leanT^AP Revisited

Melvin Fitting
mlflc@cunyvm.cuny.edu
Dept. Mathematics and Computer Science
Lehman College (CUNY), Bronx, NY 10468
Depts. Computer Science, Philosophy, Mathematics
Graduate Center (CUNY), 33 West 42nd Street, NYC, NY 10036

March 13, 1997

Abstract

A sequent calculus of a new sort is extracted from the Prolog program leanT^AP. This calculus is sound and complete, even though it lacks almost all structural rules. Thinking of leanT^AP as a sequent calculus provides a new perspective on it and, in some ways, makes it easier to understand. It is also easier to verify correctness and completeness of the Prolog implementation. In addition, it suggests extensions to other logics, some of which are considered here.

1 Introduction

In recent papers [1, 2], Beckert and Posegga present a remarkable and elegant first-order theorem prover, which they call leanT^AP. It is small, consisting in its entirety of five Prolog clauses. It makes essential use of Prolog's search mechanism—if the algorithm were to be written in a different language, one would need to emulate at least a part of Prolog. It is because of the close fit with Prolog that the program can be as small as it is, and it is as efficient as the Prolog in which it is implemented allows. Finally, because of the small size of the program, Beckert and Posegga are able to sketch a proof of soundness and completeness for it.

In trying to understand leanT^AP better, and to simplify the proof of its completeness, I tried formulating the underlying algorithm in a more abstract fashion. Much to my surprise, I found that it could readily be thought of as a sequent calculus, of a kind I had not seen before. Once formulated this way, natural modifications in the presentation suggested themselves. The result is a sequent calculus for classical logic that is of theoretical interest in its own right. In particular, almost all structural rules are missing, though it is still a system for classical logic and not for one of the substructural logics. Moreover, it is a sequent calculus whose implementation in Prolog is relatively obvious, and so soundness and completeness of the resulting implementation essentially reduces to soundness and completeness of the sequent calculus itself, and this can be proved at a more abstract level. Finally, once seen this way, extensions to other logics suggest themselves—we give a few modal versions here.

I want to emphasize that the purpose of this paper is not to present an improved version of leanT^AP. That hardly seems possible. Rather the purpose is to gain insights into the operation of leanT^AP by taking a different, more abstract, way of looking at it. That versions for modal logics were then formulated so readily suggests that such an approach can be fruitful. Nonetheless, this is perhaps the first time that a sequent calculus originated as a computer program, rather than the other way around.

2 Propositional leanT^AP

The Beckert and Posegga program is a theorem prover for first-order logic. All the essential points we wish to consider can already be addressed at the propositional level and so, in the interests of simplicity, we ignore issues of quantification. In particular, this allows us to avoid Skolemization efficiency concerns, which are more subtle for modal logics than for classical logic. From a programming point of view, confining things to the propositional case leads us to modify leanT^AP by dropping two arguments from the main predicate, simplifying one clause, and deleting another. The resulting program is as follows.

```
prove((A,B),UnExp,Lits) :- !,
   prove(A,[B|UnExp],Lits).
prove((A;B),UnExp,Lits) :- !,
   prove(A,UnExp,Lits),
   prove(B,UnExp,Lits).
prove(Lit,_,[L|Lits]) :-
   (Lit = -L,!) ; (-Lit = L,!) ; prove(Lit,[],Lits).
prove(Lit,[Next|UnExp],Lits) :-
   prove(Next,UnExp,[Lit|Lits]).
```

Beckert and Posegga assume formulas have been preprocessed into negation normal form. They use "−" for negation, "," for conjunction, and ";" for disjunction. The theorem-prover essentially runs through the branches of a classical tableau from left to right (as conventionally presented), except that the left-right spatial ordering is replaced by a temporal one invoking Prolog's search mechanism. The three arguments of `prove(Fml,UnExp,Lits)` (Beckert and Posegga actually had five arguments, because they considered the first-order case) have the following operational roles. `Fml` is the formula currently being expanded (on a given branch), `UnExp` is the list of formulas on the branch that have not yet been considered, and `Lits` is the list of literals thus far discovered to be on the branch. The first clause says: when considering a conjunction, expand the first component, and add the second to the list of formulas to be considered later—that is, if a conjunction is present on a branch, add expanded versions of both components, and do so by considering the first component, while reserving the second for later consideration. The second clause says a dual thing about disjunction—there is a split into two cases. The next two clauses have to do with branch closure. (Note that according to Prolog's way of considering clauses, we cannot get to either of the last two clauses unless `Fml` is a literal.) Clause three says that if the formula under consideration directly contradicts one of the literals we have already seen, the branch is closed (note that there is a recursion through the literal list). Otherwise, clause four says we should add the literal under current consideration to the list of literals we have seen, and then go on.

To prove a propositional formula X using leanT^AP, begin by negating it, then convert the result to negation normal form, using the notation mentioned above; call the result `N`. All this is pre-processing. Finally, issue the Prolog query:

```
prove(N, [], []).
```

A few observations, before we turn to a more abstract formulation. First, the Beckert and Posegga program keeps track of the literals encountered thus far (on each branch), and checks for a contradiction between each new literal and this list. It is at this point, during the contradiction check, that atomic formulas are negated. This is mildly wasteful since the literal being checked must be negated again and again as the list is traversed. It would be simpler to remember, not the list of literals encountered, but their *duals*, thus negating things once and for all, after which checks for closure just involve simple equality testing. Thus we replace the last two clauses with the following three.

```
prove(Lit,_,[L|Lits]) :-
   (Lit = L,!) ; prove(Lit,[],Lits).
prove(-Lit,[Next|UnExp],Lits) :- !,
   prove(Next,UnExp,[Lit|Lits]).
prove(Lit,[Next|UnExp],Lits) :- !,
   prove(Next,UnExp,[-Lit|Lits]).
```

Second, Beckert and Posegga assume formulas have been pre-processed into negation normal form. To call attention to similarities with standard sequent calculi, we incorporate this reduction into the algorithm itself, rather than separating it out. It imposes little additional complexity, and also, when dealing with non-classical logics, such pre-processing may not be possible anyway. We discuss in detail how such a modification affects the first clause, and sketch the rest.

We use the well-known α/β classification of Smullyan, [4, 3], in which α's are conjunctive and β's are disjunctive. For each formula type, *components* are defined. For instance, $(X \wedge Y)$ is an α, and its components are $\alpha_1 = X$ and $\alpha_2 = Y$. Likewise $\neg(X \vee Y)$ is also an α, with components $\alpha_1 = \neg X$ and $\alpha_2 = \neg Y$. A table is given in the next section.

Now suppose we add to the program clauses like the following: `type(X and Y, conj, X, Y)`, and `type(neg(X or Y), conj, neg X, neg Y)`, where `and` and `or` have been defined to be infix, and `neg` prefix. Then the first clause of the program is replaced with the following:

```
prove(Alpha,UnExp,Lits) :-
   type(Alpha,conj,Alpha1,Alpha2), !,
   prove(Alpha1,[Alpha2|UnExp],Lits).
```

and similarly for the second clause, which becomes the β case. We assume that doubly negated formulas are a separate category, and so an additional clause must be added to deal with them. (Smullyan considered them to be either α or β as was convenient. While this is theoretically acceptable, it adds to the overhead of an automated theorem-prover by duplicating formulas, which is why we do not take this route.)

Finally, the distinction between the first and the second arguments of `prove` has no logical role. The first argument is the formula we are currently considering, and the second is the list of remaining formulas. It would be simpler, conceptually, to combine both into a single list, and assume we always expand the head item. Thus we replace `prove(Fml,UnExp,Lits)` with a two-argument version `prove([Fml | UnExp], Lits)`. We make this modification throughout the rest of the paper, essentially to make it easier to understand the program and its variants. If it makes implementations faster one can, of course, go back to the original version at the end.

3 The sequent calculus indseq

Let us assume the various modifications to leanT^AP mentioned in the previous section have been made. Now we reformulate the algorithm behind the method more abstractly. We begin with the clauses for `type`, which are embodied in the following standard tables. (They are slightly non-standard in that they include cases for $\equiv$, essentially thinking of it as a defined connective.)

α	α_1	α_2
$(X \wedge Y)$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \supset Y)$	X	$\neg Y$

β	β_1	β_2
$(X \vee Y)$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$(X \supset Y)$	$\neg X$	Y
$(X \equiv Y)$	$X \wedge Y$	$\neg X \wedge \neg Y$
$\neg(X \equiv Y)$	$X \wedge \neg Y$	$\neg X \wedge Y$

Now we give the sequent calculus itself. Often, with classical logic, structural rules are suppressed and the left and the right of the arrow are taken as holding *sets* of formulas. This was not Gentzen's approach, and it is inappropriate for systems like linear logic where the role of the structural rules is critical. In our sequent calculus, as in Gentzen's, *sequences* of formulas are to the left and right of the arrow, not sets, but even so, there are no structural rules except for a very restricted version of thinning. Furthermore, to the right of the arrow appear only *literals*. The idea, quite simply, is that the sequent $\Gamma \rightarrow \Lambda$ should correspond to `prove(Γ, Λ)` in the final version of the program given in the previous section. Note: we use Γ to stand for an arbitrary sequence of formulas, and Λ to stand for an arbitrary sequence of literals. Either may be empty. Also, for a literal L, we use $\overline{L}$ for the complementary literal. Now, here is the system indseq.

Axioms For a literal L, $L, \Gamma \rightarrow L, \Lambda$

Rules
Double Negation Rule

$$\frac{X, \Gamma \rightarrow \Lambda}{\neg\neg X, \Gamma \rightarrow \Lambda}$$

α-Rule

$$\frac{\alpha_1, \alpha_2, \Gamma \rightarrow \Lambda}{\alpha, \Gamma \rightarrow \Lambda}$$

β-Rule

$$\frac{\beta_1, \Gamma \rightarrow \Lambda \quad \beta_2, \Gamma \rightarrow \Lambda}{\beta, \Gamma \rightarrow \Lambda}$$

Thinning For a literal L_2,

$$\frac{L_1 \rightarrow \Lambda}{L_1, \Gamma \rightarrow L_2, \Lambda}$$

Duality For a literal L,

$$\frac{\Gamma \rightarrow \overline{L}, \Lambda}{L, \Gamma \rightarrow \Lambda}$$

To prove a formula X in this system, prove the sequent $\neg X \rightarrow$, that is, refute the formula $\neg X$. This system is a sound and complete sequent formulation of classical propositional logic, though that fact may not be obvious at first glance, because of the curious mix of sequences and lack of structural rules. We show soundness and completeness for a dual system below. All we need at the moment, however, is that the relationship between this sequent calculus and the propositional leanT^AP program should be apparent. We leave you to convince yourselves of that.

4 The sequent calculus dirseq

The propositional sequent calculus of the previous section, indseq, is a little backward in its formulation, essentially because it arose from a Prolog program that was modelled on a tableau system. But it is easy to turn it into a direct version, one in which to prove X one proves $\rightarrow X$, instead of one in which one proves $\neg X \rightarrow$. To do this, we simply dualize all the rules, getting the following system, dirseq, in which only literals can appear to the *left* of the arrow.

Axioms For a literal L, $\quad L, \Lambda \rightarrow L, \Gamma$

Rules

Double Negation Rule

$$\frac{\Lambda \rightarrow X, \Gamma}{\Lambda \rightarrow \neg\neg X, \Gamma}$$

β-Rule

$$\frac{\Lambda \rightarrow \beta_1, \beta_2, \Gamma}{\Lambda \rightarrow \beta, \Gamma}$$

α-Rule

$$\frac{\Lambda \rightarrow \alpha_1, \Gamma \quad \Lambda \rightarrow \alpha_2, \Gamma}{\Lambda \rightarrow \alpha, \Gamma}$$

Thinning For a literal L_1,

$$\frac{\Lambda \rightarrow L_2}{L_1, \Lambda \rightarrow L_2, \Gamma}$$

Duality For a literal L,

$$\frac{\overline{L}, \Lambda \rightarrow \Gamma}{\Lambda \rightarrow L, \Gamma}$$

As remarked above, to prove X in this system, we prove the sequent $\rightarrow X$. Note that the Thinning rule in this system is different than in the indseq system of the previous section, though the same name has been used for both. (I want to thank Rajeev Goré for pointing out the inefficiencies of an earlier formulation of the Thinning rule.)

Now we sketch proofs of the soundness and completeness of this system. The proofs are direct, and do not reduce the issue to related facts about tableaus. It is simpler this way, though the proofs, in fact, are adapted from similar proofs about tableau systems.

Call a sequent $\Lambda \rightarrow \Gamma$ *true* under a Boolean valuation v if some member of Λ is not true under v or some member of Γ is true under v. (This is the usual notion of truth for sequents.) Call a sequent *valid* if it is true under every Boolean valuation.

Every axiom is valid, and the rules of inference preserve validity. Consequently every provable sequent is valid. In particular, if $\to X$ is provable, it is valid, and hence the formula X is also valid. This shows soundness of dirseq.

Completeness, as always, is more work. First, a few observations. Let us call a *generalized axiom* any sequent of the form

$$\Lambda_1, L, \Lambda_2 \to L, \Gamma$$

If Λ_1 is empty this is, in fact, an axiom of dirseq. And if Λ_1 is not empty, this sequent is easily seen to be provable from an axiom using repeated applications of the Thinning rule. Since all this is straightforward, we will simply allow the use of generalized axioms for the time being, and ignore the role of Thinning. Thus for the rest of this section Thinning is, in effect, not considered to be a rule.

The second observation is equally simple. Since a formula must be exactly one of: an α, a β, a double negation, or a literal, it follows that a sequent $\Lambda \to X, \Gamma$ can be the conclusion of exactly one rule—which one depends on the formula X. (Recall, we are temporarily disallowing Thinning.)

Now, by a *proof tree* for a sequent $\Lambda \to \Gamma$ we mean a tree with nodes labeled by sequents, meeting the following conditions. The sequent $\Lambda \to \Gamma$ is at the root. For each node of the tree, with sequent N as its label, one of the following: If N is a generalized axiom, the node is a leaf. If the right-hand side of N is empty, the node is a leaf. If the right-hand side of N is not empty, and P/N is a rule of dirseq, the node has one child, with P as label. If the right-hand side of N is not empty, and $(P\ Q)/N$ is a rule, the node has two children, with P and Q as labels. Note that a sequent calculus *proof* is just a proof tree in which each leaf is labeled with a generalized axiom.

Using the observations we made above, there is exactly one proof tree for each sequent. As an example, here is a proof tree for the sequent $\to (\neg P \wedge Q) \vee P$, written upsidedown, with children above parents and the root at the bottoms.

$$\dfrac{\dfrac{\dfrac{P \to P}{\to \neg P, P} \qquad \dfrac{\dfrac{\neg P, \neg Q \to}{\neg Q \to P}}{\to Q, P}}{\to \neg P \wedge Q, P}}{\to (\neg P \wedge Q) \vee P}$$

A glance at the form of the rules shows that the child or children of a node must be labeled with sequents whose right-hand sides contain one fewer formula than the sequent labeling the parent, or the same number, with all formulas but the first the same as the parent, but with the first being a simpler formula (for this purpose it suffices to count the *degree* of $X \equiv Y$ as that of $X+$ that of $Y + 4$). It follows that a proof tree for a sequent must be finite.

Suppose a node is labeled with a sequent of the form $\Lambda \rightarrow L, \Gamma$, where Λ is a consistent sequence of literals and L is a literal. If this is not a generalized axiom, the node has a child that is labeled with $\overline{L}, \Lambda \rightarrow \Gamma$. Obviously $\overline{L}, \Lambda$ will again be consistent, or we would have had a generalized axiom before. It follows that if we construct a proof tree for the sequent $\Lambda \rightarrow \Gamma$, where Λ is consistent, every node in the tree will be labeled with a sequent that has a consistent left-hand side.

Now let X be a propositional formula, and suppose we construct a proof tree for $\rightarrow X$ (a sequent which trivially has a consistent left-hand side). Either the result is a proof (every leaf is a generalized axiom), and hence by the soundness result above X is a tautology, or else at least one leaf is not a generalized axiom. Such a leaf must be labeled with $\Lambda \rightarrow$, where Λ is consistent. Then some boolean valuation v maps every member of Λ to true, and consequently falsifies the sequent $\Lambda \rightarrow$. We leave it to you to check that v also falsifies the label of every ancestor of this node, hence falsifies $\rightarrow X$, hence falsifies X.

Stating the contrapositive, if X is a tautology, a proof tree for X will, in fact, be a proof of X in dirseq, and so the sequent calculus is complete.

5 Back To an Implementation

The sequent calculus dirseq can easily be turned back into a Prolog program if we reverse the steps by which it arose. The result, of course, is more complex than leanT^AP, since we have included parsing information. But the essential style is the same, except that it is now, in a sense, a direct, rather than an indirect theorem prover. We give it in full, not because it is non-obvious, but because we can then proceed to discuss modifications to add modalities.

```
/*
  First, the various operators.
*/

:-op(100, fy, neg).
:-op(110, yfx, and).
:-op(120, yfx, or).
:-op(130, xfy, imp).
:-op(130, xfy, iff).


/*
  Next, a classification of formula types,
  and instances.
*/
```

```
type(X and Y, conj, X, Y).
type(neg(X and Y), disj, neg X, neg Y).
type(X or Y, disj, X, Y).
type(neg(X or Y), conj, neg X, neg Y).
type(X imp Y, disj, neg X, Y).
type(neg(X imp Y), conj, X, neg Y).
type(X iff Y, disj, X and Y, neg X and neg Y).
type(neg(X iff Y), disj, X and neg Y, neg X and Y).
type(neg (neg (X)), doub, X, _).

/*
  Now the heart of the matter.
  thm(Lambda, Gamma) :-
  the sequent Lambda --> Gamma is provable.
*/

thm(Lambda, [Doubleneg | Gamma]) :-
  type(Doubleneg, doub, X, _), !,
  thm(Lambda, [X | Gamma]).

thm(Lambda, [Beta | Gamma]) :-
  type(Beta, disj, Beta1, Beta2), !,
  thm(Lambda, [Beta1, Beta2 | Gamma]).

thm(Lambda, [Alpha | Gamma]) :-
  type(Alpha, conj, Alpha1, Alpha2), !,
  thm(Lambda, [Alpha1 | Gamma]), !,
  thm(Lambda, [Alpha2 | Gamma]).

thm([L1|Lambda], [L2|_]) :-
  (L1 = L2, ! ; thm(Lambda, [L2])).

thm(Lambda, [neg L | Gamma]) :-
  thm([L | Lambda], Gamma), !.

thm(Lambda, [L | Gamma]) :-
  thm([neg L | Lambda], Gamma), !.

/*
  Finally, the driver.
*/

tautology(X) :-
  thm([], [X]).
```

In the definition of a proof tree in the previous section, it was important that a node labeled with a generalized axiom be a leaf—we did not want to apply Duality to it. This is reflected in the program above, in the fact that the clause testing for closure (essentially a test for being a generalized axiom)

```
thm([L1|Lambda], [L2|_]) :-
  (L1 = L2, ! ; thm(Lambda, [L2])).
```

appears before the two clauses implementing Duality. Recall, Prolog tries clauses in the order they appear in the program.

6 A Modal Calculus for K

We wish to formulate a version of dirseq for a few propositional modal logics, and we begin with a version for the simplest normal modal logic, K. We call the resulting sequent calculus dirseqK. We add $\Box$ and $\Diamond$ to the language, in the usual way, and extend uniform notation to modal formulas, as follows.

ν	ν_0
$\Box X$	X
$\neg\Diamond X$	$\neg X$

π	π_0
$\Diamond X$	X
$\neg\Box X$	$\neg X$

These new types of formulas are not analyzed using the machinery of Boolean valuations—they require the possible world structures of a Kripke model. Consequently we enhance the sequent calculus machinery of dirseq, specifically separating off the two new categories of formulas. A *modal sequent* is an expression of the form:

$$\Lambda \rightarrow \Gamma \| \Delta \| \Omega$$

where: Λ, Γ, Δ, and Ω are finite sequences of formulas, any of which may be empty, and with Λ consisting only of literals. The intended interpretation of such a modal sequent is as follows: We say it is true at a possible world of a Kripke model provided: either some member of Λ is false at that world, or some member of Γ is true, or some member of Δ is necessary, or some member of Ω is possible. (To say a formula X is necessary at a world is to say $\Box X$ is true there; likewise X is possible at a world if $\Diamond X$ is true there.)

Sequent calculus rules are obvious modifications of the classical rules, insofar as modal operators are not specifically involved—the necessary and the possible formula lists are simply carried along unchanged. The four modal rules have a justification which we give directly after we present the full system for dirseqK.

Axioms For a literal L, $\quad L, \Lambda \rightarrow L, \Gamma \| \Delta \| \Omega$

Rules

Double Negation Rule

$$\frac{\Lambda \to X, \Gamma \| \Delta \| \Omega}{\Lambda \to \neg\neg X, \Gamma \| \Delta \| \Omega}$$

β-Rule

$$\frac{\Lambda \to \beta_1, \beta_2, \Gamma \| \Delta \| \Omega}{\Lambda \to \beta, \Gamma \| \Delta \| \Omega}$$

α-Rule

$$\frac{\Lambda \to \alpha_1, \Gamma \| \Delta \| \Omega \quad \Lambda \to \alpha_2, \Gamma \| \Delta \| \Omega}{\Lambda \to \alpha, \Gamma \| \Delta \| \Omega}$$

ν-Rule

$$\frac{\Lambda \to \Gamma \| \nu_0, \Delta \| \Omega}{\Lambda \to \nu, \Gamma \| \Delta \| \Omega}$$

π-Rule

$$\frac{\Lambda \to \Gamma \| \Delta \| \pi_0, \Omega}{\Lambda \to \pi, \Gamma \| \Delta \| \Omega}$$

Thinning For a literal L_1,

$$\frac{\Lambda \to L_2 \| \Delta \| \Omega}{L_1, \Lambda \to L_2, \Gamma \| \Delta \| \Omega}$$

Duality For a literal L,

$$\frac{\overline{L}, \Lambda \to \Gamma \| \Delta \| \Omega}{\Lambda \to L, \Gamma \| \Delta \| \Omega}$$

Necessitation

$$\frac{\to X, \Omega \| \ \|}{\Lambda \to \ \| X, \Delta \| \Omega}$$

$\Box$-Thinning

$$\frac{\Lambda \to \ \| \Delta \| \Omega}{\Lambda \to \ \| X, \Delta \| \Omega}$$

To prove a formula X in this system, prove the sequent $\to X\| \ \|$.

Here are justifications for the modal rules. The ν-Rule is straightforward: if ν_0 is in the list of necessary formulas, we are saying ν_0 is necessary, and this is equivalent to saying ν is true. Similarly for the π rule. The $\Box$-Thinning Rule is conceptually trivial. Finally, the Necessitation Rule is essentially a standard result in disguised form. To see this suppose, for simplicity, that Λ is the sequence A, B, Δ is the sequence C, D, and Ω is the sequence E, F, and modal sequents are interpreted as we said earlier. Then what we must show is: If $(X \vee E \vee F)$ is true at all worlds of a Kripke model, so is $(A \wedge B) \supset (\Box X \vee \Box C \vee \Box D \vee \Diamond E \vee \Diamond F)$. This is a consequence of the following argument, in which each step is an easily justified modal or classical inference.

$$\begin{gathered}
X \vee E \vee F \\
\neg(E \vee F) \supset X \\
\Box\neg(E \vee F) \supset \Box X \\
\Box X \vee \neg\Box\neg(E \vee F) \\
\Box X \vee \Diamond(E \vee F) \\
\Box X \vee \Diamond E \vee \Diamond F \\
(A \wedge B) \supset (\Box X \vee \Box C \vee \Box D \vee \Diamond E \vee \Diamond F)
\end{gathered}$$

Given these observations, a soundness proof is straightforward. We simply verify that all axioms are true under the intended interpretation, at all worlds in all Kripke models, and that rule applications preserve this property. We leave this to you.

Here is an example of a proof in this system, of the familiar formula $\Box(X \supset Y) \supset (\Box X \supset \Box Y)$. By soundness, the formula must be K valid (which you probably knew anyway).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{X, \neg Y \to X\| \ \| \qquad \dfrac{\neg Y \to \neg Y\| \ \|}{X, \neg Y \to \neg Y\| \ \|}\,\text{Thinning}}{X, \neg Y \to \neg(X \supset Y)\| \ \|}\,\alpha}{\neg Y \to \neg X, \neg(X \supset Y)\| \ \|}\,\text{Duality}}{\to Y, \neg X, \neg(X \supset Y)\| \ \|}\,\text{Duality}}{\to \|Y\| \neg X, \neg(X \supset Y)}\,\text{Necessitation}}{\to \Box Y\| \ \|\neg X, \neg(X \supset Y)}\,\nu}{\to \neg\Box X, \Box Y\| \ \|\neg(X \supset Y)}\,\pi}{\to \Box X \supset \Box Y\| \ \|\neg(X \supset Y)}\,\beta}{\to \neg\Box(X \supset Y), \Box X \supset \Box Y\| \ \|}\,\pi}{\to \Box(X \supset Y) \supset (\Box X \supset \Box Y)\| \ \|}\,\beta$$

Completeness is a more serious affair. We give a very brief sketch of the argument, which unsurprisingly is along standard tableau lines.

This time, by a *generalized axiom* we mean a sequent of the form

$$\Lambda_1, L, \Lambda_2 \rightarrow L, \Gamma \| \Delta \| \Omega$$

As before, we allow generalized axioms, and remove Thinning from the system.

Next, by a *block* we mean a portion of a proof attempt that does not use either the Necessitation or the $\Box$-Thinning Rules (or the Thinning Rule). The idea is to divide an unsuccessful proof attempt into blocks, with blocks connected by Necessitation and $\Box$-Thinning Rule applications. Think of each block as a possible world, and construct a Kripke model from what results. Rather than give the argument in detail, we give an example that illustrates the general ideas.

Suppose we try to prove the formula $(P \wedge \Box(P \vee Q)) \supset (\Box P \vee \Box Q)$ in dirseqK. The proof search begins with the following block.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P \rightarrow \|Q, P\| \neg(P \vee Q)}{P \rightarrow \Box Q \| P \| \neg(P \vee Q)}\;\nu}{P \rightarrow \Box P, \Box Q \| \; \| \neg(P \vee Q)}\;\nu}{P \rightarrow \Box P \vee \Box Q \| \; \| \neg(P \vee Q)}\;\beta}{P \rightarrow \neg\Box(P \vee Q), \Box P \vee \Box Q \| \; \|}\;\pi}{\rightarrow \neg P, \neg\Box(P \vee Q), \Box P \vee \Box Q \| \; \|}\;\text{Duality}}{\rightarrow \neg(P \wedge (\Box(P \vee Q)), \Box P \vee \Box Q \| \; \|}\;\beta}{\rightarrow (P \wedge \Box(P \vee Q)) \supset (\Box P \vee \Box Q) \| \; \|}\;\beta$$

Let us call the display above *block 1.*

The initial sequent of block 1 is not an axiom. It is, however, the conclusion of two different rules—Necessitation and $\Box$-Thinning. If we use Necessitation, we proceed upward via the following step:

$$\dfrac{\rightarrow Q, \neg(P \vee Q) \| \; \|}{P \rightarrow \|Q, P\| \neg(P \vee Q)}\;\text{Necessitation}$$

Now we can continue upwards with the following block:

$$\dfrac{\dfrac{\dfrac{\dfrac{P, \neg Q \rightarrow \| \; \|}{\neg Q \rightarrow \neg P \| \; \|}\;\text{Duality} \qquad \neg Q \rightarrow \neg Q \| \; \|}{\neg Q \rightarrow \neg(P \vee Q) \| \; \|}\;\alpha}{\rightarrow Q, \neg(P \vee Q) \| \; \|}\;\text{Duality}}{}$$

Let us call this *block 2.* Notice that although one fork terminates in a generalized axiom, the other does not—and no further rule is applicable.

On the other hand, we could have proceeded upward from block 1 using $\Box$-Thinning followed by Necessitation:

$$\dfrac{\dfrac{\rightarrow P, \neg(P \vee Q) \| \; \|}{P \rightarrow \|P\| \neg(P \vee Q)}\;\text{Necessitation}}{P \rightarrow \|Q, P\| \neg(P \vee Q)}\;\Box\text{-Thinning}$$

Then we could continue upward from here with the following block:

$$\dfrac{\dfrac{\neg P \to \neg P\| \; \| \qquad \dfrac{Q, \neg P \to \| \; \|}{\neg P \to \neg Q\| \; \|}\,\text{Duality}}{\neg P \to \neg(P \vee Q)\| \; \|}\,\alpha}{\to P, \neg(P \vee Q)\| \; \|}\,\text{Duality}$$

Call this *block 3*. Again, while one fork terminates in a generalized axiom, the other terminates in a sequent to which no rule is applicable.

Now, construct a Kripke model with worlds B_1, B_2, and B_3, corresponding respectively to blocks 1, 2 and 3. Set B_2 and B_3 to be accessible from B_1, corresponding to the fact that both blocks 2 and 3 yield block 1 in the proof attempt via Necessitation Rule applications. Finally, define truth at each world in accordance with initial sequents in the corresponding blocks that are not generalized axioms. Thus in B_2 take P to be true and Q to be false, in accordance with the sequent $P, \neg Q \to \| \; \|$ of block 2. Then in B_2, every sequent of block 2 from this one down is falsified. Likewise in B_3 take Q to be true and P to be false, and every sequent from $Q, \neg P \to \| \; \|$ down in block 3 will be falsified in B_3. Finally, at B_1, take P to be true (and make an arbitrary choice for Q). It is easy to check that every sequent in block 1 from $P \to \|Q, P\|\neg(P \vee Q)$ down (in fact, every sequent) will be falsified in B_1. Thus we have a Kripke model showing the invalidity of $(P \wedge \Box(P \vee Q)) \supset (\Box P \vee \Box Q)$.

Implementation is an easy matter, given the earlier discussion of the classical case. Here it is.

```
/*
  First, the various operators.
*/

:-op(100, fy, neg).
:-op(100, fy, box).
:-op(100, fy, dia).
:-op(110, yfx, and).
:-op(120, yfx, or).
:-op(130, xfy, imp).
:-op(130, xfy, iff).



/*
  Next, a classification of formula types,
  and instances.
*/

type(X and Y, conj, X, Y).
type(neg(X and Y), disj, neg X, neg Y).
type(X or Y, disj, X, Y).
type(neg(X or Y), conj, neg X, neg Y).
type(X imp Y, disj, neg X, Y).
type(neg(X imp Y), conj, X, neg Y).
type(X iff Y, disj, X and Y, neg X and neg Y).
type(neg(X iff Y), disj, X and neg Y, neg X and Y).
type(neg (neg (X)), doub, X, _).
type(box X, nec, X, _).
type(neg box X, pos, neg X, _).
type(dia X, pos, X, _).
type(neg dia X, nec, neg X, _).

/*
  thm(Lambda, Gamma, Delta, Omega) :-
  the sequent
  Lambda --> Gamma; Delta; Omega
  is provable.
*/

thm(Lambda, [Doubleneg | Gamma], Delta, Omega) :-
  type(Doubleneg, doub, X, _), !,
  thm(Lambda, [X | Gamma], Delta, Omega).

thm(Lambda, [Beta | Gamma], Delta, Omega) :-
  type(Beta, disj, Beta1, Beta2), !,
  thm(Lambda, [Beta1, Beta2 | Gamma], Delta, Omega).

thm(Lambda, [Alpha | Gamma], Delta, Omega) :-
  type(Alpha, conj, Alpha1, Alpha2), !,
  thm(Lambda, [Alpha1 | Gamma], Delta, Omega), !,
  thm(Lambda, [Alpha2 | Gamma], Delta, Omega).

thm(Lambda, [Nu | Gamma], Delta, Omega) :-
  type(Nu, nec, Nu0, _), !,
  thm(Lambda, Gamma, [Nu0 | Delta], Omega).

thm(Lambda, [Pi | Gamma], Delta, Omega) :-
  type(Pi, pos, Pi0, _), !,
  thm(Lambda, Gamma, Delta, [Pi0 | Omega]).

thm([L1 | Lambda], [L2 | _], Delta, Omega) :-
  (L1 = L2, ! ; thm(Lambda, [L2], Delta, Omega)).
```

```
thm(Lambda, [neg L | Gamma], Delta, Omega) :-
  thm([L | Lambda], Gamma, Delta, Omega), !.

thm(Lambda, [L | Gamma], Delta, Omega) :-
  thm([neg L | Lambda], Gamma, Delta, Omega), !.

thm(Lambda, [], [X | Delta], Omega) :-
  thm([], [X | Omega], [], []).

thm(Lambda, [], [X | Delta], Omega) :-
  thm(Lambda, [], Delta, Omega), !.

/*
  Finally, the driver.
*/

propk(X) :-
  thm([], [X], [], []).
```

7 Other Modal Systems

Several other normal modal systems can be created by easy modifications to the system presented in the previous section. The simplest is T—a sequent calculus dirseqT results by just replacing the π-Rule with the following, which corresponds to $X \supset \Diamond X$.

$$\frac{\Lambda \to \pi_0, \Gamma \| \Delta \| \pi_0, \Omega}{\Lambda \to \pi, \Gamma \| \Delta \| \Omega}$$

There is a corresponding single-clause change needed in the Prolog implementation.

For $K4$, the π-Rule is replaced with the following, which more-or-less builds in the validity of $\Diamond\Diamond X \supset \Diamond X$.

$$\frac{\Lambda \to \Gamma \| \Delta \| \pi_0, \pi, \Omega}{\Lambda \to \pi, \Gamma \| \Delta \| \Omega}$$

And for $S4$, we use the following version of the π-Rule, which has the effect of combining both the T and $K4$ changes.

$$\frac{\Lambda \to \pi_0, \Gamma \| \Delta \| \pi_0, \pi, \Omega}{\Lambda \to \pi, \Gamma \| \Delta \| \Omega}$$

Implementations for both $K4$ and $S4$ are a little more problematic however, since it is possible to get stuck in a loop, as the following $K4$ proof fragment shows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\to P, \Box P, \Diamond\Box P\| \; \|}{\neg P \to \|P\|\Box P, \Diamond\Box P}\,\text{Necessity}}{\neg P \to \Diamond\Box P\|P\|}\,\pi}{\neg P \to \Box P, \Diamond\Box P\| \; \|}\,\nu}{\to P, \Box P, \Diamond\Box P\| \; \|}\,\text{Duality}$$

To avoid this, one can impose a “modal depth” limit analogous to the depth-bound that occurs in leanT^AP to avoid a similar problem with quantification.

The quantifier machinery of leanT^AP can be added to the modal systems just outlined. One gets familiar quantified versions of the modal logics in which the Barcan formula is not assumed. We do not persue these issues here, because the development is relatively straightforward.

8 Conclusion

We have extracted an interesting sequent calculus from the Prolog program leanT^AP, and generalized it somewhat. On the one hand, such an approach clarifies algorithms, and makes correctness and completeness proofs simpler. On the other hand, we think the resulting formalisms are of intrinsic proof-theoretic interest, and deserve further study as formal calculi.

References

[1] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[2] Bernhard Beckert and Joachim Posegga. Logic programming as a basis for lean automated deduction. *The Journal of Logic Programming*, 28(3):231–236, 1996.

[3] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990. Second edition, 1996.

[4] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag, Berlin, 1968. Revised Edition, Dover Press, New York, 1994.