mid \nu \in S\}$
$K4$, $S4$	$\{\nu, \nu_0 \mid \nu \in S\}$

Fig. 6.

The tableau rules, in addition to the classical ones of the previous section, are these. First, for the logics T and $S4$ involving reflexivity, the rule

$$\frac{\nu}{\nu_0}$$

Fig. 7.

The intuition here is straightforward. Suppose the set of signed formulas on a tableau branch is satisfied (simultaneously true at some possible world) and we apply this rule, say on the ν formula $T\Box X$, adding ν_0 to the branch, in this case, TX. The resulting set of formulas on the larger branch is still satisfiable, indeed at the same world, because for the logics T and $S4$, if $\Box X$ is true at a world so is X ($\Box X \supset X$ is valid in these logics).

Secondly, for all four modal logics we want the following rule, though the meaning of * differs from logic to logic. The intension is, if $S \cup \{\pi\}$ is the set of signed statements on a tableau branch, it may be *replaced* by $S^* \cup \{\pi_0\}$.

$$\frac{S \cup \{\pi\}}{S^* \cup \{\pi_0\}}$$

Fig. 8.

Again the intuition is the same as in the previous case, though the details are trickier. Suppose the set of signed formulas on a tableau branch is satisfiable, and we apply this rule. Say the set originally was $S \cup \{\pi\}$ and we have replaced it with $S^* \cup \{\pi_0\}$, and say the original set was satisfied at the possible world Γ. Since π is true at Γ, it follows that there must be some world accessible from Γ, say Δ, at which π_0 is true. Since the members of S were all true at Γ, the members of S^* will all be true at Δ, as we observed earlier. Then the entire set of formulas on the revised branch is still satisfiable, though at the world Δ instead of at Γ. Thus, like the resolution rule, the tableau rules preserve satisfiability.

Definitions of closed tableau, and proof are the same as in the classical case. It is shown in [5] that, for each of the four modal logics the corresponding tableau system is sound and complete.

Here are two examples of modal tableau proofs. The first, Figure 9, is a proof in the logic K of $\Box(X \supset Y) \supset (\Box X \supset \Box Y)$.

$F\Box(X \supset Y) \supset (\Box X \supset \Box Y)$ (1)

$T\Box(X \supset Y)$ (2)

$F\Box X \supset \Box Y$ (3)

$T\Box X$ (4)

$F\Box Y$ (5)

Fig. 9a.

$TX \supset Y$ (6)

TX (7)

FY (8)

FX (9) TY (10)

Fig. 9b.

(1) is the starting formula. (2) and (3) are from (1) by the α rule; (4) and (5) are likewise from (3) by the α rule. Now we apply the π rule (from Figure 8) using (5) as the π formula. The corresponding π_0 formula is (8), while (6) and (7) constitute S^*, where S consists of (1), (2), (3) and (4). Now a β rule application to (6) quickly produces a closed tableau.

The second example, Figure 10, is a proof in $S4$ of $\Box X \supset \Box \Diamond \Box X$.

$F\Box X \supset \Box\Diamond\Box X$ (1)

$T\Box X$ (2)

$F\Box\Diamond\Box X$ (3)

Fig. 10a.

$T\Box X$ (4)

TX (5)

$F\Diamond\Box X$ (6)

$F\Box X$ (7)

Fig. 10b.

(1) is the starting formula; (2) and (3) are from (1) by the α rule. Now, applying the π rule, taking (3) as the π formula, replaces (1), (2) and (3) by (4), (5) and (6). Finally, applying the ν rule (of Figure 7) to (6) produces (7) and a closed tableau.

Now, some remarks concerning issues of implementation. Just as in the propositional case, the order in which we apply α, β and negation rules does not matter for completeness, as long as we eventually apply them in all possible ways. So again, heuristics are pertinent for the discovery of short proofs. And, as before, we can remove a signed formula occurrence from a branch once an α, β or negation rule has been applied to it. This is not the case with the ν rule for the reflexive logics T and $S4$, however, since ν formulas also figure in the definition of the $^{\#}$ operation, and so affect the π rule. Of course for these logics the ν rule need not be used if ν_0 already occurs on the branch involved. In any event, use of the π rule should be postponed, for reasons discussed in the following paragraph.

The π rule conceals destructive branching. If more than one π formula occurs on a branch, an application of the π rule to one of them erases the others. Consequently, an implementation of any of these tableau systems must 'remember' where such a choice of π rule application has been made and, if a proof is not found, it must backtrack to the choice point and try an alternative choice. For the non-transitive logics K and T this is simple since every rule application involves a reduction in formula degree, and so every sequence of branch extension rule applications must terminate. Thus a depth-first search strategy is possible, and the tableau system with backtracking provides a straightforward decision procedure.

For the transitive logics $K4$ and $S4$ things are quite different. For example, a branch containing $T\Diamond X$ and $T\Box\Diamond X$ will allow an infinite sequence of π rule applications. Since all items on a branch must be signed subformulas of the formula we are trying to prove, and a propositional formula has only a finite set of subformulas, any (deterministic) sequence of branch extension rule applications that continues forever must become periodic. In principle, by remembering all the stages we have passed through we could include a test for periodicity, and backtrack when it has been detected. Thus a tableau based decision procedure for the two transitive logics is possible. In practice, however, such a test would be quite expensive.

A cheaper alternative to a periodicity test is to set a maximum *modal depth* on the number of successive π rule applications allowed on a branch before backtracking is forced. This amounts to doing a depth-first search to a preset depth. Of course we lose decidability, but it is still the case that any formula that is $K4$ or $S4$ valid will have a proof of some finite modal depth. Also, in the next section we turn to first-order logic

where decidability is no longer an issue anyway. There a strategy of the sort just discussed is quite reasonable. The most straightforward way to implement a modal depth cut-off is to use a counter, but there is an alternative technique we consider below.

We have discussed independent tableau systems for each of the four logics we are considering. But there is a different, and conceptually simpler strategy we can adopt. The tableau system for K is straightforward to implement. And the other three logics can be embedded into K in simple ways that are appropriate for mechanization! The embeddings involve the notion of a *positive* or *negative* occurrence of a subformula, which we assume is a familiar notion.

For the logic T: In an attempt to prove a formula X, it is negative subformulas of X of the form $\Box A$, and positive subformulas of X of the form $\Diamond A$, that will give rise to ν rule applications in a T tableau for FX. Suppose we modify a formula X before attempting to prove it, essentially by replacing each negative subformula $\Box A$ by $(\Box A) \wedge A$, and each positive subformula $\Diamond A$ by $(\Diamond A) \vee A$, thus building ν rule applications directly into the formula. A more precise definition is easily given using *signed* formulas, whose signs keep track of positiveness and negativeness for us. Thus, define a translation mapping $\mathbf{T}$ as follows. $\mathbf{T}(A) = A$ if A is *signed atomic*. $\mathbf{T}(\alpha) = \alpha'$, where α has components α_1 and α_2, $\mathbf{T}(\alpha_1) = \alpha_1'$, $\mathbf{T}(\alpha_2) = \alpha_2'$, and α' is the (unique) conjunctive signed formula whose components are α_1' and α_2'. Similarly for the β, *negative* and π cases. And finally, for the ν case, $\mathbf{T}(T\Box X) = T(\Box X' \wedge X')$ where $TX' = \mathbf{T}(TX)$; and $\mathbf{T}(F\Diamond X) = F(\Diamond X' \vee X')$ where $FX' = \mathbf{T}(FX)$. It is not hard to show that there is a closed tableau for FX in the logic T iff there is a closed tableau for $\mathbf{T}(FX)$ in the logic K. Thus an implementation of the K tableau system directly provides a proof system for T.

For the logic $K4$: Above we suggested the use of a modal depth counter, to bound a depth-first search. This idea is at the heart of our embedding of $K4$ into K. We define a map **K4** with a number parameter, essentially as we did in the T case above, but now, in $\mathbf{K4}(X, n)$ every negative subformula of the form $\Box A$ is replaced by $\Box A' \wedge \Box^2 A' \wedge \ldots \wedge \Box^n A'$, (parenthesized as convenient). Likewise, replace every positive subformula of the form $\Diamond A$ by $\Diamond A' \vee \Diamond^2 A' \vee \ldots \vee \Diamond^n A'$. It is not hard to see that there is a closed $K4$ tableau for FX iff there is a closed tableau for $\mathbf{K4}(FX, n)$ in K for some n. This can be proved using *consistency properties*, as in [5]; we omit the argument here.

For the logic $S4$: Here we combine the two translations above. Define a mapping **S4** so that negative subformulas of the form $\Box A$ are replaced by $A' \wedge \Box A' \wedge \Box^2 A' \wedge \ldots \wedge \Box^n A'$, and positive subformulas of the form $\Diamond A$ replaced by $A' \vee \Diamond A' \vee \Diamond^2 A' \vee \ldots \vee \Diamond^n A'$. Then, there is a closed $S4$ tableau for FX iff there is a closed K tableau for $\mathbf{S4}(FX, n)$ for some n.

4. Classical First-Order Tableaux

The propositional language of Section 2 is extended in the usual way. *Constant* and *function symbols* are added, the class of *terms* is defined, *relation symbols* are added,

atomic formulas are defined to be relation symbols applied to the appropriate number of terms, then *formulas* are built up using the connectives of Section 2 and the *quantifiers* $\forall$ and $\exists$. We omit details. (As formulated in [16], there were no function symbols.) When we write $\varphi(x)$ we mean that φ is a formula with (at most) x free; then $\varphi(t)$ means the result of substituting the term t for all free occurrences of x in φ. For the time being, we assume all formulas appearing in tableaux have no free variables (as was the case in [16] as well).

We introduce two new categories of signed formulas, *universals* (or γ) and *existentials* (or δ). And with each one we define a collection of *instances*, one for each term.

γ	$\gamma(t)$
$T(\forall x)\varphi(x)$	$T\varphi(t)$
$F(\exists x)\varphi(x)$	$F\varphi(t)$

δ	$\delta(t)$
$T(\exists x)\varphi(x)$	$T\varphi(t)$
$F(\forall x)\varphi(x)$	$F\varphi(t)$

Fig. 11.

For proof purposes, a new collection of constant symbols, called *parameters*, is introduced. We write L for the original first-order language, and L^* for the larger language allowing these extra constant symbols. Proofs are of formulas of L, but involve formulas of L^*.

Two additional branch extension rules are required. The intention is, if a γ formula occurs on a branch, $\gamma(t)$ may be added to the end for any closed term t, while if a δ formula occurs, $\delta(c)$ may be added, where c is a parameter that has not already been used on the branch. The rules are as follows.

$\dfrac{\gamma}{\gamma(t)}$	$\dfrac{\delta}{\delta(c)}$
any term t	new parameter c

Fig. 12.

A *proof* of X is still a closed tableau beginning with FX. It is shown in [16] that tableaux provide a complete and sound proof procedure for classical first-order logic.

Figure 13 shows an example of a tableau proof for the formula $(\exists x)(\forall y)R(x, y) \supset (\forall y)(\exists x)R(x, y)$. The language L contains, say, no function or constant symbols. We use $a, b, \ldots$ for parameters.

$F\ (\exists x)(\forall y)R(x, y) \supset (\forall y)(\exists x)R(x, y)$ (1)

$T\ (\exists x)(\forall y)R(x, y)$ (2)

$F\ (\forall y)(\exists x)R(x, y)$ (3)

$T\ (\forall y)R(a, y)$ (4)

$F\ (\exists x)R(x, b)$ (5)

$T\ R(a, b)$ (6)

$F\ R(a, b)$ (7)

Fig. 13.

In Figure 13, (2) and (3) are from (1) by the α rule. (4) is from (2) by the δ rule, where δ is $T(\exists x)(\forall y)R(x, y)$, a is a parameter, of course new at this point of the proof, and we have added $\delta(a)$, or $T(\forall y)R(a, y)$. Likewise (5) is from (3) by the δ rule. Finally (6) and (7) are from (4) and (5) by the γ rule.

As usual, the rules are non-deterministic. Call a deterministic order of branch extension rule applications *fair* if, eventually each signed *negative*, α, β or δ formmula occurrence on each branch has the appropriate rule applied to it and, for a γ formula and for each closed term t of L^*, eventually the γ rule is applied using the term t. It is shown in [16] that any fair order of branch extension rule applications must produce a proof of X if a proof of X exists, that is, if X is valid.

Except for γ formulas, when a signed formula has been used on a branch, it can be removed from the branch. But this is not the case with γ formulas. In [16] a simple example of a fair deterministic proof procedure is given: construct the tableau by working with branches from left to right, starting over at the left again after the right-most branch has been worked on. On each branch apply the appropriate rule to the topmost formula, removing it from the branch if it is not a γ formula. If it is a γ formula, use the first closed term t of L^* (in some pre-determined order) such that $\gamma(t)$ does not occur on the branch, then remove the occurrence of γ in question, and *add a new occurrence at the end of the branch.*

The main mechanical difficulty is that the γ rule allows us to use *any* closed term t, and we have no guidance as to what is a good choice. One solution is to go from γ to $\gamma(x)$, where x is a *new free variable*, and later on use unification to determine what choice of x will yield a closed branch. This leads to a further difficulty. If we don't know (until later via unification) what terms have been introduced on a branch, how do we ensure the parameters in δ rule applications will be new? One way out of this difficulty is to introduce Skolem functions before commencing a proof, thus avoiding the use of the δ rule altogether. This works quite well, but it has the disadvantage that it does not extend to the modal case, because of the general lack of a Skolem normal form result. Instead we propose an alternative that, in the classical case, essential amounts to a merging of the Skolemization construction with the tableau construction. This alternative will extend to the modal setting with no difficulties.

The γ and δ rules given above are replaced by the following where, in the γ rule, x is a free variable new to the branch and, in the δ rule f is a function symbol new to the branch, and $x_1, \ldots, x_n$ are all the free variables previously introduced on the branch by the γ rule.

$\dfrac{\gamma}{\gamma(x)}$	$\dfrac{\delta}{\delta(f(x_1, \ldots, x_n))}$
new free variable x	new function symbol f

Fig. 14.

Figure 15 presents a proof, using these new rules, of the formula $(\exists x)(P(x) \supset (\forall y)P(y))$. We leave it to the reader to supply reasons for the steps this time. We use $x_1, x_2, \ldots$ for free variables.

$$F\ (\exists x)(P(x) \supset (\forall y)P(y)) \tag{1}$$

$$F\ P(z_1) \supset (\forall y)P(y) \tag{2}$$

$$T\ P(z_1) \tag{3}$$

$$F\ (\forall y)P(y) \tag{4}$$

$$F\ P(f(z_1)) \tag{5}$$

$$F\ P(z_2) \supset (\forall y)P(y) \tag{6}$$

$$T\ P(z_2) \tag{7}$$

$$F\ (\forall y)P(y) \tag{8}$$

Fig. 15.

Now the substitution $\{z_2 \rightarrow f(z_1)\}$ will produce a contradiction involving (5) and (7). Notice that (1) was used twice.

Officially, we call a tableau *closed* if there is some substitution for the free variables involved that produces a closed tableau in the earlier sense (every branch contains a contradiction). And such a substitution can be found by a straightforward application of unification. This assumes bound variables are renamed if necessary, to prevent 'accidental capture' of variables. The simplest way, though not the most efficient, to guard against such capture is to apply these closing substitutions only at the atomic level, where no quantifiers are present. This can be done without compromising completeness. We have used this device in Appendix 2 in the interests of simplicity. We do not recommend it in general.

What we have now described can be thought of as the tableau analog of the method of matings [2]. One caveat, however. If one is implementing this in Prolog, there is a natural temptation to write the γ rule in such a way that it infers $\gamma(x)$ from γ, where x is a new *Prolog* variable, and then to use Prolog's built-in unification in the search for a closing substitution. This is simple and elegant to program. Unfortunately, it does not work because Prolog leaves out the so-called occurs check, thus implementing the unification algorithm incorrectly. And it is precisely the occurs check that ensures the revised δ rule must introduce something new. The most straightforward solution is to write a correct unification procedure of one's own, in Prolog. Fortunately, this is not too onerous. We use a version taken from [17] Chapter Ten, or equivalently, [14].

With these revised rules, it is still the case that we cannot confine ourselves to working with a γ formula occurrence on a branch only once (or else we would have decidability of first-order logic). We might adopt Smullyan's device, described above, of removing a γ occurrence once used, and introducing a fresh occurrence at the

branch end. But then an attempt to prove an unprovable formula may never terminate. Smullyan's idea can be combined with setting a maximum *quantifier usage*, specifying how many times the revised γ rule is to be applied to any given γ formula. It is surprising how often a quantifier usage of 1 turns out to work. (Restricting to quantifier usage 1 is related to, but not the same as, the notion of *direct predicate calculus* of [10].) While attaching a quantifier usage parameter to each γ formula gives rather fine control over proof search, it is simpler to implement a more global notion of *quantifier depth* instead, specifying how many times the revised γ rule is to be applied on a branch, no matter to which γ formulas. This is the approach we have taken in the program given in Appendix 2. At any rate, a provable formula will be provable using some finite quantifier usage, and also using some finite (generally higher) quantifier depth.

We have now described a first-order proof procedure whose programming presents no difficulties, and we can turn to the modal analog.

5. Modal First-Order Tableaux

Here things are extraordinarily easy. We get first-order modal proof procedures by simply combining the techniques of Section 3 and Section 4. (Of course the translations into K from Section 3 must be extended to cover the quantifier cases. This is straightforward.) The resulting logics are 'monotone' in the sense that, in the Kripke models, if world Δ is accessible from world Γ, then the domain of the first-order structure associated with Δ must be an extension of that associated with Γ. These are not 'constant domain' models. In an axiomatic formulation of modal logics, monotone logics are the ones that come up most naturally; extra axioms or rules generally must be added to ensure constant domains.

We think the point has been made now, that tableau based formulations are natural and simple to implement for those logics for which they exist, and they exist for a wider variety of logics than just classical.

6. Concluding Remarks

There are well-known embeddings of Intuitionistic logic into $S4$. These, together with the $S4$ tableau system above, yield proof procedures for this logic. Alternatively, tableau systems designed directly for Intuitionistic logic exist [5], and are straightforward to implement. [6] has a resolution style system for Intuitionistic logic that was designed by analogy with the tableau system. Similar resolution style systems for modal logics can also be developed. Other resolution style systems based on different ideas have been created; see [13] for references. It would be interesting to compare the efficiency of these various systems.

The style of modal tableaux presented here does not lend itself well to logics whose Kripke models involve symmetry, such as $S5$. This is a problem, since $S5$ is a logic which has been widely used. But there is another generalization of tableaux which

covers such logics easily; and also allows for constant domains. These tableau use *prefixed* formulas. A description of them can be found in [4] and [5]. Recently, ideas derived from prefixed tableaux have been used in an automated modal proof system [18]. A comparison of implementation efficiency for logics having both kinds of tableaux would be interesting.

Finally, the logics considered here each had single necessity operators. But it is no problem to introduce an indexed family of necessity operators, which yield simple logics of knowledge. Tableau implementation issues are unproblematic.

Appendix 1, Propositional *K4*

```
/*  Tableau based theorem prover for the propositional modal logic K4.
  Works by embedding K4 into K.  Note that one must supply a 'modal depth'
  parameter.
  Melvin Fitting
  January 11, 1987.
*/
/*  Propositional operators are: and, or, neg and imp.
  And, or are left associative;
  imp is right associative.
  Also there are signs, t and f,
  and there are modal operators, box and dia.
  Operator precedences used below assume a precedence range of
  1-255.  For an implementation of Prolog that uses a different range,
  the operators defined below should be given precedences above 'not',
  'is', etc. and below 'nospy', comma, semicolon etc.
*/
?-op(100, fy, neg).
?-op(110, yfx, and).
?-op(120, yfx, or).
?-op(130, xfy, imp).
?-op(140, fx, [t,f]).
?-op(100, fy, box).
?-op(100, fy, dia).
/*  remove(X, Y, Z) :- Z is the list resulting from removing all X
        occurrences from the list Y.
*/
remove(X, [X | Xs], Ys) :- remove(X, Xs, Ys).
remove(Z, [X | Xs], [X | Ys]) :- X \= Z, remove(Z, Xs, Ys).
remove(X, [], []).
/*  append(X,Y,Z) :- X and Y appended yields Z.
*/
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
/*  member(X, Y) :- X is a member of the list Y.
*/
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
/*  Define the propositional formula types.
*/
conjunctive(t _ and _).
conjunctive(f _ or _).
conjunctive(f _ imp _).
```

```
disjunctive(t _ or _).
disjunctive(f _ and _).
disjunctive(t _ imp _).

negative(t neg _).
negative(f neg _).

necessity(t box _).
necessity(f dia _).

possibility(t dia _).
possibility(f box _).

atomicfmla(X) :-
     not negative(X),
     not conjunctive(X),
     not disjunctive(X),
     not necessity(X),
     not possibility(X).

/*  components(F, One, Two) :-
        signed formula F has One and Two as its components.
*/

components(f X and Y, f X, f Y).
components(t X or Y, t X, t Y).
components(t X and Y, t X, t Y).
components(f X or Y, f X, f Y).
components(t X imp Y, f X, t Y).
components(f X imp Y, t X, f Y).

/*  component(F, One) :-
        signed formula F has One as its only component.
*/

component(t neg X, f X).
component(f neg X, t X).
component(t box X, t X).
component(f box X, f X).
component(t dia X, t X).
component(f dia X, f X).

/*  sharp(A, B) :- the sharp operation applied to branch A produces branch B.
*/

sharp([], []).
sharp([Head | Tail], New) :-
     not necessity(Head),
     sharp(Tail, New).
sharp([Head | Tail], [Newhead | Newtail]) :-
     necessity(Head),
     component(Head, Newhead),
     sharp(Tail, Newtail).

/*  closed(Tableau) :- Tableau can be continued to closure.
*/

closed([]).
closed([Branch|Rest]) :-
     member(t X, Branch),
     member(f X, Branch),
     closed(Rest).
closed([Branch | Rest]) :-
     member(Negation, Branch),
     negative(Negation),
     component(Negation, Positive),
     remove(Negation, Branch, Tempbranch),
```

```
     append(Tempbranch, [Positive], Newbranch),
     closed([Newbranch | Rest]).
closed([Branch | Rest]) :-
     member(Alpha, Branch),
     conjunctive(Alpha),
     components(Alpha, Alphaone, Alphatwo),
     remove(Alpha, Branch, Tempbranch),
     append(Tempbranch, [Alphaone, Alphatwo], Newbranch),
     closed([Newbranch | Rest]).
closed([Branch | Rest]) :-
     member(Beta, Branch),
     disjunctive(Beta),
     components(Beta, Betaone, Betatwo),
     remove(Beta, Branch, Tempbranch),
     append(Tempbranch, [Betaone], Newone),
     append(Tempbranch, [Betatwo], Newtwo),
     closed([Newone, Newtwo | Rest]).
closed([Branch | Rest]) :-
     member(Pi, Branch),
     possibility(Pi),
     component(Pi, Pizero),
     sharp(Branch, Temp),
     append(Temp, [Pizero], Newbranch),
     closed([Newbranch | Rest]).
/*  k4tok(A, N, B) :-
        A is a signed formula K4 formula and B is its translation into
        K of 'modal depth' N.
*/
k4trans( t A, 1, t box A ).
k4trans( f A, 1, f dia A ).
k4trans( t A, N, t box (A and B)) :-
     N>1,
     M is N-1,
     k4trans( t A, M, t B).
k4trans( f A, N, f dia (A or B)) :-
     N>1,
     M is N-1,
     k4trans( f A, M, f B).
k4tok(Negation, N, Newnegation) :-
     negative(Negation),
     component(Negation, Positive),
     k4tok(Positive, N, Newpositive),
     negative(Newnegation),
     component(Newnegation, Newpositive).
k4tok(Alpha, N, Newalpha) :-
     conjunctive(Alpha),
     components(Alpha, Alphaone, Alphatwo),
     k4tok(Alphaone, N, Newalphaone),
     k4tok(Alphatwo, N, Newalphatwo),
     conjunctive(Newalpha),
     components(Newalpha, Newalphaone, Newalphatwo).
k4tok(Beta, N, Newbeta) :-
     disjunctive(Beta),
     components(Beta, Betaone, Betatwo),
     k4tok(Betaone, N, Newbetaone),
     k4tok(Betatwo, N, Newbetatwo),
     disjunctive(Newbeta),
     components(Newbeta, Newbetaone, Newbetatwo).
```

```
k4tok(Pi, N, Newpi) :-
     possibility(Pi),
     component(Pi, Pizero),
     k4tok(Pizero, N, Newpizero),
     possibility(Newpi),
     component(Newpi, Newpizero).
k4tok(Nu, N, Newnu) :-
     necessity(Nu),
     component(Nu, Nuzero),
     k4tok(Nuzero, N, Newnuzero),
     k4trans(Newnuzero, N, Newnu).
k4tok(Atomic, N, Atomic) :- atomicfmla(Atomic).
/*  Now translate and test for theoremhood.
*/
test(X, N) :-
     k4tok(f X, N, f Y),
     closed([[f Y]]),
     write('Theorem of propositional K4 at modal depth '),
     write(N),
     write('.'),
     nl.
test(X, N) :-
     write('Not a propositional K4 theorem at modal depth '),
     write(N),
     write('.'),
     nl.
```

Appendix 2, First-Order *K*

```
/*  Tableau based theorem prover for the First Order modal logic K.
  Melvin Fitting
  October 11, 1987.
*/
/*  Operators.
  And, Or are infix, left associative,
  Imp is infix, right associative,
  Neg, box and dia are prefix.
  Quantifiers are two place functions, all and some.
  Sample formula:  all(x, box some(y, r(x,y) imp neg dia p(x))).
  To use program, issue query of the form
  test(formula, number) where formula is the formula to
  be tested, and number is the quantifier depth to be used.
  In this version, unification is done explicitly, with an occurs check.
  The unification routine is from Stirling and Shapiro's book, Chapter Ten.
  Also, a bound is placed on the number of times the Gamma rule can be used
  on a branch, the 'Qdepth'.  Further, a version of Skolemization is combined
  with the tableau construction, consequently we carry along a list of active
  free variables.  We say more about this below.
*/
?-op(100, fy, neg).
?-op(110, yfx, and).
?-op(120, yfx, or).
?-op(130, xfy, imp).
?-op(140, fx, [t,f]).
?-op(100, fy, box).
?-op(100, fy, dia).
```

```
/*  remove(X, Y, Z) :- Z is the list resulting from removing all X occurrences
  from the list Y.
*/
remove(X, [X | Xs], Ys) :- remove(X, Xs, Ys).
remove(Z, [X | Xs], [X | Ys]) :- X \= Z, remove(Z, Xs, Ys).
remove(X, [], []).
/*  append(X,Y,Z) :- X and Y appended yields Z.
*/
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
/*  member(X, Y) :- X is a member of the list Y.
*/
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
/*  Define formula types.
*/
conjunctive(t _ and _).
conjunctive(f _ or _).
conjunctive(f _ imp _).
disjunctive(t _ or _).
disjunctive(f _ and _).
disjunctive(t _ imp _).
negative(t neg _).
negative(f neg _).
universal(t all(_,_)).
universal(f some(_,_)).
existential(t some(_,_)).
existential(f all(_,_)).
necessity(t box _).
necessity(f dia _).
possibility(t dia _).
possibility(f box _).
atomicfmla(X) :-
     not conjunctive(X),
     not disjunctive(X),
     not negative(X),
     not universal(X),
     not existential(X),
     not necessity(X),
     not possibility(X).
/*  components(F, One, Two) :-
        signed formula F has One and Two as its components.
*/
components(f X and Y, f X, f Y).
components(t X or Y, t X, t Y).
components(t X and Y, t X, t Y).
components(f X or Y, f X, f Y).
components(t X imp Y, f X, t Y).
components(f X imp Y, t X, f Y).
/*  component(F, One) :-
        signed formula F has One as its only component.
*/
component(t neg X, f X).
```

```
component(f neg X, t X).
component(t box X, t X).
component(f box X, f X).
component(t dia X, t X).
component(f dia X, f X).
/*  instance(F, Term, Ins) :-
        F is a signed quantified formula, and Ins is the result of removing
       the quantifier and replacing all free occurrences of the quantified
       variable by occurrences of Term.
*/
instance(t all(X,Y), Term, t Z) :- sub(Term, X, Y, Z).
instance(f some(X,Y), Term, f Z) :- sub(Term, X, Y, Z).
instance(t some(X,Y), Term, t Z) :- sub(Term, X, Y, Z).
instance(f all(X,Y), Term, f Z) :- sub(Term, X, Y, Z).
/*  sub(Term, Variable, Formula, Newformula) :-
         Newformula is the result of substituting occurrences of Term
         for each free occurrence of Variable in Formula.
*/
sub(Term, Variable, Formula, Newformula) :-
     sub_(Term, Variable, Formula, Newformula) , !.
sub_(Term, Var, A, Term) :- Var == A.
sub_(Term, Var, A, A) :- atomic(A).
sub_(Term, Var, neg X, neg Y) :-
     sub_(Term, Var, X, Y).
sub_(Term, Var, X and Y, U and V) :-
     sub_(Term, Var, X, U),
     sub_(Term, Var, Y, V).
sub_(Term, Var, X or Y, U or V) :-
     sub_(Term, Var, X, U),
     sub_(Term, Var, Y, V).
sub_(Term, Var, X imp Y, U imp V) :-
     sub_(Term, Var, X, U),
     sub_(Term, Var, Y, V).
sub_(Term, Var, box X, box Y) :-
     sub_(Term, Var, X, Y).
sub_(Term, Var, dia X, dia Y) :-
     sub_(Term, Var, X, Y),
sub_(Term, Var, all(Var, Y), all(Var, Y)).
sub_(Term, Var, all(X, Y), all(X, Z)) :-
     sub_(Term, Var, Y, Z).
sub_(Term, Var, some(Var, Y), some(Var, Y)).
sub_(Term, Var, some(X, Y), some(X, Z)) :-
     sub_(Term, Var, Y, Z).
sub_(Term, Var, Functor, Newfunctor) :-
     Functor =.. [F | Arglist],
     sub_list(Term, Var, Arglist, Newarglist),
     Newfunctor =.. [F | Newarglist].
sub_list(Term, Var, [Head | Tail], [Newhead | Newtail]) :-
     sub_(Term, Var, Head, Newhead),
     sub_list(Term, Var, Tail, Newtail).
sub_list(Term, Var, [], []).
/*  In using the Delta rule we need a new function symbol
  each time the rule is used.  We take a Skolem function symbol
  to be fun(n) where n is a number.  The count, n, is remembered by
  funcount.  The main predicate here is sko_fun(X,Y), such
  that X is a list of free variables, and Y is a previously
  unused Skolem function applied to those free variables.
```

```
*/
funcount(1).
newfuncount(N) :-
     funcount(N),
     retract(funcount(N)),
     M is N+1,
     assert(funcount(M)).
sko_fun(Varlist, Skoterm) :-
     newfuncount(N),
     Skoterm =.. [fun, N | Varlist].
/*  Finally, we need something to restart the count of function symbols,
  when necessary.
*/
reset :-
     retract(funcount(_)),
     assert(funcount(1)),
     !.
/*  unify(Term1, Term2) :-
       Term1 and Term2 are unified with the occurs check.
       See Stirling and Shapiro, The Art of Prolog, Page 152.
*/
unify(X,Y) :-
     var(X), var(Y), X=Y.
unify(X,Y) :-
     var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.
unify(X,Y) :-
     var(Y), nonvar(X), not_occurs_in(Y,X), Y=X.
unify(X,Y) :-
     nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.
unify(X,Y) :-
     nonvar(X), nonvar(Y), compound(X), compound(Y), term_unify(X,Y).
not_occurs_in(X,Y) :-
     var(Y), X \== Y.
not_occurs_in(X,Y) :-
     nonvar(Y), atomic(Y).
not_occurs_in(X,Y) :-
     nonvar(Y), compound(Y), functor(Y,F,N), not_occurs_in(N,X,Y).
not_occurs_in(N,X,Y) :-
     N>0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N-1,
     not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y) .
term_unify(X,Y) :-
     functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
unify_args(N,X,Y) :-
     N>0, unify_arg(N,X,Y), N1 is N-1, unify_args(N1,X,Y).
unify_args(0,X,Y).
unify_arg(N,X,Y) :-
     arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
compound(X) :- functor(X,_,N), N>0.
/*  sharp(Branch, Newbranch) :-
        Newbranch is the result of applying the sharp operation to Branch.
*/
sharp([], []).
sharp( [Head | Tail], New) :-
     not necessity(Head),
     sharp(Tail, New).
```

```
sharp( [Head | Tail], [Newhead | Newtail]) :-
     necessity(Head),
     component(Head, Newhead),
     sharp(Tail, Newtail).

/*  A tableau is a list of notated branches.  A notated branch is a
  pair, [varlist, branch], where branch is the list of signed formulas
  that make up the branch in the ordinary sense, and varlist is the list
  of free variables that have been introduced during the construction
  of that branch.  We use Prolog variables for these free variables.

  closed(Tableau, Qdepth) :-
     Tableau can be continued to closure, using a quantifier depth
     of Qdepth for Gamma rule applications.
 */

closed([], _).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(t X, Branch),
     atomicfmla(X),
     member(f Y, Branch),
     unify(X, Y)closed(Rest, Qdepth),.
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(Negation, Branch),
     negative(Negation),
     component(Negation, Positive),
     remove(Negation, Branch, Tempbranch),
     append(Tempbranch, [Positive], Newbranch),
     closed([[Varlist , Newbranch] | Rest], Qdepth).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(Alpha, Branch),
     conjunctive(Alpha),
     components(Alpha, Alphaone, Alphatwo),
     remove(Alpha, Branch, Temp),
     append(Temp, [Alphaone, Alphatwo], Newbranch),
     closed([[Varlist, Newbranch] | Rest], Qdepth).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(Beta, Branch),
     disjunctive(Beta),
     components(Beta, Betaone, Betatwo),
     remove(Beta, Branch, Temp),
     append(Temp, [Betaone], Newone),
     append(Temp, [Betatwo], Newtwo),
     closed([ [Varlist, Newone], [Varlist, Newtwo] | Rest ], Qdepth).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(Delta, Branch),
     existential(Delta),
     sko_fun(Varlist, Term),
     instance(Delta, Term, Deltains),
     remove(Delta, Branch, Temp),
     append(Temp, [Deltains], Newbranch),
     closed([[Varlist, Newbranch] | Rest], Qdepth).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     Qdepth>0,
     member(Gamma, Branch),
     universal(Gamma),
     remove(Gamma, Branch, Temp),
     Newvarlist = [V | Varlist],
     instance(Gamma, V, Gammains),
     append(Temp, [Gammains, Gamma], Newbranch),
```

```
     N is Qdepth-1,
     closed([[Newvarlist, Newbranch] | Rest], N).
closed([[Varlist, Branch] | Rest], Qdepth) :-
     member(Pi, Branch),
     possibility(Pi),
     component(Pi, Pizero),
     sharp(Branch, Temp),
     append(Temp, [Pizero], Newbranch),
     closed([[Varlist, Newbranch] | Rest], Qdepth).
/*  Finally, the driver.
*/
test(X, Qdepth) :-
     reset,
     closed([ [[],[f X]] ], Qdepth),
     write('First Order K theorem at Qdepth '),
     write(Qdepth),
     write('.'),
     nl.
test(X, Qdepth) :-
     write('Not a First Order K theorem at Qdepth '),
     write(Qdepth),
     write('.'),
     nl.
```

Acknowledgements

I would like to thank one of the referees for making suggestions which improved the programs in this paper. Research supported by NSF Grant CCR-8702307 and PSC-CUNY Grants 666396 and 667295.

References

1. Abadi, M. and Manna, Z., 'Modal theorem proving', in *8th International Conference on Automated Deduction*, edited by J. H. Siekmann, pp. 172–179, *Lecture Notes in Computer Science*, vol. 230, Springer-Verlag (1986).
2. Andrews, P., 'Theorem Proving via General Matings', *Journal of the Assoc. Comp. Mach.* **28**, 193–214 (1981).
3. Beth, E. W., *The Foundations of Mathematics*, revised edition, North-Holland, Amsterdam (1964); first edition (1959).
4. Fitting, M. C., 'Tableau methods of proof for modal logics', *Notre Dame Journal of Formal Logic* **13**, 237–247 (1972).
5. Fitting, M. C., *Proof Methods for Modal and Intuitionistic Logics*, D. Reidel Publishing Co., Dordrecht (1983).
6. Fitting, M. C., 'Resolution for intuitionistic logic', *Methodologies for Intelligent Systems*, edited by Zbigniew Ras and Maria Zemankova, pp. 400–407, North-Holland, Amsterdam (1987).
7. Geissler, C. and Konolige, K., 'A resolution method for quantified modal logics of knowledge and belief', in *Reasoning About Knowledge, Proceedings of the 1986 Conference*, pp. 309–324, edited by J. Halpern, Morgan Kaufmann Publishers (1986).
8. Gentzen, G., 'Untersuchungen über das logische Schliessen', *Mathematische Zeitschrift*, **39**, 176–210, 405–431; translated in *The Collected Papers of Gerhard Gentzen*, edited by M. E. Szabo, North-Holland, Amsterdam (1969).
9. Herbrand, J., 'Investigations in Proof Theory', English trans. in *Jacques Herbrand Logical Writings*, edited by Warren D. Goldfarb, Harvard University Press, Cambridge (1971).

10. Ketonen, J. and Weyhrauch, R., 'A decidable fragment of predicate calculus', *Theoretical Computer Science* **32**, 297–307 (1984).
11. Kripke, S., 'Semantical analysis of modal logic I, normal propositional calculi', *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* **9**, 67–96 (1963).
12. Kripke, S., 'Semantical considerations on modal logics', *Acta Philosophica Fennica, Modal and Many-valued Logics*, pp. 83–94 (1963).
13. Review, 'Some remarks on the possibility of extending resolution proof procedures to intuitionistic logic', paper by M. Cialdea, reviewer G. E. Mints, *Math. Reviews*, review 87j:03016, p. 5407 (1987).
14. O'Keefe, R., *Programming Meta-Logical Operations in Prolog*, DAI Working Paper No. 142, Department of Artificial Intelligence, University of Edinburgh (1983).
15. Robinson, J. A., 'A machine-oriented logic based on the resolution principle', *J. Assoc. Comput. Mach.* **12**, 23–41 (1965).
16. Smullyan, R. M., *First-Order Logic*, Springer-Verlag, Berlin (1968).
17. Stirling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge (1986).
18. Wallen, L. A., Matrix proof methods for modal logics (unpublished).