AUTOMATED REASONING

SLIDES 6:

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

Controlling Resolution:
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

Let C and D be clauses:

C \theta–subsumes D if C(\theta) \subseteq D for some \theta
C subsumes D if \forall C \vdash \forall D, where \forall C means that all variables in C are explicitly universally quantified. Equivalently, C subsumes D if \{C \wedge \neg D\} has no H-models (or if C \wedge \neg D \models \theta [* ]).
C strictly \theta–subsumes D if C \theta–subsumes D without necessary factoring in C\theta. Each literal in C\theta matches a different literal in D.

Example: Px \lor Qx subsumes Pa \lor Qa (and \theta subsumes it)
Px \lor Py \theta–subsumes Paa but not strictly
Pf(x) \lor \neg Px subsumes Pf(f(y)) \lor \neg Py but does not \theta–subsume it.

Relation between \theta subsumption and full subsumption:

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

Exercise

Subsumption Practice (ppt)

Note: Identical literals in a clause are always merged, so Pa \lor Pa is always Pa and Px \lor Px is always Px. They both strictly subsume Pa \lor Qa. If C \theta–subsumes D, but not strictly,
can first factor C to C’ where C’ strictly \theta–subsumes D.
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. \( \neg Dxy \lor Cxy \)
2. \( \neg Tx \lor Cxb \)
3. \( \neg Tx \lor \neg Cxb \)
4. \( Tc \)
5. \( \neg Dcz \)
6. \( (1,2) \lor Dcb \)
7. \( (1,2) \lor Dca \)
8. \( (1,5) \lor Dcb \)
9. \( (2,3) \lor \neg Dxb \lor Cxb \)
10. \( (3,4) \lor \neg Dxb \lor \neg Tx \)
11. \( (1,5) \lor Dca \)

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?

A Constructive View of \( \theta \)-Subsumption Deletion

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

Using \( \theta \)-subsumed clauses leads to redundancy in proof construction in 2 ways.

If C \( \theta \)-subsumes D, C\(=\)D, and D resolves with E giving R1 then either
  (i) C \( \theta \)-subsumes R1
  or      (ii) C resolves with E to give R2 that \( \theta \)-subsumes R1

*E.g.* Let C = Px, and D = Pa \lor Q; then C \( \theta \)-subsumes D.

Suppose D is resolved with \( \neg Q \lor R \) (ie not on the subsumed literal)
the resolvent (R1) is Pa \lor R, which is \( \theta \)-subsumed by Px (i.e. by C).

\[ \begin{align*}
\text{C=Px} & \quad \theta \text{-subsumes} \\
\text{D=Pa \lor Q} & \quad \neg Q \lor R \\
\text{R1=Pa \lor R} & \quad \theta \text{-subsumed by C=Px}
\end{align*} \]

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

If C \( \theta \)-subsumes D, C\(=\)D, and D resolves with E giving R1 then
  either (i) C \( \theta \)-subsumes R1,
  or      (ii) C resolves with E to give R2 that subsumes R1

Let C = Px and D = Pa \lor Q and suppose D is resolved with E = \( \neg Pu \lor Du \)
the resolvent R1 = Da \lor Q is \( \theta \)-subsumed by R2 = Du
(\text{the resolvent of C with} \( \neg Pu \lor Du \))

\[ \begin{align*}
\text{D=Pa \lor Q} & \quad \neg Pu \lor Du = E \\
\text{C=Px} & \quad \neg Pu \lor Du = E \\
\text{Da \lor Q} & = \text{R1} \\
\text{R2 = Du} & = \text{Da} \\
\text{R1 is \( \theta \)-subsumed by R2} & \quad \text{ie a smaller refutation}
\end{align*} \]

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

If C \( \theta \)-subsumes D, then using C instead of D gives a shorter refutation.
The subfree Property
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 Appendix I 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 \( k \geq 0 \) and used in R, Ck is not subsumed by any different clause that is in S or derived from S at a depth \( \leq k \).

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 6bv/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.

Factoring again  (ppt)

F is a basic factor of E1 \( \vee \ldots \vee E_n \vee H \) if F = (E \( \vee \) H)\( \theta \), where \( \theta = \text{mgu}(E_i) \) and E = E\( \theta \), H is a clause, E\( \theta \) are literals.

Examples
- \( P_x \vee P_y \) factors ==\( \Rightarrow \) \( P_x \)
- \( Q_x \vee Q_y \) factors ==\( \Rightarrow \) \( Q_a \)
- \( Q_x \vee Q_y \vee P_{xy} \) factors ==\( \Rightarrow \) \( Q_a \vee Q_x \vee Q_y \)

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

EG you cannot refute \( (P_x \vee P_y \vee \neg P_a \vee \neg P_b) \) without factoring.

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

eg factor \( P_x \) and \( P_y \) in \( Q_{xx} \vee Q_{xy} \vee P_x \vee P_y \) ==\( \Rightarrow \) \( Q_{xx} \vee P_x \)
factor \( Q_{xx} \) and \( Q_{y} \) in \( Q_{xx} \vee Q_{yy} \vee Q_{xy} \) ==\( \Rightarrow \) \( Q_{aa} \)

Or, a factor can be further factored:
eg factor \( P_x, P_y \) in \( Q_{xx} \vee Q_{xy} \vee P_x \vee P_y \vee P_a \) ==\( \Rightarrow \) \( Q_{xx} \vee P_x \vee P_a \)

which can be further factored ==\( \Rightarrow \) \( Q_{aa} \vee P_a \)

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

All about Tautologies

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: \( \neg Q_x \vee Q_y \) is not a tautology but \( \neg Q_x \vee \neg Q_x \) is.

Resolving a tautology \( T \) with \( S \) (on one of the tautologic literals) leads to a resolvent \( R \) that is subsumed by \( S \):
- e.g. if \( T=\neg Q_a \vee R_a \vee Q_a, S=P_x \vee Q_x \), then \( R=P_a \vee R_a \vee Q_a \) is subsumed by \( S \)

Two Examples:

\( \neg Q_x \vee R_a \vee Q_x \) a tautology
\( P_x \vee Q_x \)  
\( Pa \vee Ra \vee Qa \)  
\( Pa \vee Ra \vee Qa \)  
\( Pa \vee Ra \vee Qa \)  

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

Further Properties of subsumption and factoring.

Full subsumption is not usually checked as it can be a theorem proving problem itself that may not terminate. E.g. \( \neg P(x) \vee P(f(x)) \) subsumes \( \neg P(x) \vee P(f(f(x))) \) (i.e. the first clause implies the second) but it does not subsume it. Even checking for \( \theta \)-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 \( \theta \)-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**

**Exercise:** Write down all the factors of $Q_{xx} \lor Q_{xy} \lor P_x \lor P_y \lor P_a$

F is a **safe factor** (or a **reduction**) of C if F is a factor of C and F subsumes C

**Exercise:** Which factors of $Q_{xx} \lor Q_{xy} \lor P_x \lor P_y \lor P_a$ 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 $F \equiv D$ and F can replace D.

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

- e.g. $Q_{aa}$ doesn’t subsume $Q_{ua} \lor Q_{vv}$; the latter might be needed with some other substitutions for u and v (e.g. $Q_{ba} \lor Q_{bb}$) so it cannot be discarded.

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

---

**Types of Refinement**

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 which 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 order 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
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
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
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 $\theta$-subsumption, as other, stronger forms of subsumption are expensive to detect, and in the case of general subsumption may be undecidable.

4. Deletion of subsumed 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.
An Outline PROLOG program for Saturation Refinement

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

satref(Snew,Sold,K) :- member([],Snew).

satref([],Sold,K) :- writeln1('failed'), fail.

satref(Snew,Sold,K) :- Snew\=[], not member([],Snew),
resolveall(Snew,Sold,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
time 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.

---

Obstacles to the Automation of Reasoning (Summarised from Wos)

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 ∨ B. Or if ∨ 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.

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