Additional Notes on Clausal Tableaux

Krysia Broda

February 13, 2003

1 Model Elimination Tableaux - Propositional Case

Note: These notes amplify certain issues raised in the slides and include a section on linear resolution. Most of the references in this document are a little aged. The main current sources of references are the books on the main booklist, namely the Handbook of Tableau Methods [9] and the various Conference Proceedings for TABLEAUX95 onwards, which are published in the Springer LNCS series [11]. Fitting's book [10] is also useful. There is a tableau website, most easily found by searching for semantic tableau in Google. (It is held at QMW.)

Model Elimination (ME) was a resolution refinement originally introduced in the early 1970s by Loveland; a similar method called SL-resolution was introduced at the same time by Kowalski, although from different motivations. The incorporation into tableaux was investigated by several people during the 1980s and continues today. The original definition of SL-resolution and ME was given arcanely in terms of chains and boxed and un-boxed literals. However, when the restrictions of these refinements are viewed from a tableau viewpoint they appear to be rather obvious and lead to many extensions that are less (or not at all) obvious from a resolution standpoint. *Model Elimination tableaux*, with which we will mostly be concerned, are tableaux with a particular structure that are also formed according to a fixed set of rules. They are also sometimes referred to as *connection tableaux.*

When the tableau method is restricted to clausal form there are basically two necessary rules together with the closure rule. The first is a rule for dealing with disjunctions, *viz.* a splitting rule which gives rise to as many branches as there are disjuncts. The second rule deals with quantifiers and will be introduced in Section 3.

We will define a ME-ground tableau according to its structure and a general ME-tableau according to its order of development.

The *structure* of a ground ME tableau is as follows: the initial clause is called a *top clause* and each subsequent clause in the tableau is used such that its leftmost literal always matches the immediate parent literal. (The order of literals in a clause can be altered so that this property is satisfied.) Other literals in a clause may either match with complementary literals higher in the tableau or immediately below (with a leftmost literal of another clause).

The *development* of this tableau is to close branches one at a time from left to right, completely closing one branch before moving onto the next. The effect is to select the next branch to close from those most recently introduced. When a clause is used in the tableau its literals can be ordered in any way, the matching literal always being on the left. The chosen order is then fixed for that occurrence. No literal needs to occur in a branch more than once, so a clause need not be used if it would lead to that situation. A tableau with no duplications is called *regular*. It is also possible to develop the tableau by selecting any open branch, not just the leftmost. We will show that the left-right development leads to a concise notation and also makes possible some useful extensions, and so will only briefly consider other development orders. An example is shown in Figure 1 for the set of ground clauses

$$\neg green(a) \vee \neg blue(a), blue(a), green(c), on(a,b), on(b,c),$$
$$green(b) \vee \neg green(c) \neg on(b,c), green(a) \vee \neg green(b) \vee \neg O(a,b)$$

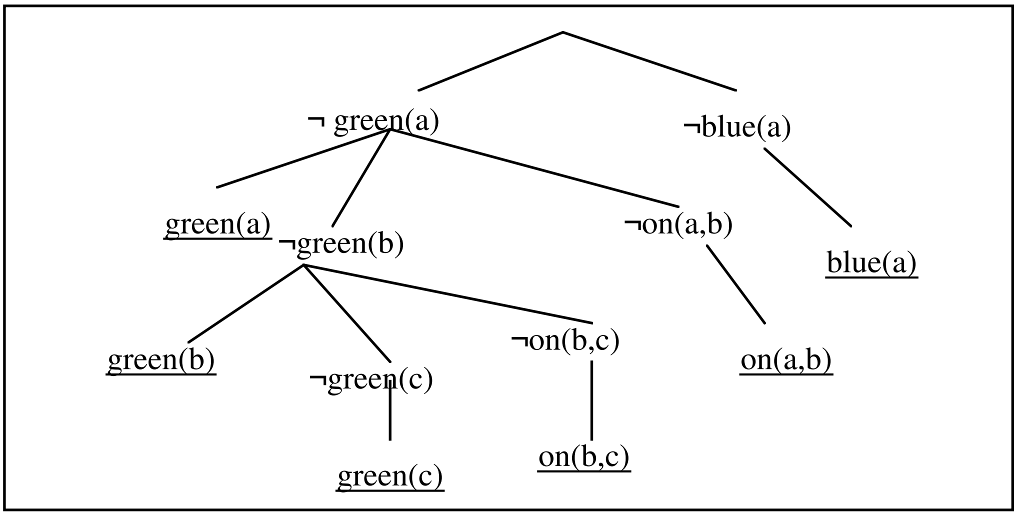

Figure 1:

You may notice that the clauses used in Figure 1 also happen to be Horn clauses and that the development follows a Prolog development. This is no coincidence. Tableau could have been used as the proof theory for logic programming (and I think it might have been more fruitful than using resolution!). Notice that the whole completed tableau corresponds to a single Prolog refutation. In a tableau using Horn clauses and in which the top clause has only negative literals there is no scope for any closure between a literal that is not the selected (leftmost) literal in a clause and another above it in the branch. (**exercise** Explain why.) Such steps are illustrated in Figure 2, for example in the very last branch of the first tableau (matching $\neg L$ with ancestor L and sometimes called *ancestor matching*) and in the third branch of the second tableau (matching $\neg M$).

The tableau of Figure 1 could have been found by using the Prolog computation rule, which is to develop the tableau from left to right, closing each branch before moving on to the next, or by a more liberal rule, which is to select branches in any order, regardless of whether the previous branch has been closed or not. At ground level, the result is the same, but for general clauses the search space that is generated is different in the two cases.

2 The Chain Notation

At each step in forming a ME tableau left-right depth-first, the relevant set of literals can be maintained in a single list, called a *chain*. Literals in a chain may be *unboxed*, corresponding to leaf literals in the tableau, or *boxed*, corresponding to internal literals in branches not yet closed. The various basic operations on a chain (and the corresponding tableau steps) are:

extension the leaf literal (at the left end of the chain) is boxed and any literals additional to the matching literal are added unboxed; (tableau step: matching a leaf literal with an input clause);

truncation boxed literals are removed from the left of a chain; (tableau step: moving from a closed branch to the next unclosed branch);

ancestor matching matching a leftmost literal with a complementary boxed literal; (tableau step: closing a branch with an internal node);

merge an unboxed leftmost literal is removed if it is identical to an unboxed literal to its right; (tableau step: closing a branch prematurely because the leaf literal appears as a leaf literal in an open branch to the right).

One benefit of the chain notation is that a search space can be represented more compactly. (Or even, that it can be represented at all!)

An extension of the notation [7] uses circled literals as well to represent *lemmas*. When a literal L has been refuted in a tableau branch (*i.e.* the tableau subtended below it has been closed) that refutation may be used again elsewhere as long as any ancestors that are needed are available in the new branch. At the ground level the latter can be guaranteed if the refutation of L is only to be re-used in branches below any of the ancestors that were necessary for the refutation. If L is refuted, then further occurrences of L may use the same refutation and this is achieved by pretending that $\neg L$ is also an ancestor in those branches in which the refutation below L may be re-used. In the tableau, such a literal $\neg L$ (sometimes also called a lemma) could be added to all open branches that lie below the used ancestor at greatest depth in the tableau. In the chain notation, when a boxed literal is leftmost, its complement is inserted as a circled literal to the left of the leftmost used ancestor. When circled literals are leftmost they are just truncated.

A simpler (but weaker) procedure is to insert the lemma $\neg L$ into the branches of all siblings of the refuted literal for it can be guaranteed that a refutation below L can be repeated a second time to refute L occurring in branches descending from any of the siblings of L. This can be done at the same time as selection and so there is no need for a different notation.

Figure 2 illustrates merging and the use of lemmas in propositional ME tableaux. The chain notation is also shown for these two examples.

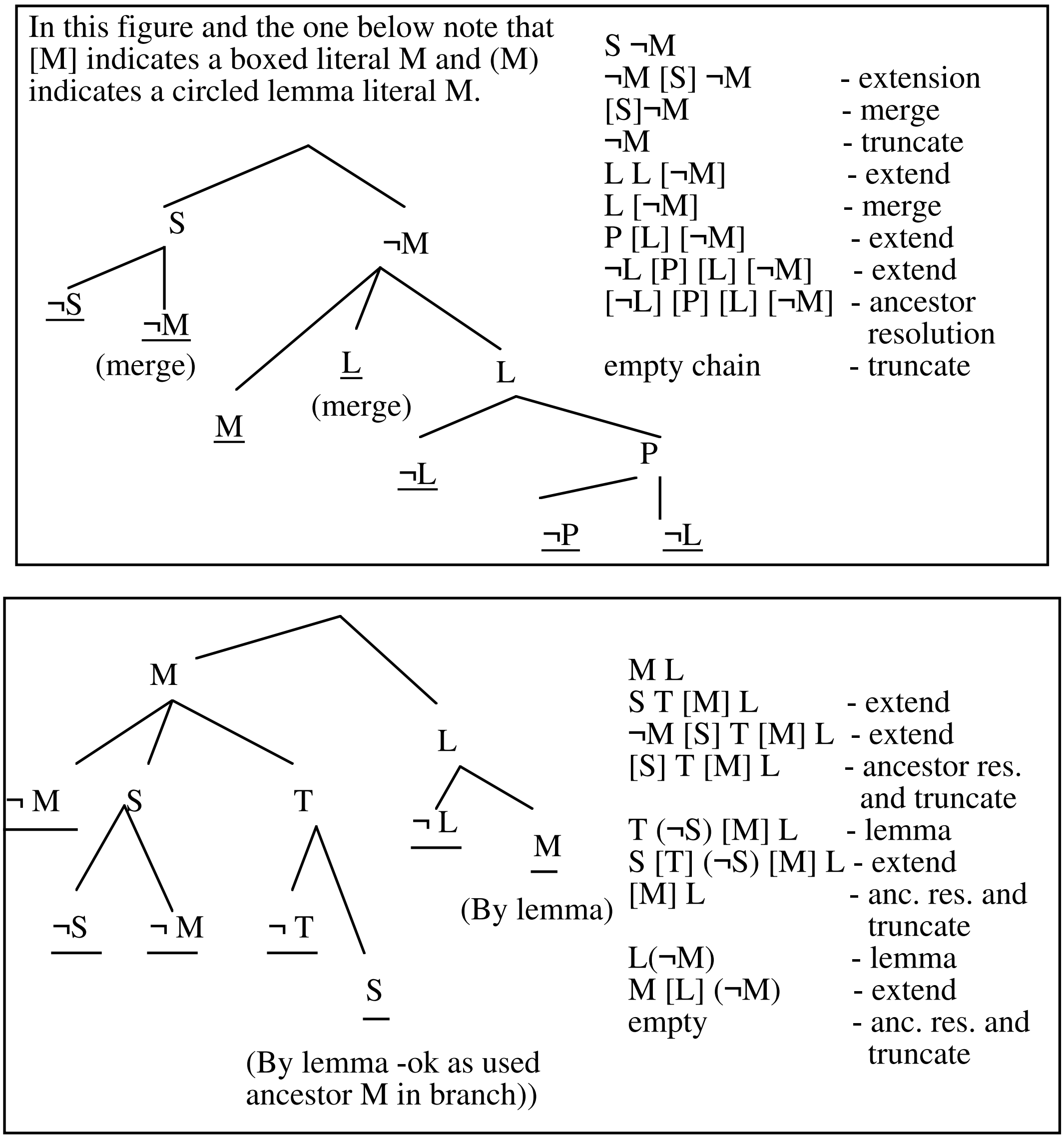

Figure 2:

Note that merging and lemma generation are not generally compatible with each other – a lot of care has to be taken. Consider the clauses $\{A \vee B, \neg A \vee B, \neg B \vee A\}$, which are satisfiable. A simple-minded combination of lemmas and merging enables a tableau formed using the clauses in the given order to be closed, incorrectly.

When generating a ME tableau there may be a choice of clauses to use at some step. In that case they are tried in order, trying to close the tableau beneath the resulting new branches and back-tracking to try another matching clause on failure. If there is a literal beneath which there is no closure possible, then any further occurrence of that literal in the tableau may also be failed under certain circumstances. These circumstances are if the available ancestors for the second occurrence are a subset of those available to the first occurrence.

3 First Order ME Tableaux

For general clauses the propositional rules are augmented by a combination of the γ-rule and splitting rule that uses a (new) free-variable for each bound variable. Free variables are applied to a clause before it is introduced via the $\vee$-splitting rule. Usually, since the leftmost literal is matched immediately, the instance placed in the tableau is that which is complementary to its matching parent. Figure 3 shows a tableau formed using the clauses

$$\neg P(x,a), P(y,y) \vee P(y,z) \vee \neg Q(z), Q(b) \vee Q(a), \neg P(a,b)$$

In particular, notice that instead of adding the free variable instance $P(y1,y1) \vee P(y1,z1) \vee \neg Q(z1)$, the unification to achieve a match between complementary $\neg P(x1,a)$ and $P(y1,y1)$ is made as part of the step. This saves unnecessary variable propagation in the tableau.

Note carefully in Figure 3 that when $\neg P(x1,a)$ has been matched and $x1 = = a$ that $x1$ cannot be bound to any other literal. In the next section it is considered under what circumstances this rule could be relaxed.

Just as in logic programming, developing search spaces in a depth first manner can lead to problems with infinite branches. This is avoided by limiting the depth of the generated tableau to a particular value d. If there is no possible closure at maximum depth d (after trying all possible unifiers), the depth d is increased. In case of a logic program simulation by tableau, in which the top clause is all negative literals and all clauses have a maximum of one positive literal, the only structure it is necessary to keep is the leaf literals of open branches as these can only be matched from below. These literals correspond to the current goal list of the logic program execution that is being simulated. In non-Horn clause tableaux literals may also match with ancestors in a branch and so it is necessary to keep the ancestor list for each literal. This is much less expensive if a tableau is developed from left to right and depth first as only one current ancestor list needs to be maintained. The list is updated as each branch is closed. (The fact that bindings are propagated in a tableau makes it particularly easy to implement general ME tableau in Prolog.)

If the chain notation is used in order to represent the search space, almost any example will show that, for example, if instead of left-right development the tableau is developed

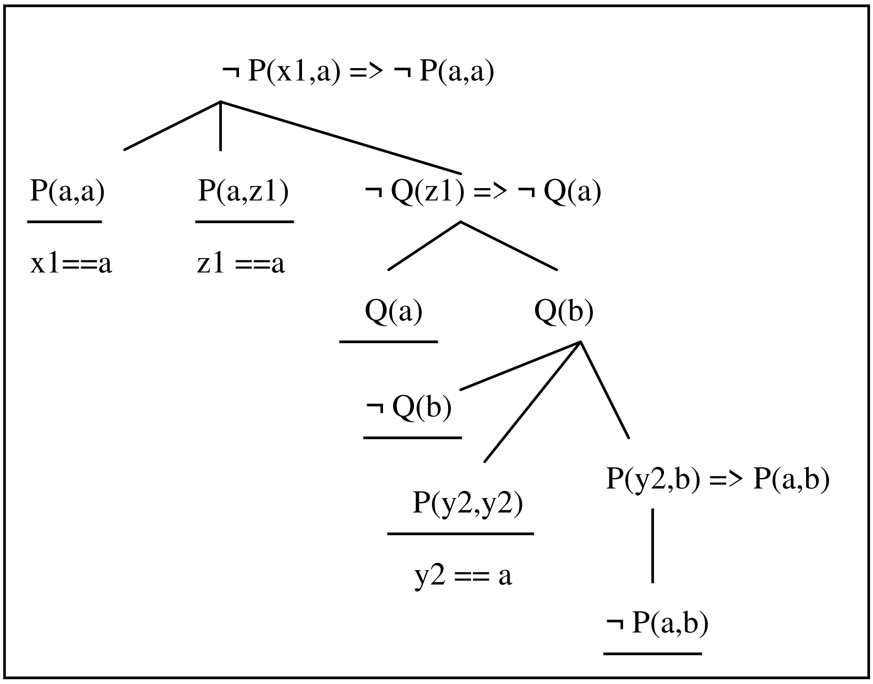

Figure 3:

from right-left, then a different search space ensues. However, unlike in logic programming, very little research has been made into benefits of reordering the branches. The relatively recent work on "nagging" [12] reported in the slides begins to look at the issue.

The additional steps of merging and lemma generation are still possible in free variable tableaux, but only with care. In the case of merging, any bindings are propagated to the whole tableau, which can result in a larger tableau having to be constructed than might otherwise have been the case. As for the use of lemmas, suppose the refuted literal L still has unbound free variables; this indicates that the lemma is $\forall L$, where the universal closure is made over the free variables. Suppose, for example, the literal L is initially $L(x, y)$ in the tableau, with x and y being free variables and that once the refutation below $L(x, y)$ has been completed, x has been bound to the constant a and y to the term $f(z)$ for free variable z. The form of L in the tableau is now $L(a, f(z))$. Suppose also that no ancestors of L were used in the closed tableau below L. If only the literal $\neg L(a, f(z))$ is added to the appropriate branches, to represent the lemma, it can only be used with one instantiation. The lemma that has been found is actually $\forall z L(a, f(z))$, for the refutation will work for any value of z, and so $\forall z \neg L(a, f(z))$ should be added to other branches that could use the lemma. This literal can be used with more than one instantiation. Rather more care has to be taken in case the lemma uses ancestors containing the variable z and because of this a lemma is often restricted to the case when no ancestor is used. In that case the universal lemma can be added to the top of the tableau. Examples of the possible use of lemmas and merging in first order tableaux are shown in Figure 4.

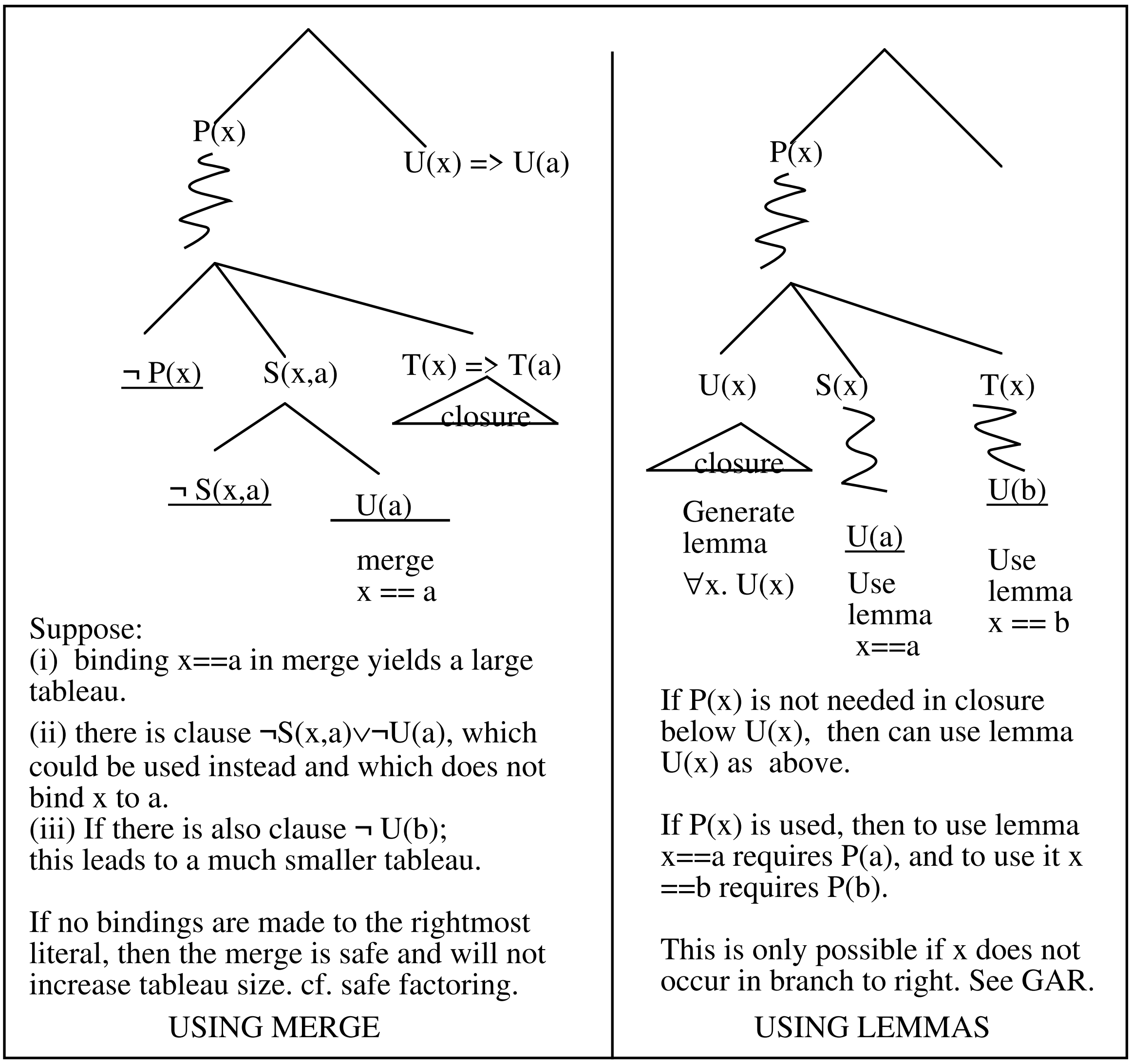
P(x)
U(x) => U(a)
¬ P(x)
S(x,a)
T(x) => T(a)
closure
¬ S(x,a)
U(a)
merge
x == a
Suppose:
(i) binding x==a in merge yields a large tableau.
(ii) there is clause ¬S(x,a)∨¬U(a), which could be used instead and which does not bind x to a.
(iii) If there is also clause ¬ U(b);
this leads to a much smaller tableau.
If no bindings are made to the rightmost literal, then the merge is safe and will not increase tableau size. cf. safe factoring.
USING MERGE
P(x)
U(x)
S(x)
T(x)
closure
Generate lemma
∀x. U(x)
U(a)
Use lemma
x==a
U(b)
Use lemma
x == b
If P(x) is not needed in closure below U(x), then can use lemma U(x) as above.
If P(x) is used, then to use lemma x==a requires P(a), and to use it x ==b requires P(b).
This is only possible if x does not occur in branch to right. See GAR.
USING LEMMAS

Figure 4:

4 Completeness of ground ME tableaux

4.1 Induction Proof for Completeness of ME Tableaux

(Note that the proof given here works for any branch selection strategy, not just left-right selection.) It uses induction on the number of non-unit clauses in a minimally unsatisfiable set.

a In case $k = 0$, all clauses are unit clauses and a correct refutation is obvious.

b The induction hypothesis states that a ME tableau can be constructed from any number m, $0 \leq m < k$, of non-unit minimally unsatisfiable clauses (*i.e.* removing any clause yields a satisfiable set) beginning from some top clause. Suppose that a set of clauses S with k non-unit clauses is minimally unsatisfiable.

Select a non-unit top clause $t_1 \vee \ldots \vee t_n$. Each branch of the top clause can be dealt with independently as in step (c).

c For each i, $1 \leq i \leq n$, let S_i be a minimally unsatisfiable subset of $(S - \{t_1 \vee \ldots \vee t_n\}) + \{t_i\}$. Note that, as S is minimally unsatisfiable, then t_i will be a member of this minimally unsatisfiable subset and hence necessary for the refutation of $(S - \{t_1 \vee \ldots \vee t_n\}) + \{t_i\}$, so t_i cannot be a pure literal and there must be a matching clause of the form $C_i = \neg t_i \vee b_0 \vee \ldots \vee b_m$ in S_i, which acts as the new top clause. By hypothesis, a ME tableau can be constructed using C_i as top clause. The first branch closes using t_i. Each of the n closed tableau can be connected to the original top clause, such that the $\neg t_i$ matches with the original t_i in $t_1 \vee \ldots \vee t_n$.

In order to be able to use the hypothesis it must be confirmed that the sentences left to use in S_i are indeed unsatisfiable (and hence can be made minimally unsatisfiable). The available clauses in the branch are

$$(S - \{t_1 \vee \ldots \vee t_n\}) + \{t_i\}$$

If these were satisfiable, then S would also have been, because any model must make t_i true and hence $t_1 \vee \ldots \vee t_n$ true also. Also, it needs to be confirmed that some clause of the form $\neg t_i \vee b_1 \vee \ldots \vee b_m$ belongs to a minimally unsatisfiable subset of S_i. By contradiction, if $S_i - \{C : C \text{ is of the form } \neg t_i \vee b_1 \vee \ldots \vee b_m\}$ were unsatisfiable then t_i could not be a member of the minimally unsatisfiable S_i.

It remains to consider how such a tableau can be lifted according to the ME-resolution procedure – *i.e.* left to right and depth first. First, note that if each clause is lifted using a new crop of free variables a unifier for the set of equations resulting from each closure certainly exists. Hence a most general unifier exists. Now, unification of a set of equations can be made by taking the equations in any order; the result is the same in all cases. In particular, they can be taken in the ME-order.

5 Generalised Closure Rule

(**Note:** I sometimes refer to the Generalised Closure Rule as GAR, short for Generalised Ancestor Resolution.)

Any tableau with open branches can be written as an equivalent quantified expression in which the quantifiers are distributed as far as possible. A tableau T_0 with open branches descending from literals $A_1, \ldots, A_n$ is written as $\forall[(A_1 \wedge T_1) \vee \ldots \vee (A_n \wedge T_n)]$, where the quantification is over variables in $A_1, \ldots, A_n$ and $T_1, \ldots, T_n$ are the representations of the trees below $A_1, \ldots, A_n$, if any. The quantifiers can be further distributed according to the following two equivalences:

$$\forall x[A[x] \vee B] \equiv \forall x.A[x] \vee B \text{ where } x \text{ does not occur in } B \tag{1}$$

$$\forall x[A[x] \wedge B[x]] \equiv \forall x.A[x] \wedge \forall x.B[x] \tag{2}$$

For example, consider the clauses

$$\{\neg P(x,a), P(y,y) \vee P(y,w) \vee \neg Q(z), Q(b) \vee Q(a), \neg P(a,b) \vee \neg P(a,c)\}$$

The representations before and after the first closure for the tableau developed as in Figure 5 are, respectively,

$$\forall y[P(y,y) \vee \forall w.P(y,w)] \vee \forall z.\neg Q(z)$$

and

$$\forall w.P(a,w) \vee \forall z.\neg Q(z)$$

The standard development would then probably match $P(a,w1)$ with $\neg P(a,b)$ so binding $w1$ to b, with the result that $P(a,w1)$ cannot also match $\neg P(a,c)$. But if $P(a,w1)$ is treated as $\forall w.P(a,w)$, then as many copies as are needed are implicitly available and both $\neg P(a,b)$ and $\neg P(a,c)$ can be closed using it. This is sound, as $\{\forall w.P(a,w), \neg P(a,b) \vee \neg P(a,c)\}$ is unsatisfiable. The saving can be considerable.

A quantifier can be distributed across more than one disjunct (branch) exactly when the bound variable is not shared between the two branches. A quantifier can be distributed across conjuncts always. A literal with at least one free variable that can implicitly be universally quantified (as in $(\forall x.)Q(x,a)$ for example) is called *universal.* The free variables are also called *universal variables.* Instances of such a literal, in which universal variables are instantiated differently (but not non-universal variables) are called *universal instances.* There can be as many of them as are needed. A simple to implement method for ME tableau and incorporating universal instances is the following:

> A variable is treated as universal if it occurs in exactly one open branch of a tableau. Any bindings to such variables (from closure) can be ignored, because there is always (implicitly) available a second fresh copy of the literal in which it occurs.

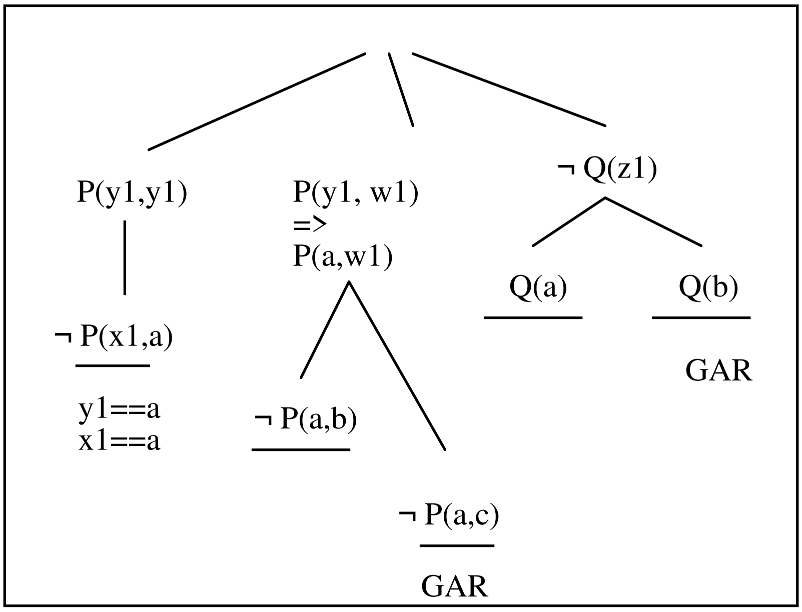

Figure 5:

This ignoring of bindings rule is called the *Generalised Closure Rule* (or sometimes Generalised Ancestor Resolution). A constructive argument for the soundness of this rule can be seen as follows.

In Figure 6 is a tableau in which the first open branch could not normally close, since the literals are $A(a, y1)$ and $\neg A(b, u1)$. But suppose a second copy of the tableau enclosed within the dotted loop is made and placed beneath $\neg A(b, u1)$, then closure can occur. There are now two copies of the remaining open branch, namely $B(y1)$ and $B(y2)$. But these can always be merged as the different instances of $A(x1, y1)$, due to different bindings for $x1$, did not affect $B(y1)$ as $x1$ did not occur in it. Thus the tableau can still be formed (although not exactly an ME tableau). Alternatively, soundness is guaranteed by considering the equivalent quantified expression for open branches in a tableau, and showing that using GAR is a valid deduction.

6 Generalising Prolog

One of the problems with ME-refinements is that all literals in an input clause must be available for selection. Searching for a matching clause can become quite expensive if clauses are large. We give a way to restrict clauses so that only one literal is used for matching. Another difficulty with ME strategies is due to the left-right depth-first search for a refutation. Such a search strategy is incomplete and so, instead, tableaux of a certain depth are generated, and if no closure results tableaux of greater depth are formed, *etc.* This is called *iterative deepening*. Both of these features are used in the refinement which

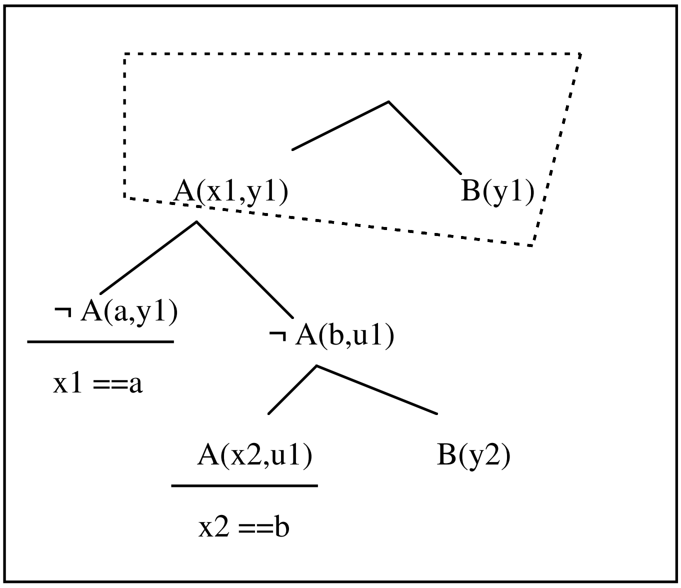

Figure 6:

we will call the *Intermediate Lemma Extension* (not to be confused with the earlier use of lemmas).

The method of the Intermediate Lemma Extension is shown on the slides. The rules are simple:

- One positive literal in each clause is identified as the *conclusion* literal. The others are called *lemma* literals.
- All refutations begin from some clause with only *negative* literals.
- Only the conclusion literal of a clause can be used to match a leaf literal. The negative literals in the matching clause are further extended. The lemma literals are not extended but collected into a *lemma clause*.
- After refuting all negated literals the lemma clause is asserted, one of the positive literals becoming its conclusion.

At first sight it seems as if many intermediate lemmas might be formed that are of no use for solving a particular goal because the simplest implementation is to generate all lemmas to a particular depth. Indeed this can be the case! But with iterative deepening it is difficult to avoid their (re)generation. To do otherwise and use a lemma as soon as it is formed would need quite a lot of careful book-keeping.

However, trials show that the method is actually quite good in practice. The reason seems to be that the use of lemmas keeps the effective depth of tableaux fairly small, a completed refutation being formed from several sub-tableaux, none of which is very deep. So the search space does not become prohibitively large after all.

When lemmas are formed, only those which are not subsumed need to be kept, and any clauses subsumed by a lemma can be deleted. To avoid the necessity of factoring lemmas (apart from simple merging of identical literals) the simplest approach is to allow more than one positive literal from a clause to match with its immediate parent (if possible). It is also possible that ancestor resolution might successfully be incorporated (although this seems to defeat the possibility of a fast search – see below). An early implementation appeared in an MSc project [1], which also incorporated some connection graph processing. A second attempt is being undertaken as a group project (2003). One of the advantages of the method is that no searching of branches for closure needs to be made, so the search for intermediate lemmas is very efficient and could use Prolog. Incorporating ancestor resolution would imply searching branches although it might be compensated for by smaller lemmas. Any lemma that is being formed that will eventually be subsumed can also be abandoned. These variations are being investigated.

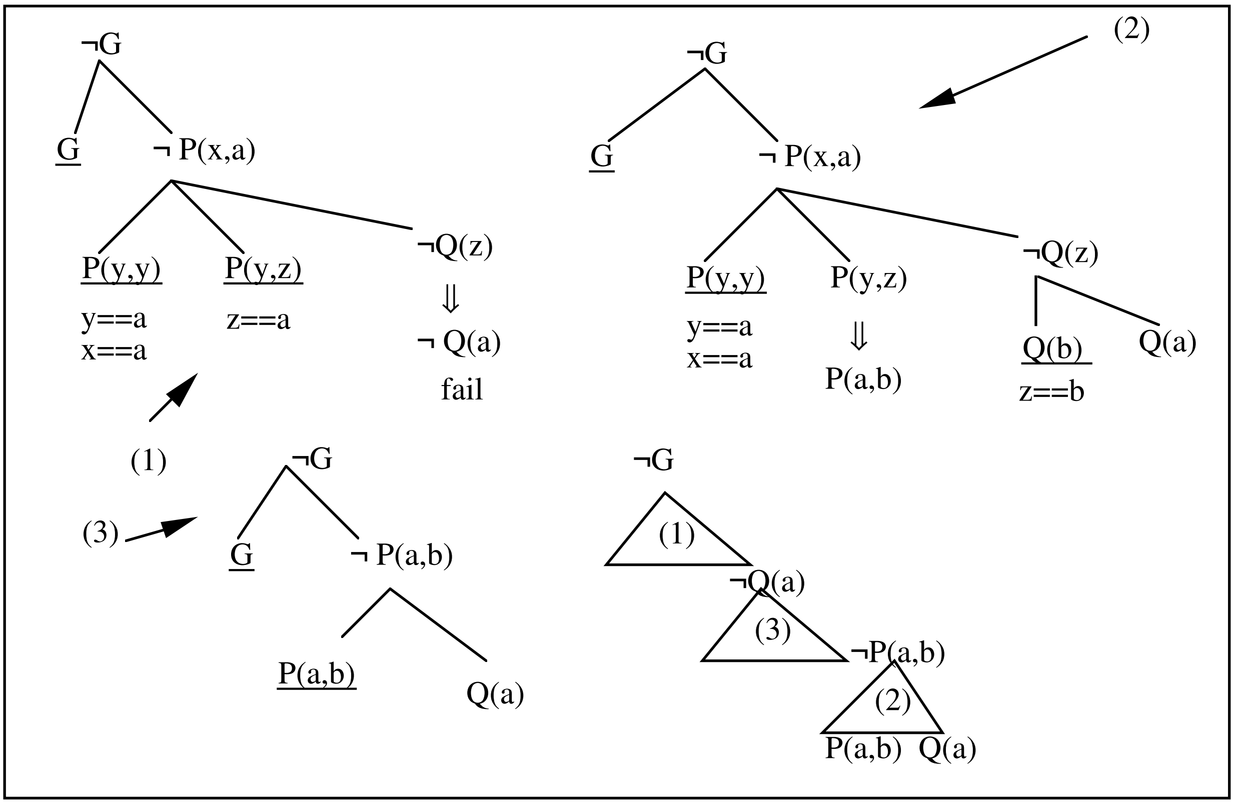

Figure 7:

For example, in Figure 7 a closed tableau using the Intermediate Lemma Extension is shown for the clauses $\{\neg P(x,a), P(y,y) \vee P(y,z) \vee \neg Q(z), Q(b) \vee Q(a), \neg P(a,b)\}$. Tableau (1) is a failed attempt; tableau (2) finds the lemma $P(a,b) \vee Q(a)$; tableau (3) uses this to find the lemma $Q(a)$ which can then be used to complete the first tableau. The structure of the final tableau is shown in the last diagram. In this example, each of the negative clauses has been augmented by a G literal. This means the initial top clause can be $\neg G$ and it is the only negative clause. It just makes things a bit simpler to implement that way.

Completeness of the method for ground clauses can be shown by induction on the number of "lemma literals" in the given clauses, where the number of *lemma literals* in a clause is $max(0, \text{number of positive literals} - 1)$ (see slides). (All in all, I rather like this refinement.)

7 KE

The tableau variation called KE, described on the slides, is a sort of half-way house between Davis Putnam and ME. The propositional version was introduced by D'Agostino and Mondadori and they have made several studies of its properties, although first order versions have not been very well investigated. An implementation called "WINKE" (written by Jeremy Pitt and Ulrich Endriss [8]), is a package using the method for pedagogic use and although it uses quantifiers, it is not particularly efficient in their use.

To see what the problems are for first order KE, it is easiest to assume Skolemisation occurs before a refutation is made, so only universal-type quantifiers need to be considered. The rule for dealing with outermost universal quantifiers could be exactly as in the standard free variable tableau. So, for example, the sentence $\forall x[P(x) \rightarrow \forall y.Q(x,y)]$ might first yield $P(x1) \rightarrow \forall y.Q(x1,y)$, before being used in an $\rightarrow$ rule. But the two rules could also be combined into a single step. If the $\rightarrow$ rule is to be used then there must be some sentence of the form $P(X)$. If not, then a PB application could be used to introduce such a sentence, with a free variable.

In case the quantifier is not outermost, for instance in $P(a) \vee \forall y.Q(b,y)$, the sentence $\neg Q(b,c)$, for example, could be used to derive $P(a)$ by the $\vee$ rule. If no suitable $\neg Q$ sentence existed, then it could be introduced by a PB application of $\neg Q(b,y1)$. This is rather like distributing quantifiers in the ME tableau.

There is still much to discover about the first order version of KE.

8 Completeness of Ground Resolution using Tableaux

Tableaux can be used to derive a resolution proof, in a similar way to that used to derive a resolution proof from semantic trees. The method shown on the slides consists of the following steps:

a) for unsatisfiable set of ground clauses S, form any closed tableau.

b) remove[1] clauses that give rise either to a duplicate literal or to a literal that is *unused* occurring in a branch. (An unused literal is an internal literal which is not matched by any leaf literal).

There may be more leaf literals than internal literals as more than one leaf literal may match the same internal literal. Because of the properties in (b), in any clause C with only leaf literals at least one of its leaf literals, L, will match the literal L^c immediately above in clause C'. (L^c is the complement of L.) Call this property (c). The resolvent of C with $C' =$ $\{$ literals in C apart from any occurrences of $L\}$ + $\{$literals in C' other than $L^c\}$. The literals in the resolvent derived from C can be put in the tableau in place of L^c and the clause C removed. The resolvent is then in the tableau in the position of the priginal C'. *e.g.* in the example given in the slides, the clause $B \vee R$ is resolved with $M \vee L \vee \neg B$ and B is replaced by $M \vee L$. If there are several literal occurrences of L in C then they are factored first. The resolvents that are formed in this way will give the proof.

After such a resolution step, it can be easily shown that property (b), and hence property (c) also, still hold and that the tableau is still closed; so the resolvent forming operation can be repeated. Each time this occurs the tableau reduces in size, until there is an occurrence of the pattern of a single root node labelled by X^c with one or more leaf nodes directly below labelled by X. This is processed to obtain the empty clause.

9 Non-clausal Sentences and a ME-like Method

The idea of ME can be extended to *q-clauses*, which consist of (possibly universally quantified) disjunctions of conjunctions of literals, or quantified literals (similar to the notation introduced to explain the generalised closure rule). Literals are rather easier to deal with practically and so a method that compiles sentences into quasi-clauses is useful.

A *q-literal* is a literal or universally quantified literal.

A *q-conjunction* is a conjunction of q-literals.

A *q-clause* is a disjunction of q-conjunctions. It may also be universally quantified.

For example, $\forall x[(P(a,x) \wedge Q(x)) \vee (Q(b) \wedge \forall y.P(y,x))]$ is a q-clause.

The translation of first order sentences into q-clauses goes as follows:

- A sentence is split into its top level disjuncts, rewriting $\rightarrow$ and $\leftrightarrow$ and $\neg\neg$ as in standard tableau rules. Each disjunct will be a conjunction of 1 or more sentences.
- Existential-type quantifiers are Skolemised.
- Any universal-type quantifiers are distributed across the disjuncts and the conjuncts within each branch as far as possible.

[1] See slide for what "removal" is in practice.

- Any remaining outermost quantifer of a sentence is replaced by a free variable.
- Any remaining non-q-literals are replaced by definitions (see below), unless there is exactly one non-q-literal, in which case it is recursively translated into a q-clause.

For example, $\forall x[P(x) \to Q(x)) \to R(x)]$ becomes $(P(x1) \to Q(x1)) \to R(x1)$, which is a single non-q literal, which becomes $(P(x1) \wedge \neg Q(x1)) \vee R(x1)$, and $\forall x(P(x) \vee Q(x)) \wedge \forall y(R(y) \vee S(y))$ is a q-clause with two non-q-literals.

There may still be a (possibly universally quantified) non-literal sentence within a conjunct, which may have one of the forms S, $\forall y.S$, $\neg S$ or $\forall y.\neg S$, where S is not a literal. It is replaced by, respectively, P_S, $\forall y.P_S(y)$, $\neg P_S$ or $\forall y.\neg P_S(y)$, where $P_S \leftrightarrow S$ or $P_S(x) \leftrightarrow S(x)$. S is then recursively translated into a q-clause. P_S is called a *definition literal*.

If a literal P_S is selected, then it will only match with its definition, one branch closing and the other extending with the q-clause for S. Definition predicates were first introduced by Plaistead and Greenbaum[6].

For example, the above non-q-clause is compiled into $\forall x.P1_S(x) \wedge \forall y.P2_S(y)$, where $P1_S$ is defined by $\forall y[P1_S(y) \leftrightarrow (P(y) \vee Q(y))]$ and similarly for $P2_S$. If the q-literal $\forall x.P1_S(x)$ is selected, it can only match with the "only if" half of the above sentence definition, namely $\forall y[P1_s(y) \to (P(y) \vee Q(y))]$. The rest of the sentence $(P(y) \vee Q(y))$ would have been compiled in turn into the q-clause $P(y1) \vee Q(y1)$ with free variable $y1$.

Sometimes, a clever choice of P_s will enable several sentences to be replaced by instances of P_s. For example, $\forall x[P(x,a) \to Q(x,a)]$, $\forall x[P(x,b) \to Q(x,b)]$ and $\forall x[P(x,x) \to Q(x,x)]$ could all make use of $\forall y \forall z[P_S(z,y) \to (P(z,y) \to Q(z,y))]$, leading to $\forall x.P_S(x,a)$, $\forall x.P_S(x,b)$ and $\forall x.P_S(x,x)$ respectively.

The difference between clausal and q-clausal tableaux is that expansion at a node can result in more than one q-literal. Thus a choice must be made of which q-literal to select from a leaf node. The easiest approach is to select one of the q-literals and allow the others to be used in ancestor resolution (closure) steps, which may be generalised closure steps.

10 Linear Resolution

An example of a linear refutation is a resolution simulation of a Prolog proof beginning from an all-negative literal clause; each resolvent has the previous resolvent as one of its parents.

More generally, a *linear deduction*, with resolvents $R_0, R_1, \ldots, R_n = \Box$, is a deduction from *top clause* R_0 such that $\forall i \geq 1, R_i =$ resolvent of R_{i-1} either with an input clause, or with $R_j, j \leq i-1$.

One benefit of allowing only linear deductions is that the search space becomes smaller, and is usually represented and searched as a tree.

As an example, consider the (by now well-known) clauses

(1) $\neg Ha$
(2) $Fx \vee Hx$
(3) $\neg Gz \vee \neg Fb$
(4) $\neg Fx \vee \neg Hb$
(5) $Gx \vee \neg Fx$

The search space, with $\neg Ha$ as top clause, is given in Figure 8. (This search space is rather special in that every resolvent is a unit clause.)

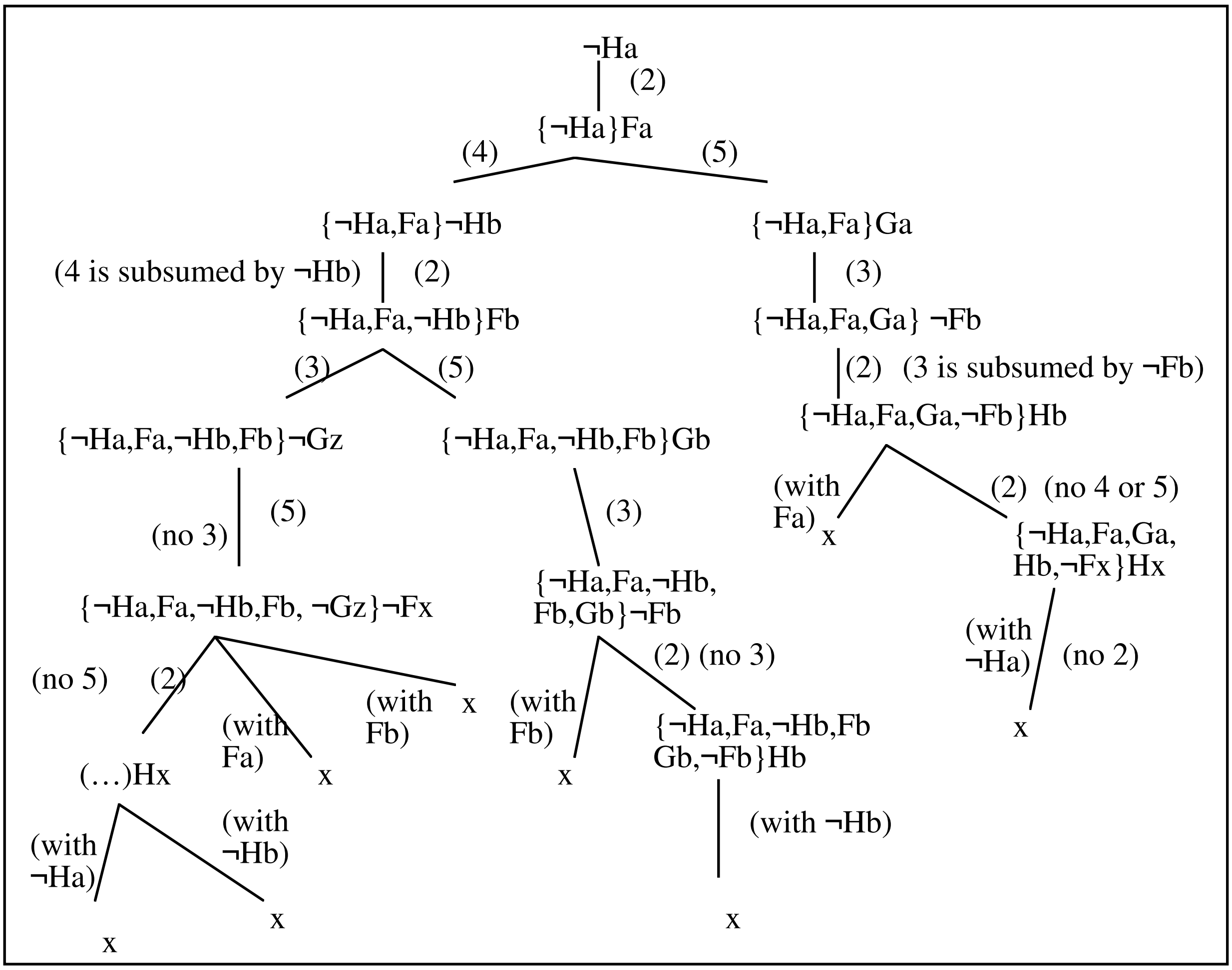

Figure 8:

At each node the previous resolvents (ancestors) are in brackets ({ }) and the current resolvent is not. Resolvents or input clauses that are subsumed are not used to further extend a node, nor is a resolvent that is subsumed, kept. You will notice that there are still duplications of refutations, in the sense that essentially the same steps are made, but no exactly identical nodes.

This tree space can be searched depth first (to a limited depth to maintain completeness) resulting in an efficient stack based implementation. Each new resolvent is stacked, the stack at any stage being all previous resolvents along the path to the current node. Of course, the stack as well as the input clause list must be searched for possible clauses to form new resolvents. (In Horn clause logic programming applications, only the input clauses need to be searched as two resolvents can never resolve.) Whereas the non-linear refinements considered so far used saturation methods and were always confluent, this is not so with linear refinements. It is often the case that a branch can lead to failure, in the sense that no more linear resolvents can be formed from the clauses at the node, which do not yet include the empty clause. However, there will always be some branch which will lead to the generation of $\Box$, so completenesss is not lost. Subsumption and tautology deletion are also compatible with the linear strategy, as are various ordering refinements.

We will not consider general linear resolution further, as ME tableaux are more easily implementable. There are also drawbacks to linear strategies. For example, some problems are solved most naturally by arguing both forwards from some clauses and backwards from others to obtain a contradiction. This is especially so if the original goal was an implication, where the best approach is often to argue forwards from the antecedent and backwards from the conclusion.

An example of this could be a goal of the form $\forall y[integer(y) \to P(y)]$, for some predicate P. Typically, this might be shown by considering three cases: *negative*(y), *zero*(y), *positive*(y). *i.e.* assuming each of these in turn and trying to prove $P(y)$. This is in contrast to a strict linear refutation from $\neg P(y)$ which might eventually derive the clause (say) $\neg$*negative*(y), which would resolve to give (perhaps) *zero*$(y) \vee$ *positive*(y). This proof structure is not as natural as the first and if a theorem prover is to report its refutations, then a requirement is that they should be reasonably natural.

Another disadvantage is that linear refutations are often longer than non-linear ones.

To maintain completeness of linear deduction it is necessary to begin with a top clause that is necessary to the unsatisfiability of S; *i.e.* removing the top clause from S renders S satisfiable. Any clause will do if S is *minimally unsatisfiable* – *i.e.* S-C is satisfiable for each clause C in S. Usually, a clause from the conclusion is used as it is likely to be of this kind.

With just the simple linear restriction, there is still much flexibility regarding the choice of literal to resolve upon next in the current resolvent. This flexibility can be further restricted by means of various literal orderings. It is also possible to restrict the allowable resolution steps which can be performed with an ancestor (a previous resolvent). One such restriction is to permit resolution with an ancestor only if the literal resolved upon is the same as that resolved upon when the ancestor was the most recent resolvent. This turns out to be a very satisfactory restriction and is the one that naturally arises in ME tableaux simulation of linear resolution.

The description of the restrictions for a resolution proof are rather complex. They are given here just to emphasise the point!

10.1 Restrictions on Linear Resolution

These restrictions were introduced into Model Elimination (Loveland) and SL-resolution (Kowalski). The first implementation of ME that incorporated Prolog technology into theorem provers is described in [5]. It was not described as a tableau procedure at that time.

There are two necessary restrictions which are incorporated into linear resolution using clauses S to give SL or ME resolution. In addition to selecting the previous resolvent as one of the parents (the linear restriction) and beginning with a top clause from a minimally unsatisfiable subset of S we require that:

- the selected literal in a resolvent R_i must be chosen from those literals most recently introduced (if any); (*i.e.* (in general) those literals that are not descended from R_{i-1});
- the variables in the resolvent are not renamed;
- if $R_1 \ldots R_k$ is the proof so far and R_{k+1} is produced by resolving R_k and R_r (say), $r < k$, resolving on literals L and M in R_k and R_r respectively, then M must be the same literal that was resolved upon when R_r was used to produce R_{r+1}. Note that the restriction on not renaming variables in the resolvent means that L and M should unify *without* the renaming of variables in them.

  For example, if $P(x)$ in a resolvent is resolved with $\neg P(a)$, then x is bound to a. If later, $P(x)$ is again resolved, then it can only be resolved with $P(a)$, not $P(x)$ as would normally be the case.

  These properties guarantee that the literals in the resolvent R_{k+1} which derive from those in R_k and which were previously derived from R_r will factor (and should be factored) with those in R_{k+1} derived anew from R_r. This also forces $r < k$, for R_k cannot be resolved with itself and meet this restriction.

If these restrictions are maintained, then any particular linear refutation P (or proof) can be represented by a ME tableau refutation T as follows:

- The top clause in P is identified with the top clause of T with the selected literal L leftmost.
- Each resolvent of P is identified with the set of leaf nodes in open branches in T.
- The selected literal is always made the leftmost one, which corresponds to the first restriction above, and either resolves with another input clause or with a previous resolvent. In the latter case, the literal available for resolution in T is in the branch

going from the root to the selected literal. This corresponds to the first restriction in the second requirement.

- The other restriction of the second requirement is naturally achieved in T, since bindings are propagated. The effect is that the same instance of the earlier resolvent is used. The second criterion allows the literals derived from the ancestor and those corresponding literals derived from the clause containing the selected literal to be merged. This is the only factoring that needs to occur. In T the factoring is implicit.

As an example of the difference between employing the restrictions on ancestor instances and not doing so, consider the clauses $Pxy \vee Qyx$ and $\neg Pub \vee \neg Pau$; Pxy and Pub unify with m.g.u. $\theta = \{x == u, y == b\}$ and the resolvent is $(\neg Pau \vee Qyx)\theta = \neg Pau \vee Qbu$. The occurrence of Pxy from $Pxy \vee Qyx$ has now become Pub and this does not unify with $\neg Pau$, so the requirement to unify without renaming is not possible. On the other hand, if the clauses $Pxy \vee Qyx$ and $\neg Pau \vee Qbu$ are resolved, (say the ancestor **is** renamed) the resolvent is $Qua \vee Qbu$; in this case the two descendants from the Q literal cannot be factored. This is exactly the case when ancestor matching is impossible in the tableau. On the other hand, consider the clauses $Axz \vee Bxz$ and $\neg Aau \vee \neg Ayy$. Unifying Axz with Aau yields the m.g.u. $\theta = \{x == a, u == z\}$ and the resolvent $(\neg Ayy \vee Bxz)\theta$ or $\neg Ayy \vee Baz$. If the resolvent is ancestor resolved, but renaming the ancestor clause as $Ax1z1 \vee Bx1z1$ and unifying Ayy with $Ax1z1$, the resolvent is $Bx1x1 \vee Baz$,which will factor to Baa, exactly the result obtained if the ancestor instance is not renamed.

Literals such as the B literals in this example may factor even when the ancestor literal does need renaming to achieve a unification. But if the ancestor literal does not need renaming, it is guaranteed that factoring can occur.

/small

References

[1] L.K.Tan, MSc. Thesis, Department of Computing, 1993

[2] W. Bibel, Automated Theorem Proving, Viewag

[3] L. Wallen, Automated Deduction for Non-Classical Logics, MIT Press, 1992

[4] P. Baumgartner, CADE 12, 1994.

[5] M. Stickel, A Prolog Technology Theorem Prover, CADE 8, 1986

[6] D. Plaistead and S. Greenbaum. A Structure Preserving Clause Form Translation, J. Symbolic Comp. (1986).

[7] R. Shostak, Refutation Graphs, Artificial Intelligence, Vol 7, No. 1 (1976).

[8] U. Endriss, An Interactive Theorem Proving Assistant, TABLEAUX-99.

[9] M. D'Agostino, D. Gabbay, R. Hahnle, J. Possega (Eds). Handbook of Tableaux Methods, Kluwer (1999).

[10] M. Fitting, First Order Logic and Automated Theorem Proving, Springer (1996).

[11] TABLEAUXxx. Conference on Tableaux and Related Methods in LNCS from 1995 onwards.

[12] D. Sturgill, A. Segre, Nagging: A distributed adversarial search-pruning technique applied to first order inference, JAR Vol 19, No 3, (1997).