Controlling Resolution

Krysia Broda

January 14, 2003

1 Introduction

The unconstrained resolution search space for real problems is nearly always very large and almost impossible to handle in its entirety. Perhaps only a portion of it can realistically be searched, in the hope that it contains a refutation. A human solving a resolution problem would have extra tools with which to tackle the size of the search space. In the first place, one would often have a particular interpretation in mind for the predicates and function and constant symbols. In these circumstances, the human solver is able to make judgements about the "utility" of a possible resolvent. For example, one might know that a particular goal can never succeed, for it is false in the particular interpretation and hence does not logically follow from the given sentences. The "Geometry Machine of Gerlenter" followed this approach; the interpretation was a geometric diagram in that case. Or, one might know that a certain procedure was functional in some arguments. This information could be used in two ways; either to fail a goal which would violate this fact (given some other goal had succeeded with one solution for the determining arguments and the new goal would give a second solution if it succeeded); or to succeed a goal in the only way possible (again given some other goal which had succeeded and hence made the solution known for the particular determining arguments in the new goal). To transfer this adjudication to an automatic system is not so easy, since the judgements depend on the large amount of domain specific knowledge the human has. Also, it is not immediately applicable to resolution, which does not distinguish between goals and data.

Secondly, a human would (perhaps subconsciously) follow some particular line of reasoning which "looked promising". In addition, one may apply a refinement (which results in a reduction of the search space) which has proved useful in previous problems. *e.g.* goal directed reasoning - as in Prolog.

In the rest of this chapter we consider several syntactic refinements that have proved useful in resolution theorem provers. However, there are probably hundreds of different refinements, used for particular types of problems, so the selection here is just that, a selection. Syntactic refinements have been the ones most investigated. Semantic refinements using domain knowledge have not been investigated thoroughly, although there have been investigations into automating mathematical induction (*e.g.* the Boyer-Moore prover, see [Bun 83] and the various Oyster/Clam systems of Bundy et al, see [Bun 94]) and into automating correctness proofs of programs (see [PVS 95]). In this course we primarily consider syntactic improvements to the search space size, some of which may also have semantic content.

It is possible to set up a distributed proving system in which a problem can be tackled by more than one strategy. If one strategy solves the given problem then all other attempts can be terminated. This approach can also be applied to sub-problems of a problem P. In particular, suppose a problem P gives rise to a sub-problem SP and that several attempts in parallel are made to solve SP. Perhaps it can be shown by one of these that SP is not solvable, and so all attempts can be terminated as they would be fruitless. Sometimes a problem (or sub-problem) P can be shown to be not solvable, by showing that a simpler problem is unsolvable. If it is the case that when the original problem has a solution then the simpler one does too, then failure to solve the simpler one means failure to solve the original one. Paralellism can be exploited in (at least) three ways to solve a problem P. It can be used to tackle P by several strategies at once. It can be used on the sub-problems of P and applied to solve several sub-problems of P in parallel, all of which need to be solved for a solution to P; or to try several different sub-problems of P at once, a solution to any one yielding a solution to P. Obviously, in all cases the parallelism can be recursively applied to sub-problems of the sub-problems.

Such parallel systems, built out of multiple processors, are now becoming commonplace (see for example [SturSeg 97], [Wol98] and more recently in CADE [CADE] and TABLEAUX [TABLEAUX] conferences). The difficulty lies in deciding how to distribute the resources to the different methods and sub-problems. With a single processor some kind of time-slicing is used, which is quite simple.

2 The Subsumption Deletion Strategy

A resolution *search space* is a graph whose nodes are states which are sets of sentences. The *initial state* is the given set of clauses, a final state includes $\Box$. A state is connected to another by a directed arc if the proof procedure allows the transformation. In the case of unconstrained resolution, the second state could be the original clauses together with any resolvent which could be formed from them. The size of the search space can be reduced by any refinement which prevents certain resolvents from being formed from a particular state.

Searching the search space can be controlled by restricting the possible routes which can be followed through the search space. *e.g.* from a set of sentences S, two resolvents $R1$ and $R2$ may be formed in the order $R1$ then $R2$, or $R2$ then $R1$. Ideally, only one of the two orders should be taken; it would be stupid to follow both, for just the same state would be reached. Obviously, a search which visits each and every node just once is a desirable requirement, as is one which seeks out final states as soon as possible, if any can be found. Sometimes, a combination of refinement and search strategy is involved. (c.f. logic progam execution (not necessarily Prolog), in which the refinement is always to resolve with the most recent resolvent and to begin with a negative clause; the search strategy is any complete tree search.)

The simplest refinement is the so-called deletion strategy for tautologies and subsumed clauses. In this refinement, a node in the search space is labelled by a set of clauses, and may either be the root node labelled by the set of initial clauses, or be a node labelled by the clauses in its parent together with a clause resulting from a resolution step between 2 clauses in its parent and minus any subsumed clauses or tautologies. This means, in effect, that no resolution step yielding a tautology or subsumed clause is ever made, for the node corresponding to this would be identical to its parent node due to the deletion. However, it may be that a resolvent subsumes clauses in the label of the parent node and these would also be deleted. Thus no node will ever contain a tautology or a subsumed clause and such clauses would be removed from the initial set of clauses. If a set of clauses $S' = S \cup \{T\}$, where T is a tautology, is unsatisfiable, then so is S unsatisfiable. Hence, by completeness, if there *is* a resolution refutation from S', then a resolution proof from S' that doesn't use T is still possible. Similarly, for subsumed clauses, if $S' = S \cup \{D\}$, where D is subsumed by some other clause in S, then S' is unsatisfiable if and only if S unsatisfiable. So the subsumed clauses are not needed for a refutation to be made.

2.1 Saturation Search

A simple combination of search strategy with refinement is the saturation search with subsumption deletion. Recall that clause S θ-subsumes clause T if literals in $S\theta \subseteq$ literals in T. From now on, by "subsumption" we will always mean θ-subsumption. Subsumption is of two kinds: *forwards subsumption*, when the resolvent is subsumed by a clause already in S, and *backwards subsumption*, when the resolvent subsumes some clause in S. The saturation search strategy is one that, from a state S all new resolvents are formed and added to S to form the next state. Only those that have not been formed before are generated. The order of deriving resolvents is not important and only one path between S and the node containing all the resolvents is followed.

The resolvents are thus formed in "waves". Many transformations allowed in unconstrained resolution are not allowed in this regime. There may be many different ways to reach the state resulting from the first wave of resolvents. Only one of these ways needs to be followed. Thus we could think of the search space as consisting just of the states which result from each wave of resolvent generation. This may not be a very practical approach for large clause sets, for often a sequence of resolvents that are formed using the previous resolvent at each step can lead to a final state in a more direct way. However, it is one which can be further restricted by many other refinements and so is used quite often.

A resolvent that is subsumed can be removed at the end of a wave since it does not take part in any steps during the wave in which it is formed. But if a clause already present is subsumed by a resolvent it can either be removed immediately the subsuming resolvent is generated, or at the end of each wave. Immediate subsumption will often mean that other resolvents are never formed and hence the exact set of clauses present at the end of each wave could depend on the order in which the resolvents in that wave are generated.

Exercise: Show the subsumption deletion strategy is sound. *i.e.* satisfiability/unsatisfiability is preserved. Also show that it does not matter whether backwards subsumption occurs immediately or at the end of each wave of resolvent formation. In other words, any additional resolvents that were formed would eventually be subsumed. In particular, show the following Lemma for clauses D, C and A:

> if D subsumes C and $R1 = \text{ resolvent}(C, A)$,
> then, if $R2 = \text{ resolvent}(D, A)$ exists, then $R2$ subsumes $R1$,
> and if $R2$ doesn't exist, then D subsumes $R1$.

A refinement is *confluent* if from any state (here at the end of each wave), the empty clause can be derived within the strategy of the refinement. (In this case confluence = completeness as there is only one branch of the search space.) Certainly the strategy is not *strongly confluent* in the sense that all possible solutions (to an original existential goal) can necessarily be derived. See example on slides. But what about confluence?

Suppose that the empty clause can be generated without subsumption deletions. Is it possible that the empty clause could not be generated by using subsumption deletion? How might this be? Certainly, since resolution is complete, from any unsatisfiable state there is a derivation of the empty clause. *But*, can this derivation be found irrevocably, whatever subsumption deletion steps are made? The answer is yes, as long as there are no particular restrictions on the order in which clauses should be used; otherwise, some problems may arise with subsumption of clause variants

As long as unsatisfiability is maintained there are two possibilities for the state following an unsatisfiable state: either the empty clause is derived or another unsatisfiable state is derived. (An empty state is not possible as such a state would be satisfiable.) We need to be sure that unsatisfiable states cannot continue to be formed forever without ever generating the empty clause.

If S_i is an unsatisfiable set of clauses in some state, then by completeness there is some refutation R_i using S_i. The problem is that subsumption deletion might remove, or prevent from being formed, some clauses used in R_i, possibly even at the first step. As long as a different refutation R'_i, no longer than R_i, can be found, which does not require in the first step a clause deleted by subsumption, then the first step in this new refutation can be made to a new state S_{i+1}. So there will be a sequence of states and refutations, $(S_0, R'_0, n_0), \ldots, (S_i, R'_i, n_i), \ldots$, where n_i is the number of resolvents formed in R'_i, such that $n_0 > n_1 > \ldots$ and the state S_{i+1} is the first state that results from making R'_i from

S_i. (It is assumed that all subsumed clauses have been removed initially, so $R_0 = R'_0$.) The number of resolvents in R'_idecreases by at least one at each stage as at least one of the resolvents in R'_i is formed. Eventually, the empty clause must be derived.

It is left as an exercise to show that R can be transformed into R', but you will need to use the above Lemma and the fact that the refutations R_i are all finite.

This result is important, as it shows that the subsumption deletion strategy will always find a refutation if one is possible. This is called *strong completeness*. It is invariably ignored in the literature, but this is a mistake, as for some refinements the subsumption strategy *can* cause non-termination. Luckily, in the more successful refinements this is not the case.

Exercise: Find an algorithm for θ-subsumption.

A feature of refinements is that sometimes they can, and sometimes they cannot, be combined with one another. For example, tautology deletion cannot be allowed in some refinements if they are even to remain sound. Usually though, deleting tautologies at worst results in incompleteness. If the problem is to find all refutations, say the original goal was existential and all substitutions or 'answers' are required, backward subsumption can be a nuisance in that it deletes the means to a possible solution before that solution has been found. This is not a problem in traditional systems which only seek one refutation. (See example on slides). Note that forward deletion of resolvents which are subsumed (or rather the non-formation of them), is also sometimes a problem. In fact, it is often the inclusion of forward subsumption steps that prevents strong completeness. An example is given in the slides on locking.

Exercise : Consider the effects in a resolution simulation of a logic programming system of subsumption deletion, both backwards and forwards.

3 Refinements using Literal Orderings

The semantic tree argument can be used to derive several syntactic restrictions on resolution.

3.1 Ordering Predicates

The simplest restriction is: order the predicate symbols by some partial order (or, more usually, total order), and form a semantic tree such that literals using the symbol highest in the ordering comes first, the next highest second, and so on. The result will be that the lowest symbol in the ordering always appears on arcs leading to leaf nodes and in the corresponding resolution proof will therefore always be resolved upon first.

The restriction on forming resolvents, or the strategy, is then: only allow resolvents which resolve on literals which have the lowest predicate in the respective clauses. Think of the lowest predicate in the ordering as being the predicate with highest priority (that is, it is most important). The method is complete since the 'ordered' tree can be used to find a proof of the correct sort in the same way that a resolution proof was found from a semantic tree before. It is obviously sound as only resolution is involved. (This is generally the case with refinements.) Moreover, lifting any ground refutation using instances of a set of clauses S and formed according to the strategy yields a refutation of S also formed according to the strategy.

Exercise: Consider features of a set of clauses which might affect the ordering that is chosen.

3.2 Ordering Atoms

Orderings on predicates can be extended to orderings on literals. Care must be taken to ensure that the ordering is well-formed. Suppose, for example, atoms $A(a)$, $A(b)$ and $B(a)$ were ordered such that $A(b) > B(a) > A(a)$. In a ground refutation, resolution steps involving $A(a)$ or $B(a)$ would occur before steps involving $A(b)$. However, a difficulty might be encountered when actually performing the resolution search, since a resolvent between two A literals, say $A(t_1)$ and $\neg A(t_2)$ may not bind all variables in the arguments t_1 and t_2. If $B(a)$ is also a literal in one of the clauses then the step may or may not eventually be allowable depending on the final bindings of t_1 and t_2. (This problem is not restricted to monadic predicates, of course.)

Example 1

Given clauses:

1. C
2. $B(y) \vee \neg A(x)$
3. $A(a) \vee \neg C$
4. $A(x) \vee \neg A(y)$
5. $\neg A(b) \vee \neg A(a) \vee \neg B(a)$

A refutation with the ordering $B(b) > A(b) > B(a) > C > A(a)$ exists. See the semantic tree and associated proof in Figure CR1. However, when forming the resolvent between clauses (2) and (4) it is not known whether the step will eventually be acceptable; *i.e.* whether $A(x)$ in clause (4) is lower in the ordering than $A(y)$ or not. It depends on the final instances actually used. The fact that a correct (according to the rules) resolution refutation exists does not help one to find it. This is an example of the difference between ground and non-ground refutations: a refinement may easily be shown to be complete, in the sense that a ground refutation exists, and also that it lifts to a valid general refutation, but nevertheless be impractical when it comes to finding a valid refutation. Thus many paths may be searched only to find that they lead to steps that are not part of the refinement. Abandoning such a searched path (especially if it happened to lead to a final state anyway) might not be the best option. In the worst case, imagine abandoning a path that did derive the empty clause. That would be stupid! Such refinements are not completely practical.

To partially avoid the difficulty illustrated above, the following is often done. Ground atoms remain totally ordered, but non-ground atoms need not be. Thus $P(x) < Q(y)$ would only be the case if $P(x)\theta < Q(y)\theta$, for all substitutions θ. An ordering that satisfies $P(x) < Q(y)$ iff $P(x)\theta < Q(y)\theta$ for all substitutions θ is called a *stable ordering*. It is easy to show that a stable ordering based on a total ground order of atoms is a partial order. All minimal literals (of the partial order) in a clause would be available for resolution, and factoring can be applied when one of the literals is minimal. (In other words, for two atoms L and M, in case neither $L < M$ nor $M < L$, then both should be available for selection.) In this case, we could be sure that the search space will include the refutation that respects the ground ordering on atoms, but it is likely to include many others as well. Indeed, the ground refutation corresponding to the refutation actually found may even be one that does not respect the ordering on ground literals!

For the given example, one ordering is $C < A(a) < A(b) < B(a) < B(b)$. Then, for example, if there were a clause that contained $B(a)$ and $B(x)$, one can resolve on $B(a)$ first, for if x is bound to b, $B(b) > B(a)$ and if x is bound to a it doesn't matter whether $B(x)$ or $B(a)$ is used. A refutation using this ordering is:

(6)	$A(a)$	(1+3)	which subsumes (3)
(7)	$\neg A(b) \vee \neg B(a)$	(6+5)	$A(a)$ is smallest
(8)	$A(x)$	(6+4)	which subsumes (6)
(9)	$B(y)$	(2+8)	which subsumes (2)
(10)	$\neg B(a)$	(7+8)	
(11)	$\Box$	(10+9)	

In fact, it is only necessary to ensure that possible ground instances of literals in the given clauses are used in the requirement for stability.

Another option would be to place restrictions on the possible bindings to variables, in order to be sure the selection was allowed by the refinement, but this could result in a potentially larger search space due to dead ends being followed.

More details can be found in [Love 78] and in [HsiangRus 86].

3.3 Locking

Another way of applying an ordering to literals is provided by locking. Each literal in each clause is given an *index*. Only literals with lowest index in a clause may be resolved upon.

The idea of indexing literals can be used in Horn clause programming. Conditions which have more importance can be given a low index, whilst low priority conditions can be given a high index. All conclusions can be given a lowest index 1 (say). Such an approach is similar to that used in Sacerdoti's hierarchical planning system STRIPS [Sac 74]; only solutions which satisfy the low index goals are investigated further.

Example 2

$$\begin{aligned}&\textit{Can Move Block}(\text{from}, A, \text{to}, C, \text{block}, B) \text{ if}\\ &\qquad B \ \textit{is-on}\ A \text{ and } \textit{robot-hand-empty} \text{ and}\\ &\qquad C \ \textit{is-clear} \text{ and } \ldots\end{aligned}$$

Here, if *robot-hand-empty* is not true it is known that it can easily be made so, by allowing the robot to drop what it is holding. So this predicate is not so important as others. Similarly, C *is-clear* can be satisfied by clearing whatever is on C. B *is-on* A is more complex and so might be solved first. Only when a solution is found to *Can move block*$(\text{from}, A, \text{to}, C, \text{block}, B)$ if B *is-on* A, will attempts be made to solve the other subgoals.

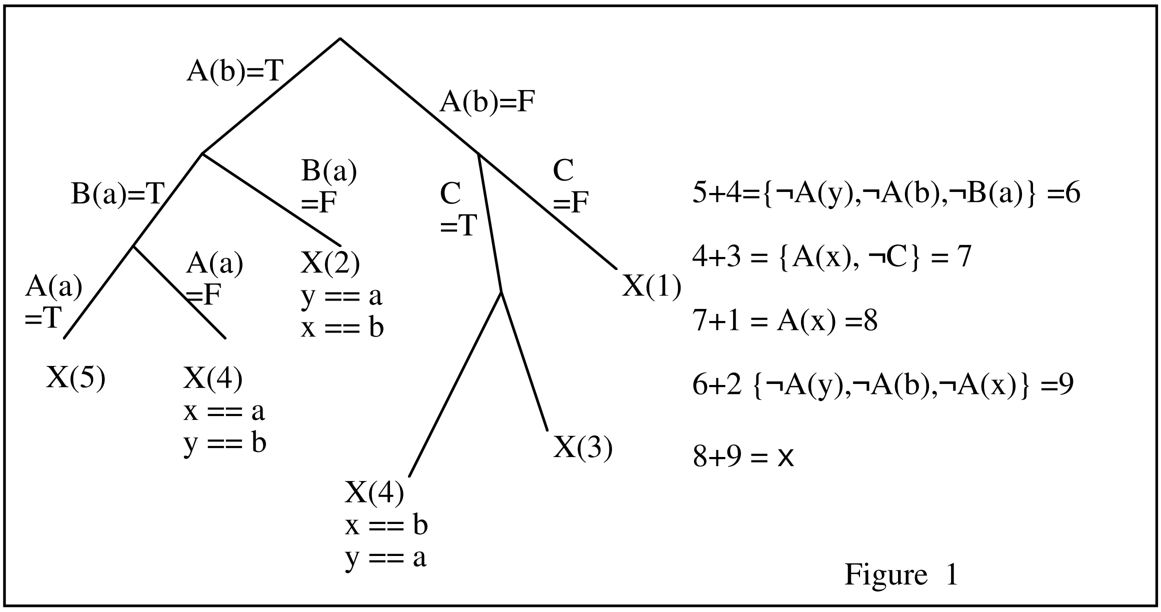
(An extension of this idea, which could be used in programming, is to make the *locks* (indices) dynamic. For example, if several literals in a query have the same lock, the locks could be updated by adding a constant to all locks in the clause in order to differentiate them. Care would have to be taken to ensure that any successful refutation is achievable (c.f. the discussion on subsumption deletion). Here again, even though unsatisfiability is maintained at each stage, that may not be enough to actually achieve the successful refutation.

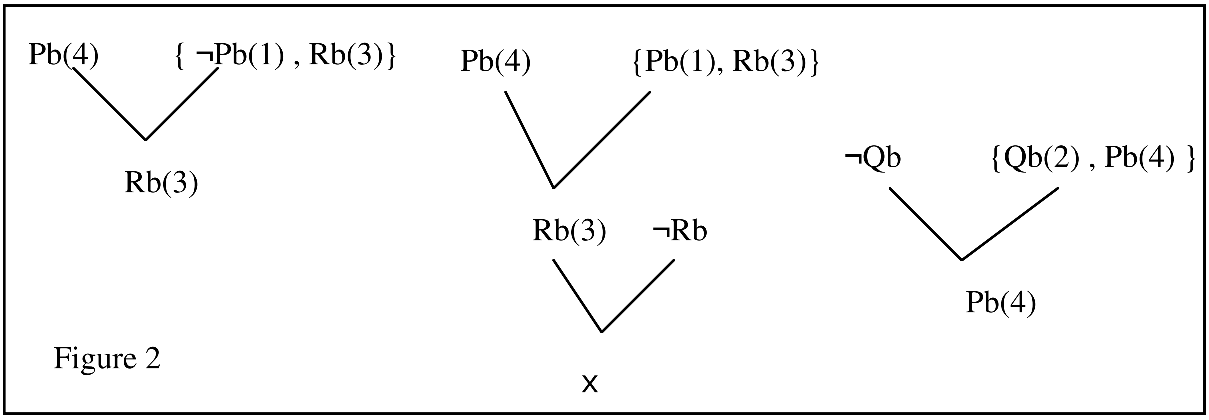
Completeness of the simple static locking cannot be proved via semantic trees, as the same ground literal may occur within several clauses but with different locks. Instead, an inductive proof can be used. It is an important technique, and so we give it here.

3.4 Completeness of Lock Resolution (ground case)

We use induction on the *excess literal count*, which is the difference between the total number of literals and the number of clauses. Thus if all clauses are unit clauses, *i.e.* have exactly one literal, the excess literal count is zero.

The proposition to prove is that, for all excess literal counts ≥ 0, if the given set of clauses S is unsatisfiable, a lock refutation P of the empty clause from S can be found. *i.e.* P is a refutation which does not violate the strategy - namely, that the only resolutions performed are those such that the literal resolved upon has lowest index in a clause. In the case of there being several identical literals in a clause with different locks assume they are merged and given the lowest of their locks.





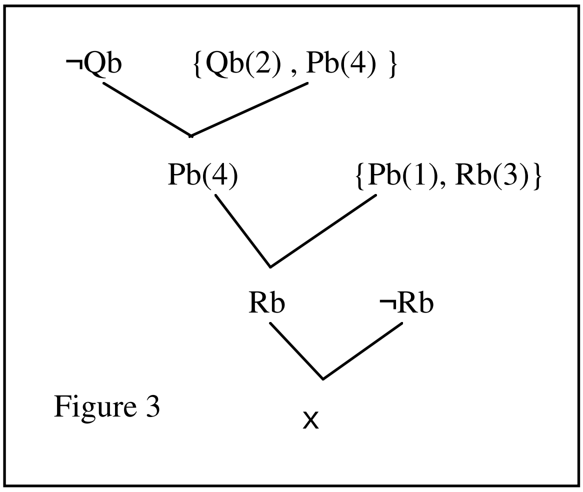


Figure 1: CR Figures 1 to 3

Base Case: If the number of excess literals is 0, all clauses are unit clauses and since they are unsatisfiable there must be a pair $X, \neg X$ which can be resolved to give $\Box$. (NOTE: A unit clause needs no lock, since its single literal will always have lowest index.)

Induction Step: Assume as inductive hypothesis that the theorem is true for all excess literal counts up to k. Suppose S has an excess literal count of $k+1$ and is unsatisfiable. (The idea is to somehow reduce the excess literal count of S, maintaining unsatisfiability, so that the hypothesis can be used.)

Suppose L_1 is one of the literals with highest lock (locks on unit clauses can be ignored) and in clause $C = L_1 \vee C'$, where $C' \neq \emptyset$. Consider the sets (i) $= (S - C) \cup \{L_1\}$ and (ii) $= (S - C) \cup C'$.

We claim both of (i) and (ii) must be unsatisfiable if S is unsatisfiable and clearly both have $\leq k$ excess literals.

Exercise: Show this. [Hint: Suppose (i) were satisfiable with model M. M would make L_1 and hence C true.]

Therefore, by the induction hypothesis there are deductions of $\Box$ using the lock strategy from (i) and (ii). Add L_1 back into deduction (ii). That is, wherever C' appeared, use $C' \vee L_1$. A lock deduction of either L_1 or $\Box$ now exists (the latter in case L_1 merged with a literal of lower index or the clause was not used). In the second case, we are done. In the first, continue with the deduction (i) to deduce $\Box$.

As an example, we use the proof method to construct a lock resolution proof for $S = \{Pb_4 \vee Qb_2, \neg Pb_1 \vee Rb_3, \neg Qb, \neg Rb\}$. (The numbers are the locks.)

L_1 is Pb_4, $C = Pb_4 \vee Qb_2$, $C' = Qb_2$.
We have

(i) $\{Pb_4, \neg Pb_1 \vee Rb_3, \neg Qb, \neg Rb\}$ (ii) $\{\neg Pb_1 \vee Rb_3, \neg Qb, \neg Rb, Qb_2\}$

We can again use the construction for (i) and (ii). In fact from (ii) we can immediately deduce $\Box$, by resolving on Qb. In (i), L_1 is Rb_3, giving another (i) and (ii) (let's call them (iii) and (iv)) as

(iii) $\{Pb_4, Rb_3, \neg Qb, \neg Rb\}$. (iv) $\{Pb_4, \neg Pb_1, \neg Qb, \neg Rb\}$

Both of these have trivial deductions of $\Box$. Working backwards now, we replace Rb_3 in proof (iv) giving the first tree diagram in Figure CR2, and then we must apply proof (iii), giving the second diagram (called (A) for future reference).

We work back to the previous step, where $L_1 = Pb_4$, and replace Pb_4 in (ii) giving the third diagram and then continue from Pb_4 using (A). The final proof is in Figure CR3 and you can see it follows the strategy.

Locking was investigated and invented by Boyer [Boyer 73]. Soundness of this proof procedure is obvious. Why?

The completeness proof of locking is only given here for ground clauses and the fact that the proof can be lifted is left as an exercise. (It is not such an easy one, for some care needs to be taken when dealing with factoring.) Of course, to be sure of finding a deduction of $\Box$, still the whole search space may need to be searched, although it is a much smaller search space now.

Locking can be used to simulate other proof procedures, as we shall see.

Exercise : Explain how locks can be added to literals in Horn clauses so the locking strategy will simulate, using resolution, logic programming execution for Horn clauses.

Note that, whereas in unconstrained resolution tautologies could always safely be deleted, without loss of completeness, this is not always the case with locking. (See example on slides, which also shows there is a problem with forward subsumption.)

3.5 Factoring

In all refinements based on orderings between literals, any restriction on factoring should be defined. That is, when is a factoring step allowed? We know that, in general, factoring is necessary whatever other restrictions (or none) are in force, so it should not be ignored. In locking, the simplest solution is that a factor operation must involve at least one literal with lowest index, for neither of the literals involved is able to contribute to a refutation until it has the lowest lock in a clause. In the predicate ordering case factoring can be restricted such that one of the literals involved has the lowest predicate symbol, or, in atom ordering, that one literal is lowest in the ordering. However, performing factoring at other times can still be useful if it leads to subsumption.

It is often the case that if factoring is combined with the binary resolution step it leads to simpler completeness proofs. However, in a practical theorem prover, factoring can be a useful separate operation even though it may lead to additional clauses being formed. For these new clauses will be simpler than their parent and may subsume other clauses. Using them can often lead to simpler refutations being found than would otherwise be discovered. Certainly, *some* factoring can sometimes be necessary for the derivation of the empty clause, although it can always be restricted to safe factoring: a clause C can be *safely factored* to F if $F \equiv C$.[1] Hence C can be replaced by F. Thus safe factoring does not create additional clauses, although disallowing other factoring may require more steps (and so a larger search space) to find a refutation. So no, or little, factoring is almost as much of a problem as unrestricted factoring.

Some refinements have an advantage in that they force factoring in a painless way when it is needed, but do not require it otherwise. It is not easy otherwise to enforce it in useful but

[1]Since F is a factor of C, $C \models F$, so $F \equiv C$ if $F \models C$ or F subsumes C.

unnecessary cases. For example, in logic programming, in the query $?A(x,a), A(b,y)$ (say), then it is possible to factor the two literals to obtain the query $?A(b,a)$. But if $A(x,a)$ and $A(b,y)$ only succeeded with other bindings to x and y, so $?A(b,a)$ did not succeed, the factoring would be a waste of time. Of course the factoring might also cause solutions to be lost! There are other similar situations which can be analysed. It is for this reason that factoring in LP is not usually attempted. Luckily, for Horn clauses, factoring is *not* necessary – the need for factoring only arises for a set of sentences that includes non-Horn clauses. However, in practice, even in LP, it might be worthwhile to factor identical literals.

Exercise: (Hard) Show that a refutation R that uses unsafe factoring can be transformed to another, R', that uses only safe factoring. To do this, show that if F is an ordinary factor of C and resolves with D on a factored literal giving $R1 \vee R2$, then C can be resolved with D twice (once for each of the factored literals) to give a resolvent $R1' \vee R2' \vee R2''$, where $R2'$ and $R2''$ arise from the two uses of D and corresponding pairs of literals in $R2'$ and $R2''$ factor safely to eventually yield $R1 \vee R2$.

Another simple search strategy which can often be combined with various refinements is the *unit preference* strategy. States which have resulted from a resolution between a unit clause and another are investigated before others. This strategy has received much attention in the literature, probably because of its obvious simplicity; resolving with a unit clause yields a resolvent with fewer literals than in the other parent clause. In special cases the resolvent may subsume the parent altogether and so replace it.

4 Hyper-resolution

An alternative way to restrict the number of possible resolvents is to make each step larger. *Hyper-resolution* tries to do this.

The strategy is a generalisation of bottom-up deduction, or forwards derivation from facts to produce further facts. When applied to Horn clauses this is exactly the form it takes. A hyper-resolvent is formed from a *nucleus*, consisting of at least one negative and none or more positive literals, and *electrons* which only have positive literals. Each negative nucleus literal is resolved with a literal in some electron, possibly with factoring, such that the unifiers are compatible with each other. *i.e.* the unifiers can be successfully combined.

Suppose the nucleus is $\neg L_1' \vee \neg L_2' \vee \ldots \vee \neg L_m' \vee A$, and the electrons are $E_i = L_i \vee B_i$, where A, B_i are positive clauses. The resolvent $(A \vee B_1 \vee \ldots \vee B_m)\theta$ is called a hyper-resolvent, where θ is the mgu of $\{L_i', L_i\}$. Note that the E_i clauses should be variable disjoint.

In practice, the resolvent is formed step-wise, using one electron at a time. The intermediate resolvents are not kept.

4.1 Completeness of Hyper-resolution

A completeness proof of hyper-resolution can be found either by employing semantic trees (as here) or, since locking can be used to simulate hyper-resolution, the induction proof used for locking can be adapted. (An exercise!)

Recall that a semantic tree is an enumeration of all Herbrand Interpretations for a set of clauses S. There is no restriction on the number of times an interpretation may appear in the tree, as long as every interpretation appears in some branch or other. We can form a closed semantic tree for (unsatisfiable) S as follows: first form a traditional closed semantic tree; let $L = \{\text{atoms in this tree}\}$. Then start a new tree, with $n+1$ ($n = \#L$) arcs emanating from the top node. Each arc except one is labelled by a different literal, namely $\neg X$ where $X \in L$. The first arc is labelled by all atoms in L. The formula

$$(X_1 \wedge X_2 \wedge \ldots \wedge X_n) \vee \neg X_1 \vee \neg X_2 \vee \ldots \vee \neg X_n$$

is a tautology, as required for a semantic tree. For those branches which cannot be closed, extend the node at their tips with $\#L$ arcs; each is labelled as before, using the set $L - X$, where the atom used in the previous arc in the branch is omitted. This procedure is continued, each time omitting those literals already labelling an arc in the branch and reducing the number of arcs by 1, until the tree closes. Note that the branch including the first arc in each case is (or includes) one of the failed interpretations in the original tree. This tree for S will eventually run through all the atoms in L. It will have potentially enumerated every HI over the language L; S is false under every one of these interpretations, for otherwise the original tree would have had an open branch corresponding to that assignment which did not falsify S. If it did not so close, then the open branches would have to be continued in some way to include other potentially falsifying interpretations. But we know there are none.

Example 3

Given clauses $S = \neg Ha$, $\neg Gx \vee \neg Fb$, $Fx \vee \neg Hb$, $Gx \vee \neg Fx$, $Fx \vee Hx$.

Part of a hyper-resolution semantic tree is given in Figure CR5. (Not all branches are completed, but they could be so, at great length.) The structure of the sub-tree formed at the bottom of a branch in a tree formed in this way will have the form shown in Figure CR4, where $L = L_1, L_2, \ldots, L_n, \ldots, L_m$. Now, the clause falsifying the leftmost branch of such a sub-tree must have one or more of $L_{n+1}, \ldots, L_m$ as a negative literal else there would be no need for the branch. Also, the clause instances falsifying the other branches must include each of $L_{n+1}, \ldots, L_m$, respectively, as occurrences. These clauses become the electrons of the ground hyper-resolvent (and will all be composed of positive literals). Consequently, it is possible to form a ground hyper-resolvent between all of these clauses, which falsifies $\neg L_1, \ldots, \neg L_n$, and will therefore consist only of positive atoms. Not all the electron clauses need to be used. In general they will not be. As in the argument for

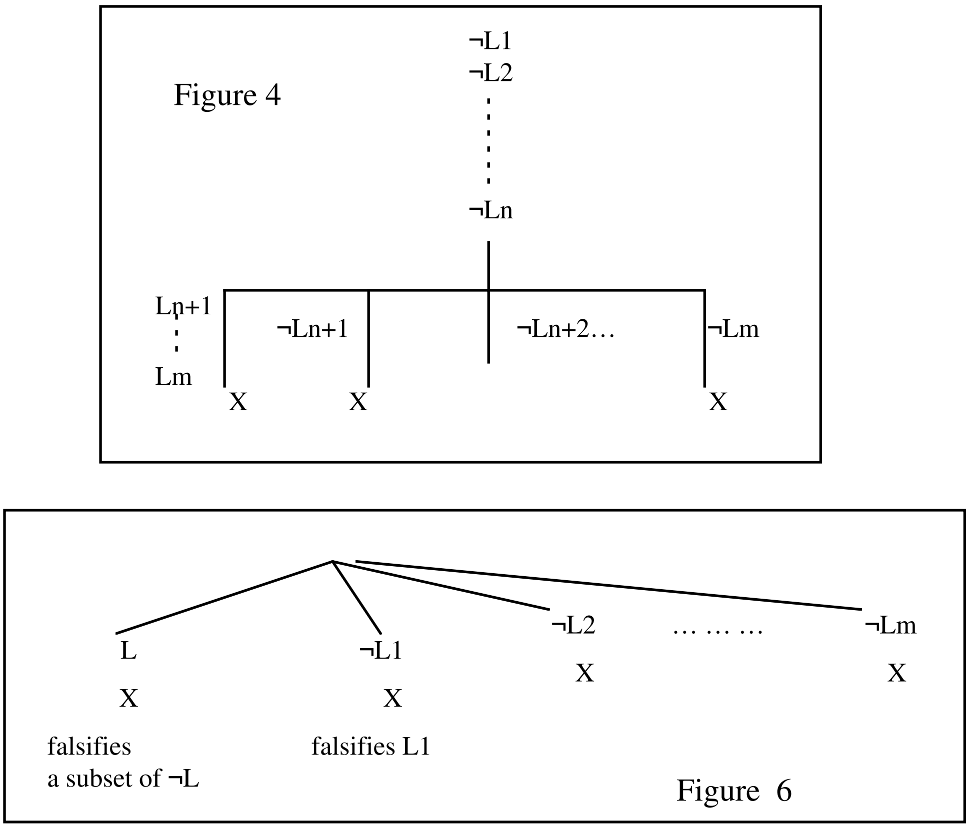


Figure 2: CR Figures 4 and 6

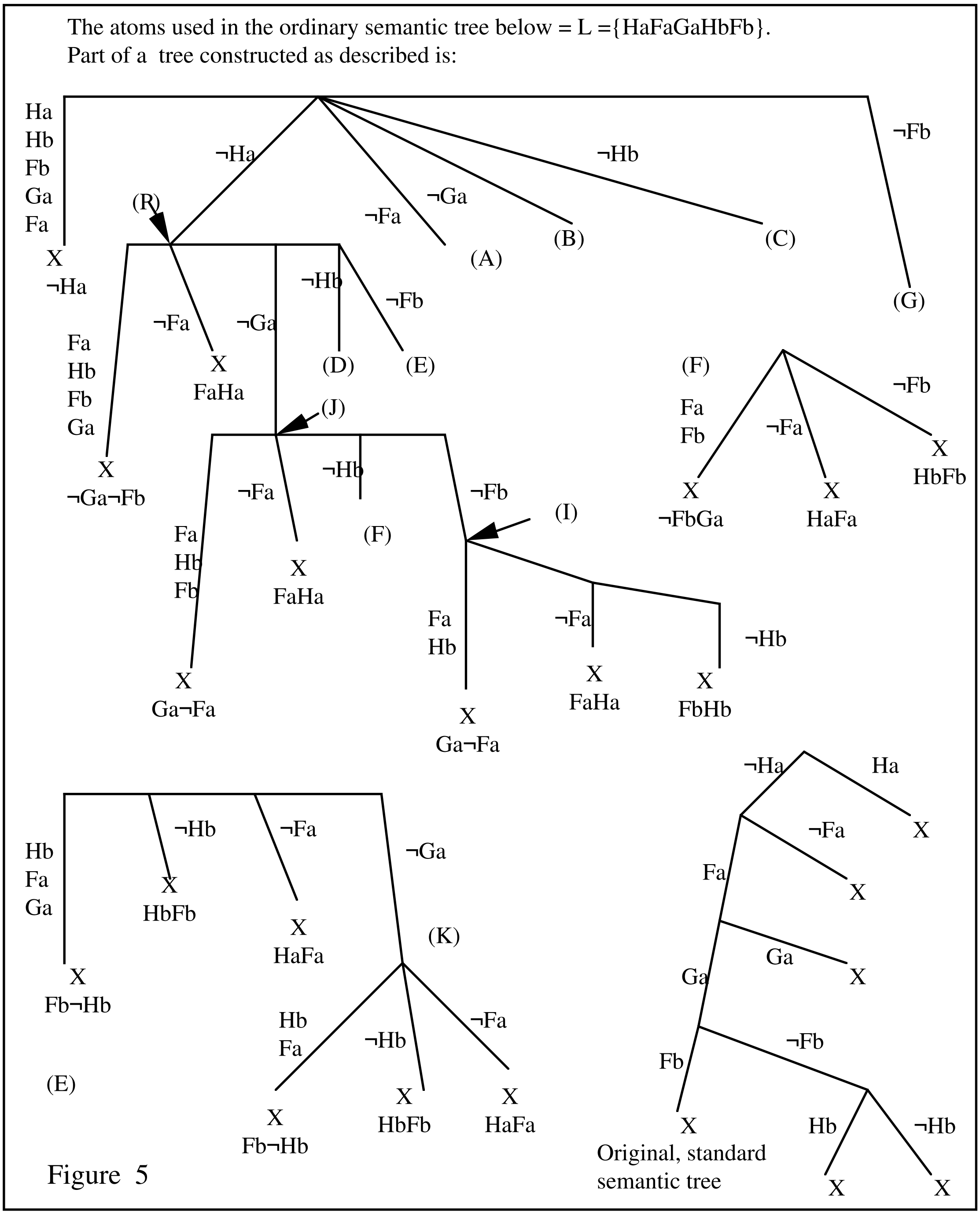


Figure 3: CR Figure 5

resolution completeness, we can continue working up the tree in this way, reducing its size as we go. Eventually, we must reach the situation where there are no literals before the split – see Figure CR6. (Note, a bit confusingly, that in that figure L is the set of literals $\{L_i\}$, whereas each L_i is a literal.

Exercise : show that this situation must occur.

Again, a final hyper-resolvent can be formed, giving $\Box$. Sometimes, as in resolution, a resolvent will falsify a node higher in the tree than the one just above the leaves involved. The resulting proof can be lifted just as for ordinary resolution.

As an example, consider the section of the tree in Figure CR5 involving the nodes labelled by (E) and (F) (given as separate sub-trees). Node (F) can be labelled by the hyper-resolvent $GaHb$. ($HaFa$ was not used).) The node next to (F) is (I) and it can be labelled by $GaHa$, which falsifies node (J). Similarly, (K) can be labelled by Fb, which falsifies (E). Moreover, (P) can be labelled, by resolving $\neg Ga \vee \neg Fb$ with $GaHa$ (J), and Fb (E), to obtain Ha. Finally, Ha can be resolved with $\neg Ha$ to obtain $\Box$.

Hyper-resolution can be generalised in an interesting way. Notice that all nucleus clauses are true and all electron clauses are false in the interpretation I in which all atoms are false. The original set of clauses S can be viewed as being split into two sets, such that one set is true and the other set is false in the interpretation and resolution can only occur between clauses which come from different sets. Almost any interpretation can be chosen to divide an arbitrary S into two sets and has been called a *setting* by Loveland[2]. So that all instances of a clause end up in the same set according to the setting, interpretations are usually chosen so that if L is true in the interpretation, so are all its instances. Otherwise, one instance of a clause may be a nucleus and another an electron, or in one instance of a nucleus it may be literals A and B that contribute to its being true, whereas in another instance it may be some other literals. Such interpretations are called *uniform*. To find uniform interpretations, if a literal L is desired true, then so (at least) must the generalisations of L, relative to the set of unsatisfiable clauses S, be true. An example of a generalisation of the set of ground literals $\{P(a), P(b)\}$, is $\{P(x)\}$. Other examples of generalisation are $\{P(a,a), P(b,b)\}$ generalising to $\{P(x,x)\}$ and $\{P(a,a), P(b,c)\}$ generalising to $\{P(x,y)\}$. In the first case, if $P(x)$ did not occur in S then there is no problem in interpreting $P(a)$ and $P(b)$ differently. But if $P(x)$ did occur in some clause in S, then both $P(a)$ and $P(b)$ (and any other instances of $P(x)$) must be interpreted in the same way for a uniform interpretation. As a further example, if S contained only the P literals $P(x,a)$, $P(x,b)$, and $P(c,a)$ were made true by the interpretation, then only the generalisation $P(x,a)$ need also to be true. *i.e.* only those literals in S of which $P(c,a)$ is an instance need also to be true. If $P(c,y)$ were also in S then both $P(x,a)$, $P(c,y)$ would need to be true for all x and y. This in turn would force $P(x,b)$ to be true. Only literals in S are considered as candidates for generalisations.

[2] Can you think of a simple set of satisfiable ground clauses S and interpretation I, such that I makes every clause in S false? (If S is unsatisfiable, then indeed, *any* I will split S into two non-empty sets. Why?)

Summarising, I is a uniform interpretation for S iff for each literal L in S, I assigns the same truth value to all instances of L. The same strategy as before is applied: only clauses which are true in the chosen interpretation may act as nucleus, and only clauses false in the interpretation can act as electrons.

If the chosen interpretation makes (positive) atoms all true and all clauses are Horn, then hyper-resolution can be made to mimic 'top-down' resolution, as in logic programming. This is dual to the original definition which mimics bottom up resolution for Horn clauses.

(**Exercise**: Check this).

Generally, when given a problem of the form $DB \models \mathit{Goal}$, the database DB is satisfiable. Thus when the problem is converted to clauses, $\mathit{conversion}(DB)$ is satisfiable, but $\mathit{conversion}(DB) \cup \mathit{conversion}(\neg G)$ is not. If we choose as a setting some model of $\mathit{conversion}(DB)$, the hyper-resolution strategy will prevent resolution between clauses in the database. This is usually a good strategy, since many inferences may be drawn from the database and most will not be relevant to the proof. On the other hand, any refutation must use some clauses from $\mathit{conversion}(\neg G)$. When the setting is chosen as above the method is called a *set-of-support* restriction, as resolvents can only be formed between clauses of which one must be from $\mathit{conversion}(\neg G)$ or derived from it. In this case electrons are the false clauses and initially come from $\mathit{conversion}(\neg G)$. The set-of-support strategy is so useful, it has become a strategy in its own right, even when hyper-resolution is not in force. Indeed, it may not be possible to find a suitable setting which makes *exactly* the DB clauses true and the goal clauses false. Usually, some of the clauses derived from the goal sentence will be true also. But still, no two clauses from the converted database can be resolved together.

Example 4

Given clauses $PbQb$, $\neg PbRb$, $\neg QbRb$, and required to show Rb.
An interpretation which makes the given clauses true and the negation of the conclusion false is $\{Rb, \neg Qb, Pb\}$. Every allowable hyper-resolvent is false in this model and occurs between sentences of which one is either the negation of the goal or derived from it.

(**Exercise:** Show this is the case.)

Hyper-resolution can be restricted still further by allowing literals in the electrons to be ordered in some way. For details see Loveland's book ([Love 78]). Hyper-resolution is usually combined with a saturation search strategy and can also be combined with subsumption deletion. Tautologies are never derived in the usual setting, and need not be derived anyway. Completeness of the generalised version can be reduced to the completeness of the standard version of hyper-resolution (see slides).

Hyper-resolution has proved to be a very practical theorem proving strategy and forms the basis of the OTTER Theorem Prover, [Wos 94]. Details of how to use OTTER will be given in the lectures.

In a database application, inconsistency may be used to show that a database (perhaps after an update) violates an integrity constraint. Assume an Integrity constraint (IC) is rewritten as $\neg(IC_1)$, where IC_1 is a conjunction of literals. Then the DB violates IC if $DB \models IC_1$ (i.e. DB, IC is inconsistent) and does not do so otherwise. For example, for the integrity constraint $A \rightarrow B$, IC_1 would be $A \wedge \neg B$. In case the integrity constraint has negated literals and the DB is a logic program, some sort of completion of DB would implictly be used in the computation. Before updating, presumably a database is consistent with its integrity constraints. Therefore, if it is inconsistent afterwards, that must be due in some way to the update. If the setting is chosen so that the update is false in it (can this always be done?) and the database and constraints are true, the set of support strategy will lead to inconsistency, if it exists, in an efficient way. This method is worth investigation – (c.f. [KowSad 93]).

Various other clever tricks are possible. For example, often a setting can be chosen that results in clauses that mimic Horn clauses. The famous steamroller problem [Cohn 87] succumbs to this treatment. (See in solutions to problems.)

References

[Boyer 73] R. Boyer, PhD Theis, University of Austin, Texas, (1973)

[Bun 83] A. Bundy, The Computer Modelling of Mathematical Reasoning, Academic Press (1983)

[Bun 94] A. Bundy et al, various CADE 1990, 1992, 1994

[CADE] CADE Conferences. See Springer Lecture Notes in Computer Science

[Cohn 87] A. Cohn, The Steamroller Problem, in A More Expressive Formulation of Many Sorted Logic, JAR (1987)

[HsiangRus 86] J. Hsiang and Rusinowitch, CADE 8 (1986).

[KowSad 93] R. Kowalski and F. Sadri, Integrity checking in deductive databases, Tech Report, IC

[Love 78] D. Loveland, Automated Theorem Proving: A Logical Basis, (1978)

[PVS 95] A Tutorial Introduction to PVS, WIFT 95, Rushby et al, SRI, (1995)

[Sac 74] E. D. Sacerdoti, Planning in a hierarchy of abstraction spaces, in Artificial Intelligence, Vol. 5, No. 2 (1974)

[SturSeg 97] D. Sturgil and A. Segre, Nagging: A Distributed Adversarial Search-Pruning Technique applied to first order Inference, in JAR, Vol 19, No. 3, (1997)

[TABLEAUX] TABLEAUX Conferences, Springer Lecture Notes in Computer Science

[Wos 94] The OTTER Theorem Prover (Manual)

[Wol98] A. Wolf, p-SETHEO: Strategy Parallelism in Automated Theorem Proving, TABLEAUX99, LNAI 1397.