Automated Theorem Proving: Background Notes I Clausal Form, Herbrand's Theorem and Resolution

Krysia Broda

January 9, 2003

1 Introduction

Almost ever since it became possible to use computers, the question has arisen in people's minds:

> can we make computers (or computer programs) think?[1]

Now, exactly what is meant by this question seems to vary according to the domain of investigation. (For example, should we classify visual pattern recognition as thinking ?) In this course, 'thinking' will mean reasoning logically from a data base of facts in order to reach conclusions. This includes areas such as common sense reasoning and puzzle solving, mathematical theorem proving, and rational argument. See [Wos 85].

The approach will provide a historical perspective of some aspects of theorem proving. In the present research climate, when attempts are being made to automate all kinds of logics and use various inference methods, this seems to be justified – for example, specific approaches for modal logics will not be covered. Theorem proving in these logics can be compiled into first order logic and approaches developed for that case then applied. The logic used in this course is therefore restricted to first order predicate calculus. There are several reasons for this, which include both the fact many of the problems encountered in the different areas are similar to those encountered in theorem proving using first order logic and the limitations of time and space. The specific area of theorem proving using equations will also be considered as the methods are so useful and interesting.

On the other hand, specific methods for reasoning in any particular domain will not be included. Since this aspect borders on Artificial Intelligence, in that each domain gives rise to domain specific methods, allied to knowledge representation issues, it provides a useful demarcation for us.

[1] How long will it be before the answer is emphatically yes, I wonder?

The *problem* we want to solve (by automated deduction) is whether certain conclusions follow from given premises; *i.e.*

Does data $\models$ conclusion?

Quite often, a justification of the answer (*i.e.* a 'proof') is also desired. In many circumstances, *any* justification of a 'yes' answer to the problem is considered adequate and of course the smallest, or simplest, is often the most sought after. But since the advent of logic programming and its associated concepts, methods of automated deduction have been used also as a means of providing solutions to non-Horn clause logic programming problems. In these applications, rather more solutions are usually expected – the goal is usually existential (of the form $\exists P$) and all witnesses that make P true are wanted, not just one.

None of the difficulties of knowledge elicitation, nor, in general, considerations of the pertinence of the premises will be of concern here. However, discussion of the form of premises, for example whether to use equations, is appropriate, for that can affect the method to be used.

There are several general classes of methods that could be employed to tackle the problem:

- proof by contradiction, in which the data and the negation of the conclusion is proved inconsistent, hence showing the validity of *data* $\models$ *conclusion*. (Refutation)
- proof by procedural or 'backwards' style rules. (*e.g.* Prolog and its procedural interpretation).
- proof by 'forwards' style rules (*e.g.* natural deduction).

We will be most interested in those features of the methods which render them suitable for automated deduction and in refinements to improve their efficacy in finding a proof. Thus, although a method may be possible, it may be impracticable to use because the search space is too large. In that situation one should try to reduce this search space.

Traditionally, many automated deduction systems have concentrated on finding refutations – *i.e.* in proving inconsistency of a given set of sentences – and no explicit mention of a conclusion is made. If the conclusion happens to take the form of an existential formula –

show there exists a y such that ...

then in a refutation the negated conclusion can be used one or more times. This corresponds either to an answer of $y = a$ (say), if used once, or, if used more than once, to something of the form "$y = a$ or $y = b$ but I do not know which". Answers like this may be of interest to the originator of the problem – for example, if the problem is a puzzle. For these reasons it is important to consider the size of the search space as well as suitable ways of searching it, for it may not be appropriate simply to avoid whole portions of it altogether.

Automated deduction became a very active field after the development of the resolution inference rule and unification. (See [Rob 65]). Before then, refutation methods tended to be of the 'British Museum' kind: for example, enumerate sets of instances of the data, hoping to find an inconsistent set. This approach is based on Herbrand's theorem for clauses (or compactness), which states that, S is inconsistent if, and only if, there exists a finite set of instances of a set of clauses S which is inconsistent. However, the difficulty is in choosing appropriate instances of sentences in the data to include in a search for inconsistency of data and negated goal.

Some of the early approaches are of interest however, notably Newell and Simon's The Logic Theorist Machine, the Davis and Putnam method, Gerlenter's Geometry machine and Prawitz methods. Accounts of these to be found in [Bun 83] and [SiekWright 83] are recommended. Section 4 will discuss the Davis and Putnam (DP) method, which is still in use today. It is especially used for model checking in classical propositional logic.

It is useful to be aware of different view points from which deduction is viewed and consequently used. For then, various relationships between the different methods can be fruitfully explored and exploited. Indeed, this is happening right now in active research on semantic tableaux and sequent calculii as a paradigm for automated deduction. Traditional methods used for clausal deduction are being adapted to the new framework and to many non-classical logics too. By studying the basic ideas you will be in a good position to adapt them to new areas, when required. The three areas covered in depth in this course will be clausal deduction, semantic tableau deduction and equational reasoning.

The course slides will include all examinable material unless otherwise stated. They may occasionally include extra material but this will be stated too. The background notes are meant to be just that. They may provide additional perspective on a topic, more details of proofs, further examples, etc. However, if you regularly attend lectures, then the slides and notes taken in lectures should be adequate for the problem sheets and examination.

2 Clausal Form

Please note: the glossary at the end of this chapter collects together definitions of various logical terms.

Ever since the Davis and Putnam method was proposed, which was the first practical automatic method, most practical methods have required the input to be transformed into clausal form, which retains the consistency or otherwise of the original sentences. Clauses are a practical notation for finding refutations since they have a uniform structure. Furthermore, most practical non-clausal theorem proving methods (for example using semantic tableaux) also make use of many of the ideas suggested for improving theorem provers for clausal form so it is of interest to study clausal methods.

For convenience we give a set of rules for conversion into clausal form. See the glossary for definitions of the various notions involved.

There are several steps (given below) used to convert formulae into clausal form. The aim is to reduce the initial sentences to *conjunctive-normal-form*, that is, a conjunction of disjunctions.

1. Apply rewrite rules to formulae to remove $\rightarrow$, $\leftrightarrow$, and push negations inwards so that only atoms are negated. ($\Rightarrow$ is read as rewrites to).
   - $A \leftrightarrow B \Rightarrow A \rightarrow B \wedge B \rightarrow A$
   - $A \rightarrow B \Rightarrow \neg A \vee B$
   - $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$
   - $\neg(A \rightarrow B) \Rightarrow A \wedge \neg B$
   - $\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$
   - $\neg\neg A \Rightarrow A$
   - $\neg\forall x.P \Rightarrow \exists x.\neg P$
   - $\neg\exists x.P \Rightarrow \forall x.\neg P$
2. Rename variables so that each quantified variable has a different name. (Makes step 4 easier).
3. For each subformula of the form $\exists x_i.S$, remove $\exists x_i$ and replace each occurrence of x_i in S by a term $f(\ldots)$, where f is a (new) function symbol not used in any of the sentences of the problem and its arguments are the variables of all universal quantifiers within whose scope $\exists x_i$ lies and which occur in S. If f has one or more arguments it is called a Skolem function and in the special case of no arguments it is called a Skolem constant. Each Skolem function introduced in this way should be different, although identical subformulas may make use of the same function f.
4. Pull the universal quantifiers into a prefix using equivalences such as
$$\forall x.A \vee B \Rightarrow \forall x[A \vee B] \text{ and } \forall x.A \wedge B \Rightarrow \forall x[A \wedge B]$$
5. Convert to conjunctive normal form using
   - $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (A \vee C)$
   - $(B \wedge C) \vee A \Rightarrow (B \vee A) \wedge (C \vee A)$
6. Distribute quantifiers back over conjunctions using
$$\forall x[A \wedge B] \Rightarrow \forall x A \wedge \forall x B$$
7. Remove tautologies and subsumed clauses.

There are various equivalences that can be used in order to reduce the scope of universal quantifiers so that the Skolem functions introduced have as few arguments as possible and/or as few as possible are introduced.

All the equivalences below can be useful for this.

- $Qx(A \vee B) \Leftrightarrow Qx.A \vee B$ for x free in A but not B (Q may be $\exists$ or $\forall$)
- $Qx(A \wedge B) \Leftrightarrow Qx.A \wedge B$ (restrictions as above)
- $\forall x(A \wedge B) \Leftrightarrow \forall xA \wedge \forall xB$
- $\exists x(A \vee B) \Leftrightarrow \exists xA \vee \exists xB$,
- $\forall x \forall y.A \Leftrightarrow \forall y \forall x.A$
- $\exists x \exists y.A \Leftrightarrow \exists y \exists x, A$

(**Note:** When dealing with clauses, we will make the assumption that the domain of any interpretation is always non-empty. Otherwise, some of the above bi-implications would only be true in one direction.)

As an example of the possibilities, consider

$$\forall y[\exists x.P(x,y) \vee \exists z.Q(z,y)] \equiv \forall y \exists z[P(x,z) \vee Q(z,y)]$$

By using the equivalence one Skolem function will result, not two. After conversion to clausal form, we have the following property in Theorem 1.

Theorem 1 *Given a set of sentences S, and their conversion to clauses C, the set C is inconsistent iff the set S is inconsistent.*

Steps 1, 2, 4, 5 and 6 apply equivalences as the rewrite rules. Thus, if before applying such a step a set of sentences S has the set of models M, then after substituting for equivalences the resulting set of sentences S' has exactly the same set of models M. That is, S is equivalent to S'. In particular, if M is empty before the application of such a step, the resulting set of sentences also has no models.

(**Exercise:** The substitutivity of equivalences should be proved to apply to nested subformulae, by induction. *i.e.* if $B \equiv B'$ then $S[B] \equiv S[B']$, where $S[B]$ is a sentence containing B, and $S[B']$ is the same sentence with one or more (identified) occurrences of B replaced by B'. [Use depth of expression tree as quantity on which to perform induction]. Also show steps 1,2, 4, 5 must terminate.)

Steps 3 and 7 do not preserve equivalences, but they do preserve satisfiability. For step 3, given a set of sentences S:

S is satisfiable iff the skolemised form of S is satisfiable
$\equiv S$ has a model (say M) iff the skolemisation of S has a model (say M')
($M \neq M'$ since M' must include denotations for skolem functions)
$\equiv S$ has no models iff the skolemisation of S has no model

(**Exercise:** For a proof of step 3 see [Bun 83] or the slides.)

For step 7, we must show that if a set of clauses S is unsatisfiable, then so is the set S', formed by removing tautologies and subsumed clauses.

a

Let $Sig(S)$ be given and suppose S contains the tautologous clause $T = A \vee \neg A \vee C$, where C is none or more literals. (There will in general be variables in A, C etc.) Suppose $S - T$ has a model M using $Sig(S)$. M will be a model of T since T is a tautology. Therefore, if S has no models, then neither does $S - T$. And obviously, if $S - T$ has no models, then neither does S. So S is unsatisfiable iff $S - T$ is unsatisfiable.

b

Let S contain the clauses S_1 and S_2, where S_1 subsumes S_2. *i.e.* $\forall S_1 \models \forall S_2$. Let S have model M using $Sig(S)$, then clearly M is a model of $S - S_2$. Also, if M is a model of $S - S_2$, it will be a model of S_2 since it is a model of S_1 and $\forall S_1 \models \forall S_2$.

If the model of $S - T$ is not required to use $Sig(S)$ (for example it uses $Sig(S - T)$, which may be a proper subset of $Sig(S)$), then M must be extended to $Sig(S)$ in order to become a model of S. Since T is a tautology the new terms can be interpreted in any way. To maintain satisfaction of $S - T$ the interpretation of the new terms should mirror those of an existing term.

(**Exercise:** Show that, for clauses $S1$ and $S2$, the property $S_1\theta \subseteq S_2$ implies the definition of subsumption: $\forall S_1 \models \forall S_2$.)

3 Some Useful Properties of Clauses and Semantic Trees

This section underpins most of the theory of resolution.

Theorem 2 *(Unsatisfiability)*

(a) A set of clauses S is unsatisifable
(b) iff there is no model of S (over any domain)

(c) iff there is no model of S over the Herbrand Universe
(d) iff each Herbrand Interpretation falsifies some ground instance of a clause in S.

Proof:

(a) iff (b) follows by definition of unsatisfiable.

(c) iff (d) follows by definition of falsifying a clause.

(b) iff (c) follows from
(i) S has no models $\Rightarrow$ S has no Herbrand models. (Obvious !)
(ii) S has no Herbrand models $\Rightarrow$ S has no models.

We prove the contrapositive of (ii): if S has some model then S has a Herbrand model. Suppose S has some model I, over domain D. We will construct a Herbrand model M^* from I, which also satisfies S.

Construction: HB = Herbrand Base, HU = Herbrand Universe, of S (or of the Signature of S).
Let $mapterm\ : HU \to D$ and $assigntruth : HB \to \{True, False\}$
We want to define *mapterm* and *assigntruth* so that *assigntruth* gives a model M^* of S.

For each constant c in HU, $mapterm(c)$= valuation of c in D under I.
For each function symbol in Signature of S and $t_1 \dots t_n \in HU$,
$mapterm(f(t_1 \dots t_n))$ = valuation of $f(mapterm(t_1) \dots mapterm(t_n))$ in D under I.
For each predicate symbol in Signature of S and terms $t_1 \dots t_n \in HU$,
$assigntruth(P(t_1 \dots t_n))$ = truth of $P(mapterm(t_1) \dots mapterm(t_n))$ in I.

Let M^* be given by *assigntruth*, then M^* is certainly a Herbrand Interpretation and we need to show M^* satisfies S. For each clause C in S, and for each substitution θ of universal variables in C by objects in HU, let θ' be the corresponding substitution by elements of D, given by replacing each equation $x == t$ by $x == mapterm(t)$. We want to show $C\theta$ is true in M^*. Now, $C\theta'$ is true in I since I is a model of S. Hence $C\theta'$ contains some literal $L\theta'$ true in I. Suppose $L\theta' = P(t'_1 \dots t'_n)$, where each t'_i is an image under *mapterm* of some $t_i \in HU$ (prove!), then $assigntruth(P(t_1 \dots t_n))$ is also true in M^* , and clause $C\theta$ is true in M^*.

For example: $S = \{\forall x[\neg P(f(x)) \leftarrow P(x)]; \quad P(a)\}$
I is the model with Domain $\{0, 1\}$, such that a is interpreted as 0 and $f(a)$, $f(f(a))$, etc., are all interpreted as 1.
Also, $P(0)$ is assigned true and $P(1)$ is assigned false.
M^* is then constructed as follows:
$HU = \{a, f(a), f(f(a)), \dots\}$;

$HB = \{P(a), P(f(a)), \ldots\}$;
$mapterm(a) = 0$;
$mapterm(f(a)) = mapterm(f(f(a))) = \ldots = 1$;
$assigntruth(P(a))$ = truth of $P(0)$ in I = true;
$assigntruth(P(f(a))$ = truth of $P(1)$ in I = false, etc.
You can check that M^* is a model of S.

§

Lemma 1 *(Konig's Lemma)*

If a tree formed with a finite number of branches descending from each node has an infinite number of nodes then it has an infinite branch.

Proof: (Outline) See Figure 1 for identification of x_0, y_0 etc. Call a node *immortal* if it has an infinite number of descendants, otherwise mortal. By assumption, x_0 is immortal, hence at least one of $y_1 \ldots y_n$ must be immortal, else x_0 is mortal. (Suppose it is y_2). Similarly, if x_k is immortal at level k, $\exists x_{k+1}$ immortal at level $k+1$ (In the case illustrated $k = 0$ and $x_1 = y_2$; x_0 is immortal at level 0.) So in any tree with an immortal node x_0 there is an infinite branch consisting of only immortal nodes, namely $x_0, x_1, x_2 \ldots$. §

Theorem 3 *(Herbrand's Theorem for semantic trees)*

(a) A set of clauses S is unsatisfiable
(b) iff the complete semantic tree for S is finite and closed (each branch ends in a failure node)
(c) iff a finite set of ground instances of S is unsatisfiable.

(See Glossary for definitions of terms used in proof.)

Proof: We show (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (a). Hence (a) $\Leftrightarrow$ (b) and (b) $\Leftrightarrow$ (c).
We also use the results of Theorem 2 and confine our models to Herbrand models.

(c) $\Rightarrow$ (a) Suppose a finite set of ground instances of S, S', has no Herbrand models. Then, for any Herbrand interpretation I, S' is false in I, so for some instance C' in S', C' false in I. Hence, for some clause C, of which C' is an instance, C is false in I and so S is false in I. Since I is any interpretation, S is unsatisfiable.

(b) $\Rightarrow$ (c) Given a closed finite semantic tree T for Signature of S, then each branch ends in a failure node labelled by an instance C' of some clause in S. Since the clauses C' also use the same signature S as the clauses C, T is a closed semantic tree for the set of clauses C'. Hence C' have no models and are unsatisfiable.

(a) $\Rightarrow$ (b) Each branch of a complete semantic tree describes a Herbrand Interpretation of S, which, since S is unsatisfiable, must falsify some ground instance of a clause in S, say C'. For a typical branch B, starting at the root, which falsifies C' there is a sub-branch B' (also starting at the root), which already falsifies C' – for C' has only a finite number of literals. B' ends in a failure node for C'. The branches B' form a finite sub-tree T' of T, which is finite, since each B' is finite (use Konig's Lemma). T' can be pruned by removing descendants of failure nodes and is then a finite closed semantic tree for S. See Figure 2.

§

Semantic trees can be used to give a decision procedure for certain classes of clauses S.

If the complete semantic tree for a set of clauses of S is finite (which is the case, for example, if S contains no function symbols) then a closed tree shows S unsatisfiable. A non-closed, complete tree gives models for S, along the open branches.

4 The First Practical Automatic Proof Procedure

Although the method of semantic trees can be used as a semi-decision procedure for first order clauses they are not a very efficient method. (Consider, for example, how to decide whether the atoms in a particular branch falsify any clause instances.) Certainly, a judicious choice of atoms at each level in the tree can help to cut down the size of the tree. But such control is not easy to automate.

An alternative approach, which makes use of Herbrand's theorem, is to enumerate sets of ground instances of a set of clauses S and use the Davis and Putnam procedure for propositional clauses (or sentences) (see below) for testing unsatisfiablility of each set. However efficient the latter procedure is, a little thought is enough to see that the real problem is in how to choose the right set of instances.

For propositional problems the Davis and Putnam procedure is *very* efficient and is still used in large theorem provers . A particular way to enumerate sets of 'useful' ground instances is described in [LeePlais 92], emphasising that in this field old methods can often prove useful when combined with newer ideas. The most practical solution to this problem for clauses is resolution, introduced by Robinson [Rob 65]. This inference rule has formed the basis of much work in automated deduction since 1964.

Before resolution was invented, Prawitz used an approach that makes use of so-called *dummy variables*[Praw 70]. Instead of taking ground instances of clauses, substitution of variables was made by a dummy term which stood for an unknown, but particular constant. Equations were set up to solve for the unknown constants. A whole calculus has been developed along these lines by Bibel [Bib 83], incorporating more recent ideas of connection graphs, and an efficient proof procedure for solving mathematical theorems is

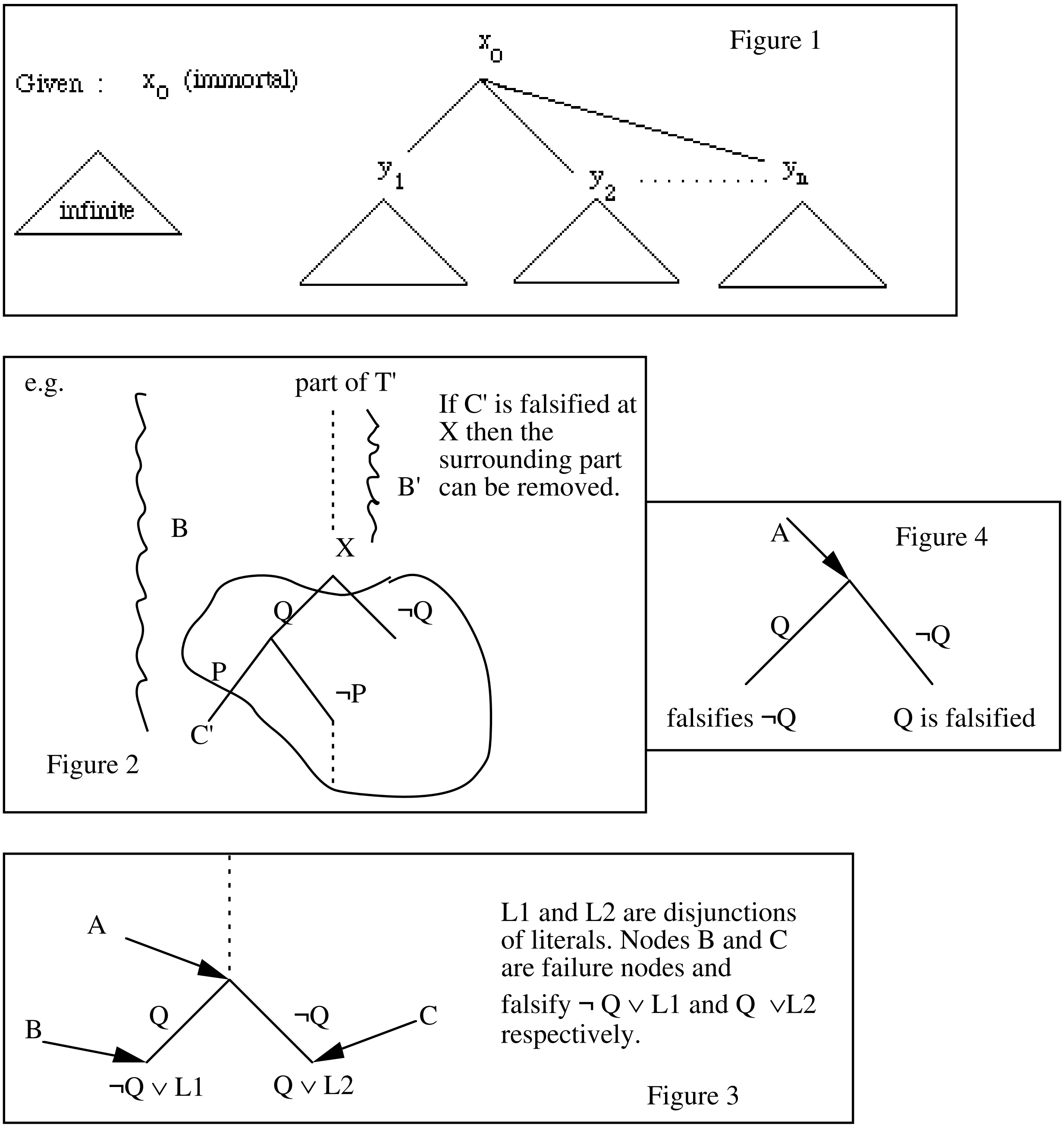

Figure 1: Figures 1 to 4

the result. We shall illustrate the flavour in a refutation setting, in this section. Wallen has adapted the methods and applied them to non-classical logics as detailed in [Wall 90]. The approach has been incorporated into Semantic Tableaux procedures to obtain another class of very adaptable theorem provers. (See [TabHand 99].) The next two sections briefly discuss the Davis and Putnam procedure, and its augmentation using dummy variables. Section 5 then continues with the theory of resolution.

4.1 The Davis Putnam Procedure

In this section we describe the Davis-Putnam procedure.

The Davis-Putnam decision procedure (DP) shows (in)consistency [2] of a given set of ground clauses S. In fact, it can be extended to arbitrary propositional sentences, as shown on the slides and it is not difficult to adapt the proofs.

The method consists of the following steps:

- Remove tautologies – (as in step 6 of preparation into clausal form.)
- Iterate the following rules, which generate smaller and smaller sets from the initial S, until either all sets are contradictory explicitly (i.e. contain P, $\neg P$) or one of them is the empty set. If the former, the set S is unsatisfiable. If the latter, S is satisfiable. The slides also show how the method can be used to extract a model in case S is satisfiable.
  The following notation is used: l^c is the complement of l – if l is positive then l^c is $\neg l$; if l is negative, say $\neg k$, then l^c is k.

  a

  If literal l is in a clause but no occurrence of l^c appears in any clause, delete all clauses containing l. (l is called *pure* in this case.)

  b

  Remove any clause which is subsumed by another clause.

  c

  For a one-literal clause l remove each occurrence of l^c in any other clause and delete all clauses containing l.

  (Notice, that if a one-literal clause l (say) is present, then (b) will anyway cause all clauses containing l to be deleted because they are subsumed by l.)

  d

  If (a), (b) or (c) cannot be applied to the set under consideration and it does not contain a literal and its negation, then choose a literal l in some clause in

[2] (in)consistency and (un)satisfiability are used interchangeably in these notes

the remaining set and replace the clause set by two sets of clauses: the first set formed as in (c), and the second set formed by removing all clauses containing l^c and removing l from all remaining clauses.

Example 1

Given

		Then,	
(1)	$P \vee \neg Q \vee R$	by (a)	remove (5)
(2)	$\neg P$	By (c)	using (2), obtain
(3)	$\neg S \vee \neg R$	(1a)	$\neg Q \vee R$
(4)	$P \vee Q$	(3)	$\neg S \vee \neg R$
(5)	$T \vee Q$	(6)	S
(6)	S	(4a)	Q

By (c) using (6), (4a), obtain (1b) R and (3b) $\neg R$.

If clause (6) had not been present, then from the stage consisting of clauses (1a), (3), (4a) the empty set would be generated: remove (3) by (a), then (1a), again by (a), and finally (4a) by (a). In this case, by making the deleted pure literals and the literals l used in step (c) true, a model for the clauses (1) - (5) can be found, namely $\{T, \neg P, \neg S, R, Q\}$. i.e. T, R and Q are assigned True and P and S are assigned False.

Example 2

Given

		Then,			
(1)	$P \vee Q$	by (d)	choosing P		
(2)	$P \vee \neg Q$		we obtain:	and	
(3)	$\neg P \vee R$	(3a)	R	(1a)	Q
(4)	$\neg R \vee \neg P$	(4a)	$\neg R$	(2a)	$\neg Q$

Both sets are immediately contradictory.
If instead we had chosen Q, we would have obtained:

(2a)	P	and	(1a)	P
(3)	$\neg P \vee R$		(3)	$\neg P \vee R$
(4)	$\neg R \vee \neg P$		(4)	$\neg P \vee \neg R$

After applying (c) in each case we obtain two contradictory sets.

Example 3

Given ($\{u, v, x, y, z\}$ = set of variables)

(1) $S(a)$
(2) $\neg P(f(a))$
(3) $P(u) \vee Q(f(v), v)$
(4) $\neg S(z) \vee \neg R(z)$
(5) $P(x) \vee \neg Q(x, y) \vee R(y)$

To use the Davis and Putnam method, you have to 'guess' at correct bindings or substitutions for the variables taken from the $HU = \{a, f(a), f(f(a)), \ldots\}$.

Note that the signature may include symbols that do not occur in any clauses in S. As an exercise, show that if an unsatisfiable set of clauses S does not mention a particular symbol, then there is a finite set of unsatisfiable instances of S which also do not mention the symbol. Use an argument based on semantic trees. This property means that only substitutions using terms mentioning symbols in S need be considered.

Moreover, it might be possible to assume that just one copy of each clause is required. If this fails to deliver an unsatisfiable set, then up to two copies could be admitted. Since the HU is, in general, infinite (as in this example), let us decide to substitute with terms of order up to 1, when one copy of a clause is allowed, up to order 2, when two copies are allowed and so on, where a is of order 1, $f(a)$ of order 2, etc.

There is just one term up to order 1 (*i.e.* just a can be substituted), so there are 5 clauses, one instance of each. If 2 copies of each clause and substitutions of terms up to order 2 are allowed, then for clause (4) there are exactly 2 instances and for clauses (3) and (5) there are 4 different instances. Since only 2 are allowed of each clause, this leads to 6 possible combinations for each of clauses (3) and (5) and a total of 36 different combinations of up to 2 instances of each clause.

However, by examining the clauses we can reduce this number substantially. There is no point in keeping any instance of (4) other than that for which $z == a$ because otherwise clauses (4), (1) would immediately be deleted by step (a) of DP. Similarly, we are forced to accept $z == a$ and $x == u == f(a)$, reducing to 1 the remaining number of different sets of possible instances using terms up to order 2 to be considered. This gives the clauses

(1a) $S(a)$
(2a) $\neg P(f(a))$
(3a) $P(f(a)) \vee Q(f(a), a)$
(3b) $P(f(a)) \vee Q(f(f(a)), f(a))$
(4a) $\neg S(a) \vee \neg R(a)$
(5a) $P(f(a)) \vee \neg Q(f(a), a) \vee R(a)$
(5b) $P(f(a)) \vee \neg Q(f(a), f(a)) \vee R(f(a))$

which can be shown to be unsatisfiable quite easily by the Davis Putnam method.

The idea then, is to treat the variables x, y, z, u, v as dummy variables – new ones for each clause instance used. Each dummy variable can be given exactly one value. The above method shows the flavour of the one described by Prawitz (see [Praw 70]) although that never gained the popularity of refinements of resolution, and, more recently, of the one introduced by Plaistead in a method called Hyper-linking [LeePlais 92].

Example 3 above was particularly simple, in that, for example, $\neg S(z)$ could only be matched by $S(a)$, making any other instance pure. Suppose the literal $S(b)$ had also occurred – then there would be 2 possible non-pure instances of clause (4).

If instead one tried to extend the Davis-Putnam method directly to first order clauses a problem is that the procedure deletes all occurrences of an atom in each step. With ground clauses this is quite sensible. With non-ground clauses it is more difficult as it is not known whether, for example, $P(z)$ and $P(w)$ are to be the same instance or not. Most modern approaches attempt to find suitable sets of ground instances before applying the procedure.

Considering again the original Davis and Putnam method, to prove its correctness you need to show that the satisfiability/unsatisfiability of a set of sets of clauses is not altered by steps (a) – (d).

In case of (a) – (c) the (un)satisfiability of the clause set being worked on is not altered. In case of (d) if the clause set being worked on is unsatisfiable then so are both resulting clause sets, and if satisfiable then at least one of the resulting clause sets is satisfiable. That the procedure terminates is guaranteed, since the number of atoms is reduced in (a), (c) and (d), and rule (b) replaces a set by one with fewer clauses and no new atoms. Thus each set of clauses will eventually have just one atom (say A), (or no atoms). Because of the preservation of satisfiability this will either be A, $\neg A$ in case unsatisfiable or just A if satisfiable. The latter will be reduced to the empty set of clauses by step (a).

We briefly outline the proof leaving the reader to fill in the details. (A, B and C are conjuctions of clauses).

a

The set of clauses S to which the rule is to be applied can be written as $(l \vee A) \wedge B \ldots$, where neither A nor B contain l^c. Then $(l \vee A) \wedge B$ is unsatisfiable iff B is unsatisfiable.

b

See proof of step (7) in conversion of formulae to clausal form (though it is much easier here, in the propositional case).

c

S can be written as $l \wedge (l \vee A) \wedge (l^c \vee B) \wedge C$, where C contains no occurrences of l or of l^c. Then $l \wedge (l \vee A) \wedge (l^c \vee B) \wedge C$ is unsatisfiable iff $C \wedge B$ is unsatsifiable. (It is actually easier to argue for the contrapositive: that $l \wedge (l \vee A) \wedge (l^c \vee B) \wedge C$ is satisfiable iff $C \wedge B$ is satisfiable.)

d

> S can be written as $(l \vee A) \wedge (l^c \vee B) \wedge C$, where C contains no occurrences of l or l^c. Then $(l \vee A) \wedge (l^c \vee B) \wedge C$ is unsatisfiable iff $A \wedge C$ and $B \wedge C$ are both unsatisfiable (or $(l \vee A) \wedge (l^c \vee B) \wedge C$ is satisfiable iff either $A \wedge C$ or $B \wedge C$ is satisfiable).

The algorithm given in the slides can also be shown to satisfy the following property by considerations similar to those used above.

> The arguments of calls to $DP(M, S)$ satisfy the invariant:
> $M \cup S$ has a model (i.e. M can be extended to a model of S) iff either
> $M' \cup S'$ has a model (in single call cases), or
> at least one of $M' \cup S'$ or $M'' \cup S''$ has a model (in the otherwise case).

Hence, if the original clauses are unsatisfiable, then the procedure must terminate eventually with the empty clause in the second argument in all calls to DP. If the original clauses have a model then the procedure terminates with a (partial) model M and the second argument empty.

5 Resolution

5.1 Ground Resolution

The *ground binary resolution* rule is defined as follows:

> Given two clauses $E \vee F$, $\neg E \vee H$ in S,
> the *resolvent* is $F \vee H$, where $F \vee H$ is not a tautology nor subsumed by any other clause in S.

$E \vee F$, $\neg E \vee H$ are called *parent clauses*, E is a literal, F and H are clauses and $E \vee F$, $\neg E \vee H$ have no duplicate literals. Any duplicate literals can be removed without affecting satisfiability.

A proof by resolution begins with a set of clauses S_0, and continually forms resolvents, adding them to S_0, to form S_1, S_2, etc. The aim is to show S_0 is unsatisfiable. If we have, for every i, S_i unsatisfiable iff S_{i+1} unsatisfiable, then we can conclude S_0 unsatisfiable iff S_n (say) is unsatisfiable. Now it may be easy to see S_n is unsatisfiable, for perhaps it contains P and $\neg P$. In that case S_{n+1} would contain $\Box$, the *empty clause*, which is always false, and results from resolving P with $\neg P$.

For resolution to be a useful inference rule, it must therefore be *sound. i.e.* S_i unsatisfiable iff $S_i \cup \{R\}$ is unsatisfiable, where R is some resolvent formed from S_i. For then, as explained above, if we derive S_n containing the empty clause we can conclude that S_0 is unsatisfiable; i.e. we cannot conclude unsatisfiability incorrectly, by deriving $\Box$ from some satisfiable set.

Secondly, resolution should be *complete*. If we keep forming resolvents, then for an unsatisfiable set of clauses S we should be sure that we can derive $\Box$. Although completeness is not so important as soundness, it is certainly desirable.

We now prove soundness and completeness of ground resolution. (In the slides a proof of the soundness of resolution is given that depends on various logical implications and equivalences between clauses, their instances and implications. The proof there is obviously applicable to ground clauses – just ignore any mention of substitutions. The proof here goes back to the basic definitions of satisfiability.)

Soundness of ground resolution

Let S be a set of clauses and R a resolvent of two clauses in S. Then S is unsatisfiable iff $S \cup \{R\}$ is unsatisfiable.

Proof: If S is unsatisfiable, then so is $S \cup \{R\}$, where R is some resolvent. (Obvious).

Now suppose $S \cup \{R\}$ is unsatisfiable. *i.e.* every Herbrand interpretation for the signature of S falsifies either a clause in S or it falsifies R. Obviously those interpretations which falsify a clause in S falsify S, so consider an interpretation I which falsifies R. Suppose R is the resolvent of $\neg E \vee H$, $E \vee G$, *i.e.* R is $H \vee G$. Suppose I assigns true to E, then I falsifies every literal in $\neg E \vee H$ (since I falsifies $H \vee G$). If I assigns false to E, it falsifies every literal in $E \vee G$. In either case I will falsify S. Thus S has no models and $S \cup \{R\}$ unsatisfiable $\Rightarrow$ S unsatisfiable. §

The proof can equivalently be viewed as showing that R is a logical consequence of S *i.e.* any model of S is also a model of R. Hence the logic is sound. We cannot deduce unsatisfiable sets of clauses from satisfiable ones, nor satisfiable sets from unsatisfiable ones.

Completeness of ground resolution

Note: Because resolution is a refutation method, completeness of resolution refers to *refutation completeness*. *i.e.* for an unsatisfiable set of clauses S, $\Box$ can be derived from S by resolution. Resolution is not complete in the following sense: from a set of clauses S any logical consequence of S can be deduced. *e.g.* $P \models P \vee Q$, but $P \vee Q$ cannot be deduced from P directly by resolution. $P \vee Q$ is not a resolvent of $\{P\}$.

There are several methods for proving refutational completeness of resolution. We shall give a proof based on semantic trees, as it is constructive (and instructive) and can also be adapted for different refinements of resolution. (A refinement restricts the possible resolutions which can be performed in some syntactic way, so that the search for $\Box$ is reduced, since there are fewer possibilities at each stage.) Other methods of proof will be considered later.

Proof: Recall Herbrand's Theorem for Semantic Trees, which states that we can construct a closed finite semantic tree for an unsatisfiable set of clauses. In any such tree which is non-empty there will be at least one occurrence of the pattern shown in Figure 3.

Node B must be labelled by a clause containing literal $\neg Q$, else, assuming the branches are falsified as soon as possible, the clause would be falsified at A. Similarly, at node C, Q must be present. The resolvent $R = L_1 \vee L_2$ is falsified at A, else B and C would not both be failure nodes. Hence $S \cup \{R\}$ has a smaller closed semantic tree than S alone. The argument can be repeated until the tree has just one node of type A. See Figure 4.

Hence, if the search for resolvents follows those steps indicated by the semantic tree, eventually enough resolvents will be deduced so that the semantic tree has the form in Figure 4 and the empty clause can be deduced. Then the resolvent of $\neg Q$ and Q is $\Box$, which is falsified at A. §

Of course, the proof of completeness does not help us in finding the right resolvents to form (unless you have a semantic tree). A complete breadth-first search of the possible sets of clauses derivable using resolution still has to be made, in which all resolvents are found, so as not to fail to find the correct ones needed to derive $\Box$. The study of how to control this search has occupied, and continues to occupy, many of the people involved in automated deduction and will form the topic of later lectures.

5.2 General Resolution

So far we have proofs of soundness and completeness for ground resolution. The general resolution rule *lifts* ground resolution and in so doing includes unification. It is defined as follows –

> Given two clauses $E \vee F$, $\neg G \vee H$ in a set of clauses S,
> where E, F, $\neg G$, H are disjunctions of literals, all those in G being negative, and with no variables in common between the two clauses, a *resolvent* is $(F \vee H)\theta$, where θ is the most general unifier (mgu) of the literals in E, G and $(F \vee H)\theta$ is neither a tautology nor subsumed by any other clause in S.
> (θ is mgu of E, G if $E\theta = G\theta$. *i.e.* all literals in $E\theta$ are identical to all literals in $G\theta$.)

This definition combines three operations which are usually separated in practice. Namely, *binary resolution* in which E, G are single literals, *factoring* in which literals in a clause are unified and then merged and *checking forward subsumption* when the resolvent is subsumed by another clause in S. That factoring is necessary is easily seen by the example $S = \{p(u) \vee p(v), \neg p(x) \vee \neg p(y)\}$. (Try to deduce $\Box$ without any factoring – you can't!).

The unification theorem states that a mgu exists for a set of literals if any unifier exists. Robinson gave an algorithm for finding the mgu, but it is not the most efficient. Perhaps one of the neatest algorithms is given in [MartMont 82]. We assume here that such an algorithm exists and is known to the reader. (One algorithm is given in the slides.)

By combining the operations of factoring and binary resolution, simpler proofs result for completeness. It is left to the reader to show that the proofs given can be adjusted to cope with the separate operations. The proof of soundness is given on the slides (but note that it depends on several equivalences and one implication, which all have to be shown too).

The completeness proof given on the slides is similar to the ground level proof, but uses in addition a "lifting lemma", which states that, for each ground resolution step between ground instances C_1' and C_2' of clauses C_1 and C_2 and which derives R' and indicated in the semantic tree argument, there is a corresponding resolution step between C_1 and C_2 deriving R such that R' is a ground instance of R. This lemma is illustrated in Figure 5 and the slides. It is assumed that C_1 and C_2 have no common variables and that $E\phi = G\lambda$. Therefore, $F\phi \vee H\lambda = (F \vee H)\phi\lambda$. Also, $E\phi\lambda = G\phi\lambda$ and since θ is the mgu of E and G, $\theta\rho = \phi\lambda$ for some substitution ρ.

Hence $(F \vee H)\phi\lambda$ is an instance of $(F \vee H)\theta$ (see example 4 to follow), namely the instance obtained by applying substitution ρ. We should also show that $(F \vee H)\theta$ is not a tautology: this follows as the ground resolvent is not a tautology; nor subsumed by any other clause in S: this follows because there are no failure nodes in the branch above the one being considered.

(Since any instance of a tautology is a tautology, and no tautology is ever falsified, no tautology labels a failure node, and no resolvent generated by the tree argument is a tautology. Moreover, if a resolvent were subsumed by some clause in S, that clause could already label the node falsified by the resolvent.)

From a ground semantic tree one can apply the lifting lemma to each resolvent found in the processing of the tree. The general resolvent will be falsified in the reduced tree. In the last step the two clauses labelling the left and right branches will each be factorable to a single literal.

Given two general clauses there may be several resolvents. This is in contrast with the ground case, where there is only one (check that you see why the merging imposes this). Figures 5 and 6 illustrate the lifting lemma, in which $\neg S(x) \vee \neg S(a) \vee R(z)$ is resolved with $S(y) \vee P(y)$ to obtain $R(z) \vee P(a)$.

(The resolvents $\neg S(a) \vee R(z) \vee P(y)$, with $\theta = \{x == y\}$ and $\neg S(x) \vee R(z) \vee P(a)$ with $\theta = \{y == a\}$ can also be obtained, but their generation does not involve implicit factoring of $\neg S(x)$ and $\neg S(a)$.)

Note that the completeness proof only needs to show the existence of a refutation by resolution. There may be refutations other than those given by particular semantic trees. Also, other resolvents may not be relevant to $\Box$ being generated.

Example 4

(1) $\neg S(x) \vee \neg S(a) \vee R(z)$
(2) $\neg R(b)$
(3) $S(y) \vee P(y)$
(4) $\neg P(a)$

If we apply the alternative step in the above illustration to obtain $\neg S(a) \vee R(z) \vee P(y)$, we can still derive $\Box$.

However, in the next example there are three resolvents involving clauses (1) and (3): $R(a,a)$, $P(y) \vee R(a,y)$ and $P(x) \vee R(x,a)$, but only the latter leads to $\Box$.

(1) $P(x) \vee P(y) \vee R(x,y)$
(2) $\neg R(a,b)$
(3) $\neg P(a)$
(4) $\neg P(b)$

The above examples illustrate a difficulty when working with the resolution rule, namely, in deciding when to apply the factoring implicit in forming the resolvent. In fact, factoring is usually separated from resolution, each step in a derivation being either a binary resolution step, or a factoring step. For instance, in Example 4 we could either binary resolve (1) and (3) to derive $P(y) \vee R(a,y)$ and then do so again with (3) to derive $R(a,a)$, or we could factor (1) to obtain $P(x) \vee R(x,x)$ and then binary resolve with (3) to obtain $R(a,a)$.

The control of factoring is an important feature of some other refinements of resolution. If factoring results in a clause that subsumes the original clause then we call the operation *safe factoring*, as the original clause can be removed by subsumption. Performing such safe factoring reduces the size of the clauses and is a good thing. On the other hand, factors which do not subsume the original clauses can only be added to the clauseset, so making it larger.

6 Glossary

In this section are included many definitions of terms used in the preceeding sections. It is assumed that the reader is at least acquainted with the notions of model, first order sentences and unification. If not, a good introductory text is [Bun 83].

Signature A *signature* Σ defines a *language*. It is a triple $< C, F, P >$ of constants, C, of function symbols, F and predicates P. The basic formulas of the language, called *atoms*, are formed from the elements in the signature in the usual way. A signature is sometimes restricted to the constants, function symbols and predicates occurring in some sentences S and is then called the signature of S.

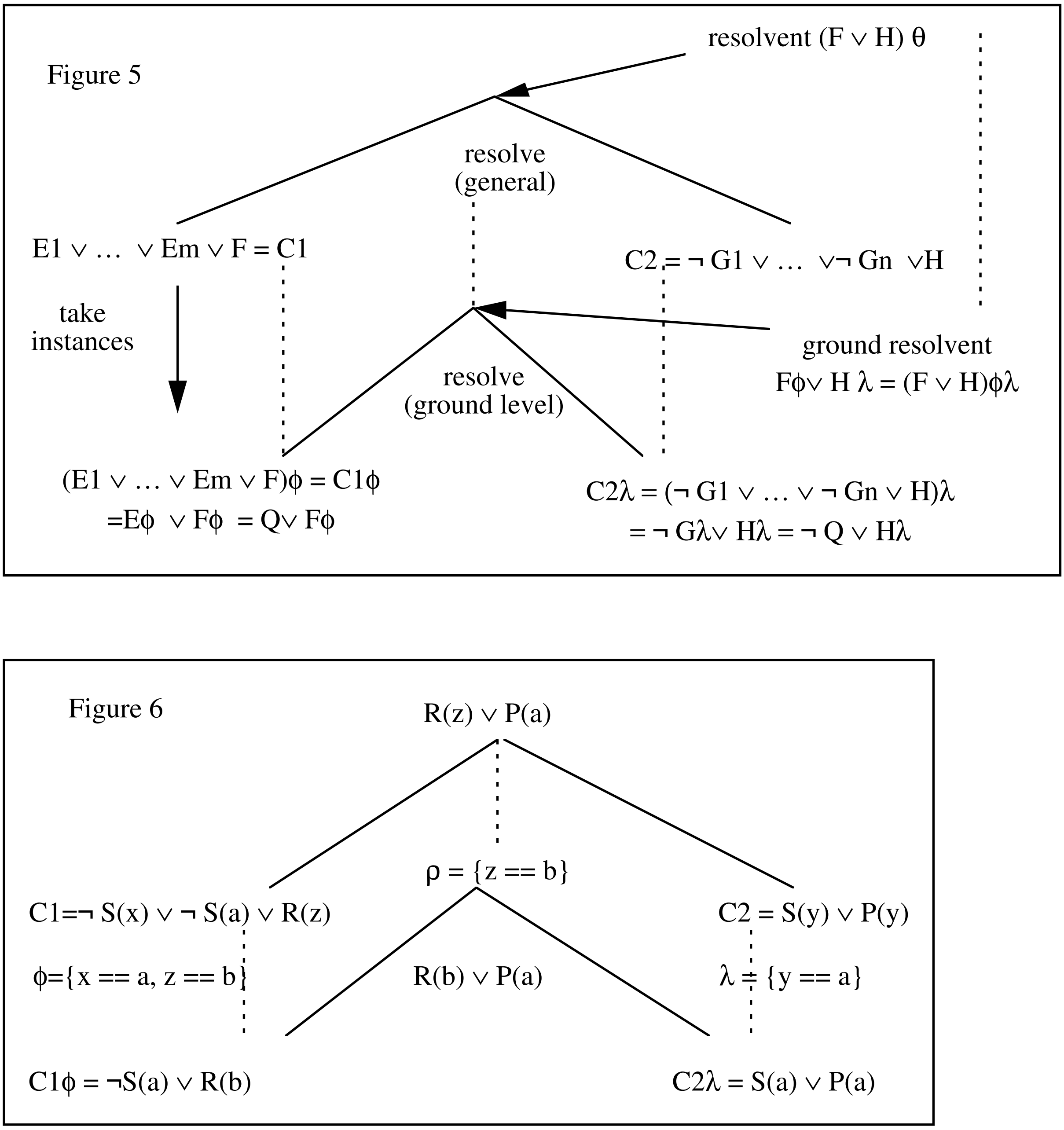

Figure 2: Figures 5 and 6

(Oftten, when we consider a set of sentences or clauses S, the signature will be implicit.)

$\models$

Given a signature Σ and $\{A_i\}, \{B_j\}$ written in Σ, $(A_1, A_2, \ldots, A_n \models B_1, \ldots, B_m)$ holds iff, whenever $A_1, \ldots, A_n$ are all true in some interpretation over Σ, then one of $B_1, \ldots, B_m$ is true also in that interpretation (A_i, B_j are first order formulae). Often $m = 1$.

$\square$

Empty clause, or contradiction, *i.e.* always false in an interpretation. Formed by resolving A with $\neg A$.

Clause First order formula of the form $L_i \vee \ldots \vee L_n$, where each L_i is a literal, and if x_i are the variables in L_i they are assumed to be universally quantified. Clauses can be equivalently written using $\leftarrow$ or 'if' .

(*e.g.* $P(x,y) \vee \neg Q(x) \vee S(x,z)$ becomes $P(x,y) \leftarrow Q(x) \wedge \neg S(x,z)$, or $\neg Q(x) \leftarrow \neg P(x,y) \wedge \neg S(x,z)$, or $P(x,y) \vee S(x,z) \leftarrow Q(x)$, etc.)
A clause is often identified with the bag of literals in it. *e.g.* the above clause might also be written as $\{P(x,y), \neg Q(x), S(x,z)\}$.

Literal An atom or its negation.

Atom $P(t_i \ldots t_n)$ where t_i are terms (from the signature) and P is a predicate (relation name).

Term a constant, a variable, or $f(t_i \ldots t_n)$, where t_i are terms and f is an n-ary function symbol.

Conjunctive Normal Form (of a formula) A formula $\forall x_i [D_1 \wedge \ldots \wedge D_k]$, where each D_i is a disjunction of literals (*i.e.* a clause) is in *conjunctive normal form.* It is equivalent to $\forall x_i.D_1 \wedge \ldots \wedge \forall x_i.D_k$. The variables in each prefix are restricted to those occurring in D_i. They are also often consistently renamed, so each prefix uses a different set of variables. For example,
$\forall x,y,z(P(x,y) \wedge P(y,z)) \equiv \forall x,y,z.P(x,y) \wedge \forall x,y,z.P(y,z)$
$\equiv \forall x,y.P(x,y) \wedge \forall y,z.P(y,z) \equiv \forall x,y.P(x,y) \wedge \forall w,z.P(w,z)$.

Tautological clause Clause of the form $L_i \vee \neg L_i \vee \ldots \vee L_n$.

Skolem Function/Constant Introduced when existential quantifiers are removed.

Substitution An assignment to a finite set of variables (written as a finite set $\{x_i == t_i\}$, *e.g.* $\{x == a, y == x, z == f(a)\}$ meaning a is to be assigned to x, x is to be assigned to y, etc. (and sometimes written as $\{a/x, x/y, f(a)/z\}$, when a/x is read as a for x, or sometimes as $\{x/a, y/x, z/f(a)\}$, when x/a is read as x is replaced by

a). In a substitution θ, the x_i are unique (and are called the $vars(\theta)$), no $x_i = t_i$ and no x_i occurs in t_i.
A substitution θ can be *applied* to a term, atom or clause, written as $t\theta$, $P\theta$ or $C\theta$ respectively, by simultaneously replacing occurrences of variables in $vars(\theta)$ by the corresponding terms given by θ.
If no variable in $vars(\theta)$ occurs in any t_j then θ is idempotent.
An *idempotent* substitution θ satisfies $(X\theta)\theta = X\theta$. We will usually restrict ourselves to idempotent substitutions.
Two substitutions $\theta = \{x_i == t_i\}$ and $\sigma = \{y_j == s_j\}$ can be *composed*, by forming the substitution $\theta\sigma = \{x_i == t_i\sigma\} \cup \{y_j == s_j\}$, where no $x_i = t_i\sigma$, no x_i occurs in $t_i\sigma$ (the former are removed, the latter cause failure of composition) and $x_i \cap y_j = \emptyset$ (any assignments for $y_j \in \{x_i\}$ are omitted).
e.g. the composition of (non-idempotent) $\sigma = \{x == a, y == x, z == f(y)\}$ with itself, $\sigma\sigma$, is $\{x == a, y == a, z == f(x)\}$.
For substitutions σ and θ, $\sigma\theta$ satisfies $X(\sigma\theta) = (X\sigma)\theta$ for any formula X; $\theta\sigma$ is not usually the same as $\sigma\theta$.
Two idempotent substitutions θ and ρ can be *combined* to give a third idempotent substitution $\theta\rho$ if the equations for θ and ρ are unified.
(Note, this is not the same as *composition* of substitutions if the same variable x occurs on the L.H.S. of an equation in both θ and ρ, for the composition retains only the substitution for x in θ in $\theta\rho$ and of x in ρ in $\rho\theta$. However, it is the same if that is not the case and the composition is also idempotent.)

Unsatisfiable A set of clauses is *satisfiable* if there is a model for it and unsatisfiable otherwise. (Unsatisfiable and inconsistent are used interchangeably, as are consistent and satisfiable.)

Subsumes Assume C and D are variable disjoint. C subsumes D (C and D clauses), if $C\theta \subseteq D$, where θ is a substitution for the variables in C. This is properly called θ -subsumption, although that is usually what is meant by subsumption. $C\theta$ means the instance of clause C obtained by applying the substitutions given by θ. If $C\theta$ has no variables it is called a *ground instance* of C. Also, more generally, if $\forall C \models \forall D$ then C subsumes D, where $\forall C$ is the universal closure of C.
Usually, we even use *strict* θ-subumption, in which each literal in the subsuming clause is used to subsume a unique literal in the subsumed clause, so the number of literals in D is $\geq$ than the number of literals in C.
Notice that subsumption is defined for any sentences, whereas θ-subumption is only defined for clauses. The latter implies subsumption, but not the other way round, as in θ-subsumption there is just one instance of C involved, whereas in subsumption C may be used more than once in deriving D.
(*e.g.* $P(a, x) \vee P(y, b)$ subsumes, but does not strictly subsume, $P(a, b)$.
$\forall x[P(x) \rightarrow P(f(x))]$ subsumes, but does not θ-subsume, $\forall x[P(x) \rightarrow [(f(f(x)))]$.)

Herbrand Universe for Σ Set of terms formed from function and constant symbols in

Σ. (If there are no constants in Σ then add an arbitrary constant a.)

Herbrand Base (of Σ) Set of ground atoms formed from predicate symbols in Σ and terms from the Herbrand Universe of Σ.

Herbrand Interpretation (of Σ) Interpretation of Σ over the Herbrand Universe of Σ, such that all constants are interpreted as themselves[3], and if f is an n-ary function symbol it is interpreted as a function which maps $(h_1, \ldots, h_n)$ to $f(h_1, \ldots, h_n)$. Each variable free atom can be assigned true or false. Thus an H-Interpretation can be represented by a subset of the Herbrand Base – an atom is assigned true by the interpretation iff it belongs to the subset representing the interpretation.

Ground clause instance A clause instance in which some ground term has been substituted for each variable.

Clause instance Result of applying a substitution to a clause, written $C\theta$. If the substitution is simply a renaming of variables, that is each t_i in θ is a variable, the new instance is called a variant of the original clause.

Standardised apart Two clauses with no variables in common are standardised apart. They are also called *variable disjoint*.

Factor A *factor* of a clause $L_i \vee \ldots \vee L_n$ is an instance of the clause in which two or more literals have been *merged*; *i.e.* under the instance forming substitution two (or more) literals become identical and can be combined (as in $A \vee A \vee B \equiv A \vee B$). (In some places a clause is considered a factor of itself).

Safe Factor A factor of C which subsumes C. (*e.g.* $\{P(a, y), R(x), R(a)\}$ factors to $\{P(a, y), R(a)\}$ which subsumes it.)

Semantic tree (for Σ) (enumeration of all Herbrand Interpretations of Σ) is a tree whose arcs are labelled with elements from the Herbrand Base of Σ (and their negations). If the finite number of arcs emanating from a node are labelled with $N_1 \ldots N_n$ then $N_1 \vee N_2 \vee \ldots \vee N_n$ is a tautology, and for each node x, the labels on the path from the root to the node do not include a complementary pair (*i.e.* A, $\neg A$ for any atom A).
A tree is complete if each branch contains A_i or $\neg A_i$ for every $A_i \in$ Herbrand Base. Such a tree will include all interpretations in its branches. The simplest form of tautology used (and the usual one) is $A \vee \neg A$, but others can also be useful. *e.g.* $A \vee B \vee (\neg A \wedge \neg B)$. A branch *falsifies* an instance $L_1 \vee \ldots \vee L_n$ of a clause in S if it includes the literals $\neg L_1 \ldots \neg L_n$ as labels on its arcs. The node at the end of a branch which falsifies some clause instance is called a *failure node* and is such that no higher node in the tree falsifies that instance. A semantic tree whose branches all end in failure nodes is *closed*.

[3]Strictly, as names of themselves.

For an unsatisfiable set S, since no H-interperation satisfies S, each branch must falsify some clause instance. Such an instance is finite and thus is falsified in the branch, after a finite number of arcs, at a failure node. (Such a branch covers an infinite number of interpretations if the HB is infinite.)
An *inference node* (in a semantic tree) is a node directly above two failure nodes.

Ground resolvent If $E \vee F$, $\neg E \vee H$ are two ground clauses then the *resolvent* is $F \vee H$. In any semantic tree the resolvent of the clause instances at failure nodes is falsified at the inference node above the two failure nodes (and perhaps higher in the tree).

Unifier A *unifier* θ of a set of literals L_k is a substitution which, when applied to the literals, results in the same literal in each case;
i.e. θ unifies $\{L_1 \ldots L_n\}$ if $L_1\theta = L_2\theta = \ldots = L_n\theta$.

Most general unifier (mgu) A unifier θ, such that for any other unifier σ of the set of literals unified by θ, $\theta\rho = \sigma$ for some third unifier ρ. (It is the minimal substitution needed for matching the literals.)

Soundness of a refutation procedure If S_1 can be derived from S by the rules of the procedure then $S \models S_1$ *i.e.* $S \cup \{S_1\}$ is unsatisfiable iff S is unsatisfiable.

Completeness of a refutation procedure If S is unsatisfiable then $\Box$ can be derived from S using the rules of procedure.

Refutation search space Graph of all possible derivations from initial clauses using rules of the refutation procedure.

Refutation A path in a refutation search space that ends with the derivation of $\Box$.

References

[TabHand 99] M. D'Agostino, D. Gabbay, R. Hahnle and J. Possega (Eds), Handbook of Tableaux Methods, Kluwer Academic Publishers, (1999)

[D'AgMon 94] M. D'Agostino and M. Mondadori, The Taming of the Cut, JLC, Vol 4, No 3, (1994)

[Bib 83] W. Bibel, Automated Theorem Proving, Vieweg (1983)

[BlasBurk 89] K. Blasius and H-J. Burckert (Eds), Deduction Systems in Artificial Intelligence, Ellis Horwood (1989)

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

[CADE] CADE 4-11 (Conference on Automated Deduction), Lecture Notes in Comp. Science, Springer

[ChLee 73] C. Chang and R. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic press (1973)

[Duffy 91] D. Duffy, Principles of Automated Theorem Proving, Wiley (1991)

[Fitt 96] M. Fitting, First Order Logiv and Automated Theorem Proving, Springer, (1996)

[LassPlot 92] J-L. Lassez and G. Plotkin (Eds), Computational Logic (1992)

[LeePlais 92] S-J Lee and D. Plaistead, Eliminating Duplication with the Hyper-Linking Strategy, J. of Automated Reas. Vol. 9, No. 1 (1992)

[MartMont 82] A. Martelli and U. Montanari, An Efficient Unification Algorithm, ACM Transactions on Programming Lang. and Systems, Vol. 4, No. 2 (1982)

[Praw 70] D. Prawitz, A proof Procedure with Matrix Reduction , Lecture Notes in Mathematics (1970)

[Rob 65] A. Robinson, A Machine Oriented Logic based on the Resolution Principle, JACM, Vol. 12, No. 1 (1965)

[Rob 79] A. Robinson, Logic: Form and Function, Edinburgh Press (1979)

[SiekWright 83] J. Siekmann and G. Wrightson (Eds), Automation of Reasoning Vol I. and Vol. II, Springer (1983)

[Wall 90] L. Wallen, Automated Deduction in Non-classical Logics, MIT Press (1990)

[Wos 85] L. Wos et al, An Overview of Automated Deduction , Journal of Automated Reasoning (JAR), (1985)

[Wos 88] L. Wos, Automated Reasoning, 33 Basic Research Problems, Prentice Hall (1988)

[Wos 92] L. Wos et al, Automated Reasoning: Introduction and Applications (2nd Ed), Mc.Graw Hill (1992)