AUTOMATED REASONING

SLIDES 6:

CONTROLLING RESOLUTION
Simple Restrictions revisited:
Subsumption
Tautology removal
Factoring
Saturation Search improved
Refinements and Search Spaces

KB - AR - 13

Controlling Resolution: 6ai

In this group of slides we'll look at some basic ways to control resolution. It is easy to make resolution steps, but for even a medium sized problem the number of resolvents increases rapidly, so some method is needed to control their generation.

In Slides 3 we introduced saturation search and considered factoring. Here we'll look in more detail at subsumption and its relation to factoring. In Slides 7 and 8 we'll consider other ways of restricting the resolvents.

A number of "difficulties" for theorem provers have been presented by Wos and are summarised in the optional material for these slides. Larry Wos led the group at Argonne that produced Otter – a wonderful theorem prover that you will use soon. The successor is Prover9, but Otter is easier for beginners. This prover uses a saturation search as its basic strategy, but with many additional ways of restricting resolvents. The important thing is that the strategy is systematic.

You already saw that unrestricted resolution generates many redundant clauses. There are some very simple restrictions that are almost universally adopted in theorem provers, called *Tautology deletion*, *safe factoring* and *subsumption*. We consider these next.

Subsumption 6bi

Let C and D be clauses:
C ***θ–subsumes*** D if Cθ ⊆ D for some θ

C ***subsumes*** D if ∀C |= ∀D, where ∀C means that all variables in C are explicitly unversally quantified. Equivalently,
C **subsumes** D if {C+¬D} has no H-models (or if C+¬D==>* [ ]).

C **strictly θ–subsumes** D if C θ–subsumes D without necessary factoring in Cθ. Each literal in Cθ matches a different literal in D.

Example: Px ∨ Qx subsumes Pa ∨ Qa (and θsubsumes it)
Pxa ∨ Pyx θ–subsumes Paa but not strictly
Pf(x) ∨ ¬Px subsumes Pf(f(y)) ∨ ¬ Py but does not θ–subsume it.

Relation between θsubsumption and full subsumption:

θ–subsumption implies subsumption
(but not the converse - find a counter example involving a recursive clause).
(Usually checks are made for strict θ–subsumption only.)

Subsumption Practice (ppt) 6bii

Exercise

Does first clause (strictly) θ–subsume the second?

1. Qxx ∨ Qxy ∨ Qyz and Qaa ∨ Qbb
2. Qax ∨ ¬Rxa and Qab ∨ Qac
3. Qzy and Quv
4. Qxx and Quv
5. Sf(x)x and Sug(u))
6. Sf(x)y and Sug(u))
7. Quv and Qxx
8. Px ∨ Py and Pa
9. Qxx ∨ Qyx and Qzb

Note: **Identical** literals in a clause are always merged, so Pa ∨ Pa is always Pa and Px ∨ Px is always Px. They both strictly subsume Pa ∨ Qa.

If C θ–subsumes D, but not strictly,
can first factor C to C' where C' strictly θ–subsumes D.

Subsumption in Use

There are two *species* of subsumption:

Forward subsumption: a resolvent is subsumed (so no need to generate it).
Backward subsumption: a resolvent subsumes (can remove other clauses).

In a saturation search

- forward subsumption can be applied either:
(i) as soon as a subsumed resolvent is generated, or (ii) after each stage
- backwards subsumed clauses can be removed either:
(i) when a subsuming resolvent is generated, or (ii) at the end of each stage.

Exercise (from slides 3): (Stage 0 and Stage 1 (no subsumption yet)
1. Dca ∨ Dcb   2. ¬Dxy ∨ Cxy   3. ¬Tx ∨ ¬Cxb   4. Tc   5. ¬Dcz
6. (1,2) Cca ∨ Dcb   7. (1,2) Dca ∨ Ccb   8. (1,5) Dcb
9. (2,3) ¬Dxb ∨ ¬Tx   10. (3,4) ¬Ccb   11. (1,5) Dca

Compare:
a) using forwards subsumption:
removal of a subsumed clause immediately or at the end of a stage
b) using backwards subsumption:
removal of a subsumed clause immediately or at the end of a stage
What would you recommend as a good subsumption strategy?

An Important Property of Subsumption:

Subsumed clauses can be removed from S without affecting satisfiability:

If C, D in S and C subsumes D, then S is unsatisfiable iff S-{D} is unsatisfiable
Hence S ⇒* [] iff S- {D} ⇒* []
(Proof is in ppt)

Observation: the sets of derivations using S and using S-{D} are not the same
eg Backwards subsumption can mean some "proofs" are lost.

Example
1. Pxy ∨ ¬Qx ∨ ¬Ry   2. ¬Puv   3. Qa   4. Rc   5. Qb

6. (1,2) ¬Qx ∨ ¬Ry causes 1 to be removed
7. (6,3) ¬Ry causes 6 to be removed   8. (4,7) []

There are no other proofs even if 8 is not used to remove all other clauses.
Without backward subsumption there is another derivation using (6) and (4).

A Constructive View of θ–Subsumption Deletion

(Assume for simplicity that factoring is unnecessary for this slide and the next)

Using θ–subsumed clauses leads to redundancy in proof construction in 2 ways.

If C θ–subsumes D, C≠D, and D resolves with E giving R1 then
either (i) C θ–subsumes R1
or (ii) C resolves with E to give R2 that θ–subsumes R1

e.g. Let C = Px, and D=Pa ∨ Q; then C θ–subsumes D.

Suppose D is resolved with ¬Q ∨ R (ie not on the subsumed literal)
the resolvent (R1) is Pa ∨ R, which is θ–subsumed by Px (i.e. by C).

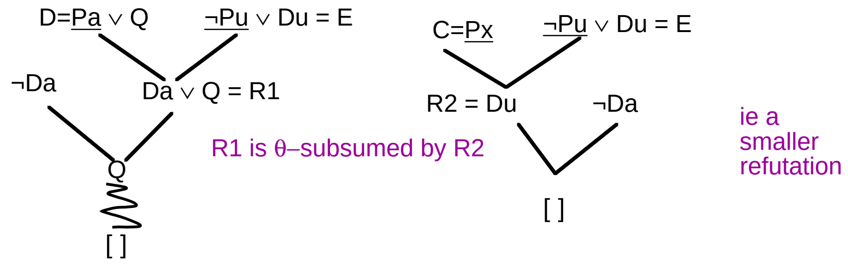
C=Px θ–subsumes D = Pa ∨ Q   ¬Q ∨ R   ie generates a redundant clause

R1 = Pa ∨ R is θ–subsumed by C=Px

Using D in this case simply leads to further redundant θ–subsumed clauses.
This is an example of the first case - resolve on a non-subsumed literal

If C θ–subsumes D, C≠D, and D resolves with E giving R1 then
either (i) C θ–subsumes R1,
or (ii) C resolves with E to give R2 that subsumes R1

Let C = Px and D = Pa ∨ Q and suppose D is resolved with E = ¬Pu ∨ Du
The resolvent R1 = Da ∨ Q is θ–subsumed by R2 = Du
(the resolvent of C with ¬Pu ∨ Du)



Using D like this yields clauses that can be θ–subsumed if C is used instead.
This is an example of the second case – resolve on a subsumed literal in D

If C θ–subsumes D, then using C instead of D gives a shorter refutation.

The subfree Property 6bvii

As illustrated on Slides 6bv/vi, using subsumed clauses leads to redundancy in a proof.

It can be shown that the following *Property SubFree* holds for refutations formed using saturation search. (See Appendix1 for the proof.)

Property SubFree:
Let S be a set of unsatisfiable clauses. Then, there is a refutation R from S such that for each clause Ck at depth k≥0 and used in R, Ck is not subsumed by any different clause that is in S or derived from S at a depth ≤k.

In other words, no resolvent in the refutation R is subsumed by a clause in S or by a previously generated clause

The proof of Property SubFree uses this fact (illustrated on slides 6biv/bv):

if C subsumes D and a step in a refutation uses D (resolving with K) to derive R,
then either C subsumes R,
or resolving C and K leads to resolvent R' that subsumes R.

The proof of this fact is not difficult and is left as an exercise.

All about Tautologies 6ci

A clause is a ***tautology*** if all its instances contain an atom and its negation.

Important Property of Tautologies

If T is a tautology and T is in S, then S is satisfiable iff S-T is satisfiable.
i.e. T can be removed S.

Important: ¬ Qx ∨ Qy is not a tautology but ¬ Qx ∨ Qx is.

Resolving a tautology T with S (on one of the tautolgous literals) leads to a resolvent R that is subsumed by S:

e.g. if T=¬Qa ∨ Ra ∨ Qa, S=Px ∨ Qx, then R=Pa ∨ Ra ∨ Qa is ***subsumed*** by S

Two Examples:

Px ∨ Qx
¬Qa ∨ Ra ∨ Qa
a tautology
Pa ∨ Ra ∨ Qa

Pa ∨ Qa
¬Qx ∨ Ra ∨ Qx
a tautology
Pa ∨ Ra ∨ Qa

Question: Does Prolog have to worry about tautologies? Explain.

Factoring again (ppt) 6cii

F is a basic **factor** of E1 ∨... ∨En ∨H if F=(E ∨ H)θ,
where θ=mgu{Ei} and E=Eiθ, H is a clause, Ei are literals.

Examples
Px ∨ Py factors ==> Px
Qua ∨ Qvv factors ==> Qaa
Qxy ∨ Qaz ∨ Rxy factors ==> Qaz ∨ Raz

Factoring is not easy to implement efficiently, but sometimes it **is** necessary.

EG you cannot refute {Px ∨ Py , ¬Pa ∨ ¬Pb} without factoring.

The above definition of a basic factor concentrates on one predicate symbol at a time. When applying θ it could be that other literals become identical and can be merged.

eg factor Px and Py in Qxx ∨ Qxy ∨ Px ∨ Py ==> Qxx ∨ Px
factor Qxx and Qay in Qxx ∨ Qay ∨ Qxy ==> Qaa

Or, a factor can be further factored:
eg factor Px,Py in Qxx ∨Qxy ∨Px ∨Py ∨Pa ==> Qxx ∨Px ∨Pa
which can be further factored ==> Qaa ∨ Pa

We use "factor" to mean either a basic factor,
or the result of several steps of basic factoring

Further Properties of subsumption and factoring. 6ciii

Full subsumption is not usually checked as it can be a theorem proving problem itself that may not terminate. E.g. ¬ P(x)∨P(f(x)) subsumes ¬P(x)∨P(f(f(x))) (i.e. the first clause implies the second) but it does not θ–subsume it. Even checking for θ–subsumption can be hard. e.g. if C is a clause with (say) 5 literals, all positive and all of predicate P, and D is a similar kind of clause, there are many possible ways in which C might subsume D? (How many?) One simple way to check for θ–subsumption is given in the exercise solutions.

Checking for factoring is also not so easy. Moreover, if *every* factor of a clause is added then the number of clauses can increase very quickly. But factors *are* often useful, especially if they instantiate a clause. The factored clause might resolve with fewer clauses than the original, so fewer resolvents are then considered. One strategy might be to favour factored clauses when forming resolvents. However, the original clause cannot normally be discarded. Factors are also sometimes necessary – see Slide 3ci for an example.

A factor of C is implied by C (**Why?**). If also the factor subsumes C, then it implies C and hence C and the factor are equivalent. We call this *safe factoring* (or *reduction*). In this case C *can* be discarded. Finding safe-factors is worthwhile – the factor is smaller than C as at least two literals have been made identical. Moreover, it has been shown that these are the only kinds of factors that might be necessary in order to find a refutation, although smaller refutations might be found if other factors may be generated and used.

(By the way, note that a refutation is sometimes called a proof, since it is a proof of a contradiction, the empty clause.)

Safe Factoring

6civ

Exercise: Write down **all** the factors of Qxx ∨Qxy ∨Px ∨Py ∨Pa

F is a **safe factor** (or a *reduction*) of C if F is a factor of C <u>and</u> F subsumes C

Exercise: Which factors of Qxx ∨Qxy ∨Px ∨Py ∨Pa are safe factors?

If F is a safe factor of D then F is equivalent to D and can replace D:

Why is this?
If F subsumes D, then F $\models$ D (by definition)
If F is a factor of D then D $\models$ F. (Why?)
Then F ≡ D and F can replace D.

If F is a factor of D but does **not** subsume D, then F *cannot* replace D.

e.g. Qaa doesn't subsume Qua ∨ Qvv; the latter might be needed with some other substitutions for u and v (e.g. Qba ∨ Qbb) so it cannot be discarded.

In fact, only **safe factoring** (or *reduction*) is necessary (not proved here)

Safe Factoring (continued)

6cv

F is a **safe factor** (or a *reduction*) of C if F is a factor of C <u>and</u> F subsumes C

The (only) safe factor of <u>Qxx ∨Qxy ∨Px ∨Py ∨Pa</u> is Qxx ∨ Px ∨ Pa

Another characterisation of safe factor:

C safely factors to F iff for some G and H,
C=G∨H, <u>where G is a disjunction of ≥2 literals which will factor</u>
<u>and H is a disjunction of ≥1 literals,</u>
and F = (G∨H)θ = Gθ ∨ H =H for some θ
(i.e. θ doesn't affect H and Gθ merges with H)

Exercise: Show H <u>is</u> then a safe factor of G∨H (not so easy)

Questions:
(i) Identify G and H in Qxx ∨Qxy ∨Px ∨Py ∨Pa and its safe factor Qxx ∨Px ∨Pa

(ii) Qaa is not a safe factor of Qxx ∨Qxy ∨Px ∨Py ∨Pa.
Show that there is no division of C into G and H to satisfy the above criterion.

Search Spaces and Refinements

6di

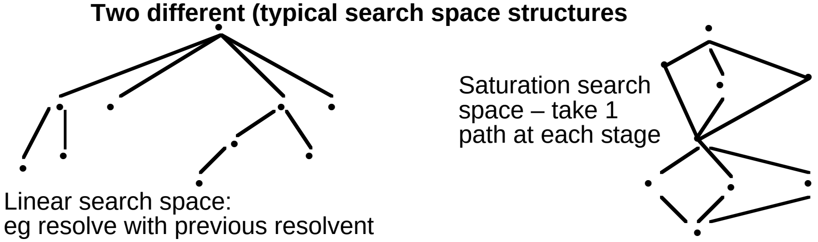
Fact: Unrestricted resolution leads to formation of far too many resolvents.

Some simple restrictions we have already seen are to:

- delete a tautology (clause with a literal and its negation eg P(x) ∨ Q(x,y) ∨ ¬P(x)
- delete a subsumed clause (which is redundant)
- generate factors (maybe just safe factors)

More generally we need further ways to restrict resolution. The most popular methods exploit the syntactic form of a clause to reduce the search space.

Two different (typical search space structures



Saturation search space – take 1 path at each stage

Linear search space:
eg resolve with previous resolvent

Types of refinement

6dii

Within a systematic search for a refutation, there will still be choices:

The possibilities give rise to a ***Search Space*** and we may ask

- how to control its size, and in
- what order to search it, or even
- if it contains all required proofs.

- ***strategy refinements*** concern <u>which</u> resolvents will be formed –

e.g. each resolution step except the first must always involve the resolvent generated at the previous step (as at least one of the clauses of the step)

e.g. each resolution step must resolve (in each clause) on a literal with the alphabetically lowest predicate

A strategy refinement affects the structure of the search space and so controls its size and the particular refutations that are possible.

- ***order refinements*** concern the <u>order</u> in which resolvents are formed –

e.g. take resolvents with smaller clauses first

An order refinement affects how the search space is searched and (in case only one refutation is desired), which refutation that would be.

In the next two lectures we'll look at a selection of syntactic resolution refinements.

Example 1:*Saturation Search* 6diii

uses the "forwards subsumption" strategy refinement
so which steps would be omitted?

and may also use the "subsumption/safe-factoring at end strategy
so which steps would be omitted?

and the stage-by-stage, or breadth-first" order refinement,
so how would the search space be explored?

and looks for one or all proofs within the search space

Other strategy refinements might be to enforce order of literal selection, or to make each step a combination of resolution steps.

Example 2: *Prolog* 6div

uses the "selection rule and linear" strategy refinement
what are these and how do they affect the Prolog search space?

and the "clause order and depth-first search" order refinement
what are these and how do they affect the order solutions are found?

and looks for all proofs in the search space

Other order refinements might be parallel search,

or investigate useless paths first –
eg goals ?...,L,... and there is no matching clause for L

Can you think of any other strategies?

Questions:
a) Explain how Prolog uses resolution, and.
b) What feature(s) of Prolog makes it unsound in some circumstances (that is, leads it to give the wrong answer)?

Miscellaneous notes on Search Spaces 6dv

A search space for a strategy refinement may be searched in (at least) 3 kinds of ways and in practice aspects of all 3 ways are used. The saturation search is just one way. Although space consuming, it is in common use. It is guaranteed to find a short proof in the search space if one exists. Two others are:

(1) Each path is taken in order and followed to its conclusion. This is not, in general, possible as a path might not terminate. Instead, a depth d is chosen and all paths are followed to this depth. If no proof is found then d is increased and the process is repeated. The partial proofs found previously to depth d are repeated. This method could miss a short proof if it did not happen to be explored first and d is initially too large.

(2) Search according to some heuristics, often chosen to be data dependent. eg one might be able to remove early on paths that become obviously useless.

We usually require a strategy refinement to be *complete*, that is

- the search space generated should contain all required solutions (proofs).

although a weaker form ensures the search space contains at least one proof (if any exists).

We also require an order refinement to be search-complete, or *fair*. That is, every branch will eventually be developed, or shown to be redundant. eg depth-first search with no depth limit is generally not fair, because of the possibility of infinite branches.

Study of refinements of resolution began in about 1970 and still continues
- for other techniques such as Tableaux and equational reasoning, refinements are also still being proposed (we'll cover some later in the course)
- whereas for natural deduction refinements are not very common

Summary of Slides 6

1. Without control, resolution generally produces a large number of resolvents, many of which are redundant.

2. Some simple control methods are forwards and backwards subsumption, tautology deletion and safe factoring. Subsumption removes clauses that would most likely, if used, lead to longer derivations. Tautologies, if used, lead to subsumed resolvents. Safe factoring replaces a clause with an equivalent, but smaller clause.

3. Generally, subsumption detection is limited to strict θ-subsumption, as other, stronger forms of subsumption are expensive to detect, and, in the case of general subsumption may be undecidable.

4. Deletion of sbsumed clauses and tautologies does not affect unsatisfiability.

5. Control of resolution (and indeed of other techniques) is a much researched area, and continues to be so. Most strategy refinements are syntactic. Semantic refinements tend to be in specialised domains.

6. A search space is the set of possible steps that can be made. For a given problem and general strategy there may be several different search spaces that can be generated, depending on the particular *strategy refinement*. Any search space may often be searched in different ways as well, depending on the *order refinement*.

For example, in Logic Programming, different search spaces will result depending on the selection of literal from each query. In Prolog, the selection is always the leftmost literal of the most recent literals added to the query. But other choices are possible. Prolog searches its search space from left to right and depth-first. It is also possible to search each branch to depth 1, then each branch to depth 2 and so on, although this uses up rather more space than depth-first. But depth-first search is not complete if branches may be infinite, as they often are in Prolog. (**Exercise**: Find such an example.)

7. A refutation (using subsumption) can constructively be transformed into a simpler refutation that does not use subsumption.

START of OPTIONAL MATERIAL (SLIDES 6)

- Saturation search in Prolog
- Wos's Obstacles to Theorem Proving

An Outline PROLOG program for Saturation Refinement

6gi

(it performs subsumption checks at the end of each saturation stage)

```
satref(Snew,Sold,K):- member([],Snew).
satref([],Sold,K):- writenl(['failed'], fail.
satref(Snew,Sold,K):- Snew≠[],  not  member([],Snew),
    resolveall(Snew,Snew,R1),resolveall(Snew,Sold,R2),
    append(R1,R2,R3), forwardsubsumed(R3,Snew,Sold,R4),
    backsub(R4,Snew,Sold, R5,Snew1,Sold1),
    append(Snew1,Sold1,Sold2),K1 is K+1,
    satref(R5,Sold2,K1).
```

- `satref(New,Old,K)` holds if `New` are resolvents formed from `Old` at stage `K` of saturation search, and `New` **union** `Old` is unsatisfiable.
- `resolveall(X,Y,Z)` holds if `Z` are all non-tautologous safe-factored resolvents between clauses in `X` and `Y`.
- `forwardsubsumed(X,Y,Z,W)` holds if removing clauses from `X` that are subsumed by a clause in `Y` or `Z` leaves `W`.
- `backsub(X,Y,Z,X1,Y1,Z1)` holds if clauses from `X`,`Y` and `Z` that are backward subsumed by clauses from `X` are removed leaving, respectively, `X1,Y1,Z1` .
- Initial call `satref(Init, [ ],0)`.

To check for subsumption immediately a resolvent is formed `resolveall` needs `Sold` as an argument and its second call from `satref` to be more complex.

Using Prolog to program a Saturation Search Theorem Prover

6gii

The Program on 6gi is only an outline. How might clauses be represented in Prolog? The easiest way is to represent a clause as a list of literals, but for quicker pattern matching, maybe a clause could be represented by a pair of lists - the first list of the pair being the positive literals and the second the negative ones. e.g. $P(x)\vee Q(x)\vee\neg R(x)$ becomes the pair ([P(x),Q(x)],[R(x)]). The data is then a list of such pairs. The empty clause would be the pair ([], []). In that case, for example, the condition `not member([],Snew)` in clause 3 would need to be changed to `not member(([],[]),Snew)`. Much of the work is done by `resolveall` which has to form all resolvents between the clauses in its first two arguments and then remove tautologies and form safe factors of the remainder, if any.

As an example of forming safe factors, consider the clause $P(x,x)\vee P(a,z)\vee P(a,a)\vee Q(z,v)\vee Q(a,u)$. In this case, by factoring literals in pairs, we could reach successively $P(x,x)\vee P(a,a)\vee Q(a,v)\vee Q(a,u)$, $P(a,a)\vee Q(a,v)\vee Q(a,u)$, $P(a,a)\vee Q(a,u)$, which subsumes the original. So, does this approach always lead to all factors?
Exercise: Try to show that it does. You might also try to program it and to consider its efficiency. Since most factors are *not* safe factors, there are clever heuristics to detect when safe factoring *isn't* possible, before trying all possibilities.

Heuristics are useful for detecting when (strict) θ-subsumption occurs. Most clauses don't subsume each other and this can be detected before a full check for subsumption is made.

E.g. if C has 5 literals and D has 4, then C cannot (strictly) θ-subsume D. If the predicates in the two clauses don't subsume each other, then nor will the clauses.
E.g. $P(...)\vee P(...)\vee Q(....)$ won't strictly subsume $P(..)\vee R(...)$ or vice versa, whatever the arguments. In conclusion, checking for subsumption is not an easy task (see Problem sheets and Slide 6cv).

Continued from Slide 6gii

Programs mostly test for strict θ-subsumption and simple cases of safe-factoring only. An example of what can happen otherwise is that some factor of C may subsume D, even though it isn't a safe factor. e.g. $C = P(x,y) \vee P(y,x)$ θ–subsumes $D=P(x,x) \vee R$ (but not strictly); $C[x==y] = P(x,x)$ is a non-safe factor of C which strictly subsumes D. You can begin to see the kinds of problems faced when constructing an efficient theorem prover.

As for the program on 6gi, notice that it contains two calls to `append`. It might be more efficient to represent the lists of clauses as *difference lists*, so that they can be appended in constant time. A general difference list representation has the form of a pair of lists, (Z, W), where W is a suffix of Z. E.g. ([1,2,3,4|W], W) represents the list [1,2,3,4]; i.e. the difference between the list consisting of 1,2,3,4 followed by some W, and W. Appending ([1,2,3,4|W],W) to ([5,6|Z],Z) results in the binding W==[5,6|Z] and the new list ([1,2,3,4,5,6|Z],Z). The single clause for append using difference lists is `append((X,Y),(Y,Z),(X,Z))`.

To resolve two clauses such as ([P(x),Q(x)],[R(x)]) and ([S(v),R(v)][]) using Prolog, first use `copy_term` to make a copy of the two clauses with fresh variables (so their variables do not become bound by Prolog when unifying R(x) and R(v)). Then the resolvent is formed, in this case ([P(x1),Q(x1),S(x1)],[]), where x1 is the fresh variable for clause1, v1 is the fresh variable for clause2 and ¬R(v1) is resolved with R(x1) with unifier v1/x1. Prolog helpfully propagates this binding to other occurrences of v1 to give the desired resolvent. A copy operation is also needed to implement a subsumption check. The potentially subsuming clause C is copied and the potentially subsumed clause D is grounded to Dg - its variables are bound to new ground terms, using the `numbervars` predicate. Then a check is made of whether C is a subset of Dg for some instance of C. (**Exercise**: Check why this works.)

6giii

Obstacles to the Automation of Reasoning (Summarised from Wos)[1]

6giv

1 Data retention: the program keeps too much information in its data base.

2 Redundant information: the program keeps generating the same information, or subsumed information, over and over again.

eg if A in data, no need for $A \vee B$. Or if $\forall x. A(x)$ in data no need for A(b).

3 Inadequate focus: the program gets lost too easily and wanders down useless paths.

1,2,3 usually result in the program generating too many conclusions, many of which are redundant or irrelevant. Problem is to detect the redundant clauses.

4 The inference rules may be too *fine-grained*, resulting in the problems 1-3, or they may be too large, or too restrictive, resulting in the alternative problem of too little information being drawn.

5 There are no general guidelines for selecting the appropriate means to control the problems in 1-4. **Remedies:** control by strategy

– syntactic kinds prohibit certain paths – easy for the program;

– semantic kinds harder – but focus on paths likely to solve problem.

– More generally, could allow deductions only if they yield facts;

6 The program may not use an appropriate representation of the information pertinent to the problem. **Remedy:** lots of experience.

7 Ordinary computing difficulties such as indexing in the database and finding appropriate information.

[1]From Automated Reasoning, 33 Basic research Problems, Prentice Hall, 1988