Theorem Proving

Lectured by : Sonia Berman Date : 28-4-86 to 23-5-86

References : "Symbolic Logic & Mechanical Theorem Proving" by Chang & Lee, Academic Press 1973
"Mathematical Theory of Computation" by Manna

Contents :

1 Propositional Logic.

1.1 Introduction

A proposition is a declarative sentence that is either true or false, but not both. The symbols P, Q, R, ... that are used to denote p's are atomic formulae, or atoms.

Compound propositions can be built from propositions using logical connectives $\sim \wedge \vee \rightarrow \leftrightarrow$

Well-formed formulas* (wff's) are defined recursively by

- at atom is a wff
- if G is a wff, then $(\sim G)$ is a wff
- if G, H are wff's, then $(G \wedge H)$ $(G \vee H)$, $(G \rightarrow H)$ and $(G \leftrightarrow H)$ are wff's
- nothing else is a wff.

The precedence of the connectives is $\sim$ $\wedge \vee$ $\rightarrow$ $\leftrightarrow$ (greatest → least)

The values are:

G	H	$\sim G$	$G \wedge H$	$G \vee H$	$G \rightarrow H$	$G \leftrightarrow H$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

1.2 Interpretations of Formula in Prop. Calc.

Defn Given a prop. formula G let $A_1, A_2, \ldots, A_n$ be the atoms in G. An interpretation of G is an assignment of truth values to $A_1, \ldots, A_n$ in which every A_i is assigned a particular value T or F.

* wff anf formula are used interchangeably throughout

D: A formula G is true under (or in) an interpretation iff G is evaluated to T in the interpretation; otherwise G is false under the interpretation

There are 2^n distinct interpretations of a formula containing n distinct atoms. If the atoms in a wff are $A_1, A_2, \ldots, A_n$ (~~it can be ordered easily~~), we can represent an interpretation by a set $\{m_1, \ldots, m_n\}$, where m_i is either A_i or $\sim A_i$.

Eg $\{P, \sim Q, \sim R, S\}$ represents the interpretation with P and S true, and Q and R false.

1.3 Validity & Inconsistency in Prop. Calc

D: A wff is valid iff it is true under all its interpretations, else it is invalid (also called a tautology)

A wff is inconsistent iff it is false under all its interpretations, else it is consistent (satisfiable) (also called a contradiction or unsatisfiable)

Note A wff is valid $\Leftrightarrow$ its negation is unsatisfiable

A wff is unsatisfiable $\Leftrightarrow$ its negation is valid

Valid $\Rightarrow$ consistent

Inconsistent $\Rightarrow$ invalid

wff f is invalid $\Rightarrow \exists I : f(I) = F$ (My notation)

wff f is consistent $\Rightarrow \exists I : f(I) = T$.

If a formula F is true (false) under an interpretation I, we say that I satisfies F, or F is satisfied by I. (falsifies, is falsified by)

If I satisfies F, I is called a model of F

1.4 Normal Forms in Prop. Logic.

It is often necessary to transform a wff from one form to another, especially to a normal form. This is accomplished by replacing a formula in the given formula by a formula equivalent to it.

D: Two formulas F and G are <u>equivalent</u> (written $F = G$), iff the truth values of F and G are the same under every interpretation of F and G.

We need an adequate supply of equivalent formulas in order to carry out the transformation of formulas. Let ■ denote the formula that is always true, and □ the formula that is always false. Using these, we have the following laws. (proof by truth tables)

① $G \leftrightarrow H = (G \rightarrow H) \wedge (H \rightarrow G)$

② $G \rightarrow H = \sim G \vee H$

③ <u>Commutative</u>:

(a) $G \vee H = H \vee G$ (b) $G \wedge H = H \wedge G$

④ <u>Associative</u>

(a) $(G \vee H) \vee I = G \vee (H \vee I)$ (b) $(G \wedge H) \wedge I = G \wedge (H \wedge I)$

⑤ <u>Distributive</u>

(a) $G \vee (H \wedge I) = (G \vee H) \wedge (G \vee I)$ (b) $G \wedge (H \vee I) = (G \wedge H) \vee (G \wedge I)$

⑥ (a) $G \vee \square = G$ (b) $G \wedge ■ = G$

⑦ (a) $G \vee ■ = ■$ (b) $G \wedge \square = \square$

⑧ (a) $G \vee \sim G = ■$ (b) $G \wedge \sim G = \square$

⑨ $\sim(\sim G) = G$

⑩ <u>De Morgan</u>

(a) $\sim(G \vee H) = \sim G \wedge \sim H$ (b) $\sim(G \wedge H) = \sim G \vee \sim H$.

D: a <u>literal</u> is an atom or the negation of an atom.

D: A wff F is in conjunctive normal form iff F has the form $F \triangleq F_1 \wedge F_2 \wedge \ldots \wedge F_n$, $n \geq 1$, where each of the $F_1, \ldots, F_n$ is a disjunction of literals.

D: A wff F is in disjunctive normal form iff F has the form $F \triangleq F_1 \vee F_2 \vee \ldots \vee F_n$, $n \geq 1$, where each of the $F_1, \ldots, F_n$ is a conjunction of literals.

The following transformation procedure can be used to get a formula in normal form:

Step 1: Use laws ① & ② to eliminate $\rightarrow$ and $\leftrightarrow$

Step 2: Repeatedly use rules ⑨ & ⑩ to bring the negation signs immediately before the atoms.

Step 3: Repeatedly use the distributive laws ⑤ and the other laws to obtain a normal form.

Exercise: Obtain a disjunctive normal form for $(P \vee \neg Q) \rightarrow R$, and a conjunctive normal form for $(P \wedge (Q \rightarrow R)) \rightarrow S$

Answers: $(\neg P \wedge Q) \vee R$; $(S \vee \neg P \vee Q) \wedge (S \vee \neg P \vee \neg R)$

1.5 LOGICAL CONSEQUENCES.

D: Given formulas $F_1, \ldots, F_n$ and a wff G, G is a logical consequence of $F_1, \ldots, F_n$ iff for any interpretation I in which $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is true, G is also true. $F_1, F_2, \ldots, F_n$ are axioms (or postulates, premises) of G.

Thm 1.1 (Deduction Theorem): Given formulas $F_1, \ldots, F_n$ and a wff G, G is a logical consequence of $F_1, \ldots, F_n$ iff the formula $((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \rightarrow G)$ is valid

PROOF: ($\Rightarrow$) Suppose G is a logical consequence of $F_1 \dots F_n$. Let I be an arbitrary interpretation. If $F_1 \dots F_n$ are true in I, then, by the definition of logical consequence, G is true in I. Hence $((F_1 \wedge F_2 \wedge \dots \wedge F_n) \rightarrow G)$ is true in I.

Suppose, however, that $F_1, \dots, F_n$ are false in I, then $((F_1 \wedge \dots \wedge F_n) \rightarrow G)$ is true in I. Thus $(F_1 \wedge \dots \wedge F_n) \rightarrow G)$ is valid.

($\Leftarrow$) Suppose $((F_1 \wedge \dots \wedge F_n) \rightarrow G$ is valid. For any interpretation I, if $(F_1 \wedge \dots \wedge F_n)$ is true in I, G must be true in I. Thus G is a logical consequence of $F_1, \dots, F_n$.

Thm 1.2 Given formulas $F_1, \dots, F_n$ and a formula G, G is a logical consequence of $F_1, \dots, F_n$ iff the formula $(F_1 \wedge \dots \wedge F_n \wedge \sim G)$ is inconsistent

If G is a logical consequence of $F_1 \dots F_n$, the formula $((F_1 \wedge \dots \wedge F_n) \rightarrow G)$ is called a theorem, and G is called the conclusion of the theorem.

Exercises

① Given that • stock prices go down when interest goes up.
• most people are unhappy when stock prices go down
• interest rates are going up

Deduce that most people are unhappy.

② Given that: • if congress refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns

Will the strike be over if congress refuses to act and the strike starts?

Suppose congress refuses to act, the strike is over, and the president of the firm did not resign. Has the strike lasted more than a year?

③ Suppose we can perform the following chemical reactions:

$$MgO + H_2 \rightarrow Mg + H_2O$$

$$C + O_2 \rightarrow CO_2$$

$$CO_2 + H_2O \rightarrow H_2CO_3$$

Suppose we have some MgO, H_2, O_2 and C. Show that we can make H_2CO_3.

2 The First-Order Logic (Predicate Calculus)

2.1 The first-order logic introduces three more logical notions: terms, predicates and quantifiers

D: Terms are defined recursively as
- a constant is a term (eg John, Mary, 3)
- a variable is a term (eg x, y, z, x_1, x_2, ...)
- if f is an n-place function symbol, and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term
- nothing else is a term

A predicate is a mapping from a list of constants to T or F.

D: If P is an n-place predicate symbol, and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atom.

D: An occurrence of a variable in a formula is bound iff the occurrence is within the scope of a quantifier employing the variable, or is the occurrence in that quantifier. An occurrence of a variable in a formula is free iff the occurrence is not bound.

D: A variable is free in a formula if at least one occurrence of it is free in the formula. A variable is bound in a formula if at least one occurrence of it is bound.

Eg in $(\forall x)\ P(x, y)$ x is bound, y is free
in $(\forall x)\ P(x, y)) \wedge (\forall y)\ Q(y)$, y is both free and bound.

D: A wff in first order logic is defined recursively as:
- an atom is a formula
- if F, G are wff's, then $(\sim F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ are wffs
- if F is a wff and x is a free variable in x, then $(\forall x)F$ and $(\exists x)F$ are wffs
- nothing else is a formula; all formulae are generated by formulas are generated by a finite number of a finite number of applications of these rules

Exercises: ① Symbolise the following:

- every rational is a real
- there exists a prime number
- for every number x, there exists a number y such that $x < y$

② Symbolise the following by a formula:

- Every man is mortal
- Confucius is a man
- Therefore, Confucius is mortal.

③ Symbolise the following axioms of $\mathbb{N}$.

- for every number, there is one and only one immediate successor.
- there is no number for which 0 is the immediate successor.
- for every number other than 0, there is one and only one predecessor.

2.2 Interpretations of Formulae in FOL.

D: An interpretation of a formula F (over a domain D) in the first order logic consists of a non-empty domain D, and an assignment of values to each constant, function symbol and predicate symbol occurring in F as follows:

- to each constant, we assign an elt in D.
- to each n-place function symbol, we assign a mapping from D^n to D.
- to each n-place predicate symbol, we assign a mapping from D^n to $\{T, F\}$.

For every interpretation of a formula over a domain D, the formula can be evaluated to T or F using the following

rules:

- if the truth values of formulas G & H are evaluated, then the truth values of the formulas $\sim G$, $(G \wedge H)$, $(G \vee H)$, $(G \to H)$ and $(G \leftrightarrow H)$ are evaluated by the usual means.
- $(\forall x)\, G$ is evaluated to T if the truth value of G is evaluated to T for every x in D; otherwise F.
- $(\exists x)\, G$ is evaluated to T if the truth value of G is T for at least one elt x in D; otherwise F.

Note that any formula containing free variables cannot be evaluated. We will in general assume that formulas either do not contain free variables, or that they may be treated as constants.

Example Consider the formula $G: (\forall x)(P(x) \to Q(f(x), a))$

There is one constant a, a one-place function f, a one-place predicate P, and a two-place predicate Q in G.

The following is an interpretation I of G:

Domain $D = \{1, 2\}$

Assignment for a: 2

Assignment for f: $f(1) = 2$, $f(2) = 1$

Assignment for P and Q: $P(1) = F$ $\quad Q(1,1) = T$ $\quad Q(2,1) = F$

$P(2) = T$ $\quad Q(1,2) = T$ $\quad Q(2,2) = T$

Exercise Show that G is true under I.

Evaluate the truth values of the following formulas under the interpretation I above:

- $(\exists x)(P(f(x)) \wedge Q(x, f(a)))$ $\quad (T)$
- $(\exists x)(P(x) \wedge Q(x, a))$ $\quad (F)$
- $(\forall x)(\exists y)(P(x) \wedge Q(x, y))$ $\quad (F)$

We can now define analogously:

D: A formula G is <u>consistent</u> (<u>inconsistent</u>) iff there (does not) exists an interpretation I such that G is evaluated to T in I. We say I is a model of G, and I satisfies G.

D: A formula G is <u>valid</u> iff every interpretation of G satisfies G.

D: A formula G is a <u>logical consequence</u> of formulas $F_1, F_2, \ldots, F_n$ iff for every interpretation I, if $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is true in I, G is also true in I

The first-order logic can be considered as an extension of the propositional logic. When a formula in the FO logic contains no variables and quantifiers, it can be treated as a formula of prop. logic.
Since there is an infinite number of domains, there are in general an infinite number of interpretations of a formula. Therefore, unlike in prop logic, it is not possible to verify a valid or inconsistent formula by evaluating the formula under all possible interpretations.

<u>Exercises</u> Prove the following:

- $(\forall x)\ P(x) \wedge (\exists y) \sim P(y)$ is inconsistent
- $(\forall x)\ P(x) \rightarrow (\exists y)\ P(y)$ is valid
- $P(a) \rightarrow ((\exists x)\ P(x))$ is consistent
- $(\forall x)\ P(x) \vee ((\exists y) \sim P(y))$ is valid

2.3 Prenex Normal Forms in FOL.

D: A formula F in the first-order logic is said to be in a prenex normal form iff F is in the form of

$$(Q_1 x_1) \ldots (Q_n x_n)(M)$$

where every $(Q_i x_i)$, $i = 1, 2, \ldots, n$ is either $(\forall x_i)$ or $(\exists x_i)$ and M is a formula containing no quantifiers.
$(Q_1 x_1) \ldots (Q_n x_n)$ is called the prefix and M the matrix of F.

Laws for Quantifiers (extension to def of equivalence)

① $(Qx)\ F(x) \vee G = (Qx)(F(x) \vee G)$
② $(Qx)\ F(x) \wedge G = (Qx)(F(x) \wedge G)$
} where G does not contain any occurrences of x

③ $(\forall x)\ F(x) \wedge (\forall x)\ G(x) = (\forall x)[F(x) \wedge G(x)]$
④ $(\exists x)\ F(x) \vee (\exists x)\ G(x) = (\exists x)[F(x) \vee G(x)]$
⑤ $(Q_1 x) F(x) \vee (Q_2 x)\ G(x) = (Q_1 x)(Q_2 x)[F(x) \vee G(x)]$
⑥ $(Q_1 x)\ F(x) \wedge (Q_2 x)\ G(x) = (Q_1 x)(Q_2 x)[F(x) \wedge G(x)]$

De Morgan
$$\sim(\forall x)\ P(x) = (\exists x) \sim P(x)$$
$$\sim(\exists x)\ P(x) = (\forall x) \sim P(x).$$

Transforming WFF's into Prenex Normal Form

Step 1: Use $F \leftrightarrow G = (F \rightarrow G) \wedge (G \rightarrow F)$
$F \rightarrow G = \sim F \vee G.$

to eliminate $\leftrightarrow$ and $\rightarrow$.

Step 2: Repeatedly use $\sim(\sim F) = F$ and De Morgan's laws to bring the negation signs immediately before the atoms.

Step 3: Rename bound variables if necessary

Step 4: Use the laws ①-⑥ to move the quantifiers to the left of the entire formula to obtain a prenex normal form.

Example $(\forall x)\, P(x) \rightarrow (\exists x)\, Q(x)$

$= \sim ((\forall x) P(x)) \vee (\exists x) Q(x)$

$= (\exists x)(\sim P(x)) \vee (\exists x) Q(x)$

$= (\exists x)(\sim P(x) \vee Q(x))$.

Exercise: $(\forall x)(\forall y)((\exists z)(P(x,z) \wedge P(y,z)) \rightarrow (\exists u)\, Q(x,y,u))$

2.4 APPLICATIONS OF FOL.

Exercises:

① Show that if Confucius is a man, and all men are mortal, the Confucius is mortal.

② No used-car dealer buys a used car for his family. Some people who buy used cars for their families are absolutely dishonest. Conclude that some absolutely dishonest people are not used-car-dealers.

③ Show that if some people like all doctors but no patient likes a quack, then no doctor is a quack.

A demonstration that a conclusion follows from axioms is called a proof. A procedure for finding proofs is called a proof procedure.

3 HERBRAND'S THEOREM.

3.1 Herbrand (1930) developed an algorithm to find an interpretation that falsifies a given formula. If the given formula is valid, no such interpretation can exist and his algorithm will halt after a finite number of trials.

3.2 SKOLEM STANDARD FORMS.

The proof procedures we discuss are actually refutation procedures (proving a formula inconsistent; and thus its negation valid). These procedures apply to a "standard form" of a formula, based on the following ideas.

- a formula can be transformed into prenex normal form
- the matrix, since it contains no quantifiers, can be transformed into a conjunctive normal form.
- without affecting the inconsistency property, the existential quantifiers in the prefix can be eliminated by using Skolem functions.

We have already shown how to perform the first two transformations; we now consider eliminating existential quantifiers.

Let a wff F be in prenex normal form $(Q_1 x_1)\ldots(Q_n x_n)M$ where M is in conjunctive normal form. Suppose Q_r is an existential quantifier in the prefix, $1 \le r \le n$. If no universal quantifier appears before Q_r, we choose a new constant c different from other constants occurring in M, replace all x_r in M with c, and delete $(Q_r x_r)$ from the prefix. If $Q_{s_1}\ldots Q_{s_m}$ are all the universal quantifiers appearing before Q_r, $1 \le s_1 < s_2 \ldots < s_m < r$, we choose a new m-place function symbol f different from other function

symbols, replace all x_i in M with $f(x_{s_1}, x_{s_2}, \ldots, x_{s_m})$ and delete $(Q_i x_i)$ from the prefix.
After we have dealt with all existential quantifiers in this way, the formula we obtain is a (Skolem) standard form of F. The replacement constants and functions are the Skolem functions.

D: A clause is a disjunction of literals. We customarily denote the empty clause (which is unsatisfiable) as $\square$

When it is convenient, we shall regard a set of literals as synonymous with a clause, eg $P \vee Q \vee \neg R = \{P, Q, \neg R\}$. A clause consisting of r literals is an r-literal clause.
A one-literal clause is a unit clause.
A set S of clauses is regarded as a conjunction of all clauses in S, where every variable S is considered governed by a universal quantifier.

Theorem 3.1 Let S be a set of clauses that represents a standard form of a formula F. Then F is inconsistent iff S is inconsistent.

A formula may have more than one standard form. For simplicity we would like our Skolem functions to be as elementary as possible (ie with the least number of arguments). We should thus move existential quantifiers as far to the left as possible.

Based on the theorem, it is generally the case that the input to a proof procedure consists of a set S of clauses. A formula which has been decomposed into such a set is said to be in clause form

Example (Sonia)

Convert the following wff to clause form

$$(\forall x)\{P(x) \rightarrow [(\forall y)\{P(y) \rightarrow P(f(x,y))\} \wedge \sim(\forall y)\{\sim Q(x,y) \vee P(y)\}]\}$$

① Eliminate $\rightarrow$

$$(\forall x)\{\sim P(x) \vee [(\forall y)\{\sim P(y) \vee P(f(x,y))\} \wedge \sim(\forall y)\{\sim Q(x,y) \vee P(y)\}]\}$$

② Reduce scope of $\sim$

$$(\forall x)\{\sim P(x) \vee [(\forall y)\{\sim P(y) \vee P(f(x,y))\} \wedge (\exists y)\{Q(x,y) \wedge \sim P(y)\}]\}$$

③ Standardise variables - rename variables to be unique for each quantifier.

$$(\forall x)\{\sim P(x) \vee [(\forall y)\{\sim P(y) \vee P(f(x,y))\} \wedge (\exists z)\{Q(x,z) \wedge \sim P(z)\}]\}$$

④ Eliminate $\exists$

$$(\forall x)\{\sim P(x) \vee [(\forall y)\{\sim P(y) \vee P(f(x,y))\} \wedge \{Q(x,g(x)) \wedge \sim P(g(x))\}]\}$$

⑤ Move $\forall$ left

$$(\forall x)(\forall y)\{\sim P(x) \vee [\{\sim P(y) \vee P(f(x,y))\} \wedge \{Q(x,g(x)) \wedge \sim P(g(x))\}]\}$$

⑥ Eliminate $\forall$ - implicitly understood that all variables are universally quantified

⑦ Obtain conjunctive normal form

$$\sim P(x) \vee [\{\sim P(y) \vee P(f(x,y))\} \wedge \{Q(x,g(x)) \wedge \sim P(g(x))\}]$$

$$= \{\sim P(x) \vee \{\sim P(y) \vee P(f(x,y))\}\} \wedge \{\sim P(x) \vee \{Q(x,g(x)) \wedge \sim P(g(x))\}\}$$

$$= \{\sim P(x) \vee \sim P(y) \vee P(f(x,y)\} \wedge \{\sim P(x) \vee Q(x,g(x))\} \wedge \{\sim P(x) \vee \sim P(g(x)\}$$

⑧ Make a set of clauses

$$\{\sim P(x) \vee \sim P(y) \vee P(f(x,y)),\ \sim P(x) \vee Q(x,g(x)),\ \sim P(x) \vee \sim P(g(x))\}$$

Need to prove one false to prove whole false.

3.3 The Herbrand Universe of a Set of Clauses.

By defn, a set of clauses is unsatisfiable iff it is false under every interpretation over all domains. Since it is inconvenient and impossible to consider all interpretations over all domains, we would like to find one special domain H such that S is unsatisfiable iff S is false under all interpretations over this domain. This domain is called the Herbrand universe of S

D: Let H_0 be a set of constants appearing in S. If no constant appears in S, then H_0 is to consist of a single constant, say $H_0 = \{a\}$. For $i = 0, 1, 2, \ldots$, let H_{i+1} be the union of H_i and the set of all terms of the form $f^n(t_1, \ldots, t_n)$ for all n-place functions f^n appearing in S, where the t_js are elements of H_i. Then each H_i is called the i-level constant set of S, and H_∞ is called the Herbrand universe of S.

By an expression we mean a term, a set of terms, an atom, a set of atoms, a literal, a clause, or a set of clauses. When no variable occurs in an expression we call the expression a ground expression. (Hence ground term, ground literal, etc.)

D: Let S be a set of clauses. The set of ground atoms of the form $P^n(t_1, \ldots, t_n)$ for all n-place predicates P^n occurring in S, where $t_1, \ldots, t_n$ are elements of the Herbrand universe of S, is called the atom set or Herbrand base of S.

D. A ground instance of a clause C of a set S of clauses is a clause obtained by replacing variables in C by members of the Herbrand universe of S.

Examples

① Let $S = \{P(a), \sim P(x) \vee P(f(x))\}$

Then $H_0 = \{a\}$

$H_1 = \{a, f(a)\}$

$H_2 = \{a, f(a), f(f(a)\}$ etc.

so $H = \{a, f(a), f(f(a), \ldots\}$

② Let $S = \{P(x) \vee Q(x), R(z), T(y) \vee \sim W(y)\}$. Since there is no constant in S, we let $H_0 = \{a\}$. There is no function symbol in S, hence $H = H_0 = H_1 = \ldots = \{a\}$.

③ Let $S = \{P(f(x)), a, g(y), b\}$

Then $H_0 = \{a, b\}$

$H_1 = \{a, b, f(a), f(b), g(a), g(b)\}$

$H_2 = \{a, b, f(a), f(b), g(a), g(b), f(f(a)), f(f(b)),$
$f(g(a)), f(g(b)), g(f(a)), g(f(b)), g(g(a)), g(g(b))\}$

$\vdots$

④ Let $S = \{P(x), Q(f(y)) \vee R(y)\}$. $C = P(x)$ is a clause in S and $H = \{a, f(a), f(f(a)), \ldots\}$ is the Herbrand universe of S. Then $P(a)$ and $P(f(f(a)))$ are both ground instances of C.

We now consider interpretations over the Herbrand universe. Let S be a set of clauses. An interpretation over the Herbrand universe of S is an assignment of constants, function symbols and predicate symbols occurring in S. We define a special interpretation over the Herbrand universe of S, called the H-interpretation of S

D: Let S be a set of clauses; H the Herbrand universe of S; and I an interpretation of S over H. I is said to be an <u>H-interpretation</u> of S if it satisfies the following conditions:

- I maps all constants in S to themselves.
- Let f be an n-place function symbol and $h_1, \dots, h_n$ be elements of H. In I, f is assigned a function that maps $(h_1, \dots, h_n) \in H^n$ onto $f(h_1, \dots, h_n) \in H$.

There is no restriction on the assignment to each n-place predicate symbol in S. Let $A = \{A_1, A_2, \dots, A_n, \dots\}$ be the atom set of S. An H-interpretation I can be conveniently represented by a set $I = \{m_1, m_2, \dots, m_n, \dots\}$ in which m_j is either A_j or $\sim A_j$.

<u>Example</u>: $S = \{P(x) \vee Q(x),\ R(f(y))\}$.

$H = \{a, f(a), f(f(a)), \dots\}$

$A = \{P(a), Q(a), R(a), P(f(a)), Q(f(a)), R(f(a)), \dots\}$

Some H-interpretations for S are:

$I_1 = \{P(a), Q(a), R(a), P(f(a)), Q(f(a)), R(f(a)), \dots\}$

$I_2 = \{\sim P(a), \sim Q(a), \sim R(a), \sim P(f(a)), \sim Q(f(a)), \sim R(f(a)), \dots\}$

$I_3 = \{P(a), Q(a), \sim R(a), P(f(a)), Q(f(a)), \sim R(f(a)), \dots\}$.

An interpretation I of a set S of clauses does not necessarily have to be defined over the Herbrand universe, and thus may not be an H-interpretation; however, we can construct an H-interpretation I^* corresponding to I.

D: Given an interpretation I over a domain D, an <u>H-interpretation I^* corresponding to I</u> is an H-interpretation satisfying:
Let $h_1, \dots, h_n$ be elements of H. Let every h_i be mapped to some d_i in D. If $P(d_1, \dots, d_n)$ is assigned T (F) by I,

then $P(h_1, \ldots, h_n)$ is also assigned $T(F)$ in I^*.

Lemma: If an interpretation I over some domain D satisfies a set S of clauses, then any one of the H-interpretations I^* corresponding to I also satisfies S.

Theorem: A set S of clauses is unsatisfiable iff S is false under all the H-interpretations of S.

Note:
- a ground instance C' of a clause C is satisfied by an interpretation I iff there is a ground literal L' in C' such that L' is also in I, i.e. $C' \cap I \neq \Phi$.
- a clause C is satisfied by an interpretation I iff every ground instance of C is satisfied by I.
- a clause C is falsified by an interpretation I iff there is at least one ground instance C' of C such that C' is not satisfied by I.
- a set S of clauses is unsatisfiable iff for every interpretation I, there is at least one ground instance C' of C such that C' is not satisfied by I.

Examples ① $C = \sim P(x) \vee Q(f(a))$

$I_1 \triangleq \{\sim P(a), \sim Q(a), \sim P(f(a)), \sim Q(f(a)), \ldots\}$

$I_2 \triangleq \{P(a), Q(a), P(f(a)), Q(f(a)), \ldots\}$

$I_3 \triangleq \{P(a), \sim Q(a), P(f(a)), \sim Q(f(a)), \ldots\}$

C is satisfied by I_1 & I_2 but falsified by I_3

② $S = \{P(x), \sim P(a)\}$. There are only two H-interpretations, $I_1 = \{P(a)\}$, $I_2 = \{\sim P(a)\}$

S is falsified by both, and therefore is unsatisfiable

3.4 SEMANTIC TREES

Finding a proof for a set of clauses is equivalent to generating a semantic tree.

D: If A is an atom, then A and $\neg A$ are <u>complements</u> and $\{A, \neg A\}$ is a <u>complementary pair</u>

A clause which contains a complementary pair is a tautology.

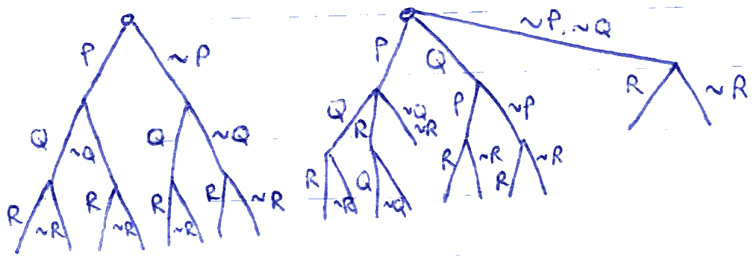
D: Given a set S of clauses, let A be the atom set of S. A <u>semantic tree</u> for S is a (downward) tree T, where each link is attached with a finite set of atoms or negations of atoms from A in such a way that:

- for each node N, there are only finitely many immediate descendants with links $L_1, \ldots, L_n$. Let Q_i be the conjunction of all the literals in the set attached to L_i, $i = 1, \ldots, n$. Then $Q_1 \vee Q_2 \vee \ldots \vee Q_n$ is a valid propositional formula
- for each node N, let $I(N)$ be the union of all the sets attached to the links of the branch of T down to and including N. Then $I(N)$ does not contain any complementary pair.

D: Let $A = \{A_1, A_2, \ldots, A_k, \ldots\}$ be the atom set of a set S of clauses. A semantic tree is said to be <u>complete</u> iff for every leaf node N of the tree, $I(N)$ contains either A_i or $\neg A_i$ for $i = 1, 2, \ldots$.

Note that for each node N in a semantic tree for S, $I(N)$ is a subset of some interpretation for S. $I(N)$ is thus called a <u>partial interpretation</u> for S.

Example: $A = \{P, Q, R\}$, the atomset of S



are both complete semantic trees of S.

$S = \{P(x), P(a)\}$ has the complete ST 

When the atomset A of a set S of clauses is infinite, any complete semantic tree for S will be infinite. Such a tree corresponds to an exhaustive survey of all possible interpretations for S. If S is unsatisfiable, then S fails to be true in each of these interpretations. Thus we may stop expanding nodes from a node N if $I(N)$ falsifies S.

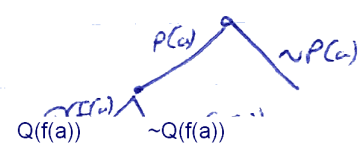
D: A node N is a failure node if $I(N)$ falsifies some ground instance of a clause in S for every ancestor node N' of N.

D: A semantic tree T is closed iff every branch of T terminates at a failure node.

D: A node N of a semantic tree is called an inference node if all the immediate descendant nodes of N are failure nodes.

Example: $S = \{P(x), \neg P(x) \vee Q(x), \neg Q(f(a))\}$

$A = \{P(a), Q(a), P(f(a)), Q(f(a)), \ldots\}$

closed semantic tree 

A semantic tree is a binary tree extending below a root node. Corresponding to the way in which truth values are assigned to the atoms p_i of a Herbrand base, a certain path may be followed down the tree. If we assign T to p_1 we branch left immediately below the root node (else right). Regardless of which node has been branched to, if p_2 is T we go left, else right. This process continues until a truth value has been assigned to each element of the Herbrand base. If the Herbrand base is infinite any complete interpretation will result in an infinite path down the tree. The complete tree containing all possible paths thus represents all possible interpretations for the clauses in S.

3.5 HERBRAND'S THEOREM

To test whether a set S of clauses is satisfiable, we need consider only interpretations over the Herbrand universe of S. If S is false under all these interpretations then we can conclude that S is unsatisfiable.

Theorem A set S of clauses is unsatisfiable if and only if corresponding to every complete semantic tree of S, there is a finite closed semantic tree.

Equivalently: a set S of clauses is unsatisfiable iff there is a finite unsatisfiable set S' of ground instances of clauses of S.

PROOF: ($\Rightarrow$) Suppose S is unsatisfiable. Let T be any complete semantic tree for S. For each branch B of T, let I_B be the set of all literals attached to all links of the branch B. Then I_B is an interpretation for S. Since S is unsatisfiable, I_B must falsify a ground instance C' of a clause C in S. Since C' is finite (a clause) there is a failure node N_B on branch B within finite distance of the root. As every branch of T has a failure node, there is a closed semantic tree T' for S. As only a finite number of links are connected for each node of T, T' must be finite

($\Leftarrow$) Suppose corresponding to each complete semantic tree T of S there is a finite closed semantic tree T'. Then every branch of T' contains a failure node (ie every H-interp. falsifies S *) However, for any interpretation I that satisfies S, there is an H-interpretation I^* corresponding to I satisfying S. So from * we see that S is unsatisfiable

(61)

A refutation based on the second version of Herbrands' theorem could successively generate sets $S_1', \ldots, S_n', \ldots$ of ground instances of clauses in S and successively test $S_1', S_2', \ldots$ for unsatisfiability. This would detect a finite N such that S_n' is unsatisfiable (if N exists)

(See exercises, Chang & Lee p67 ff).

4. The Resolution Principle

4.1 Introduction

Herbrand's procedure has one major drawback: it requires the generation of sets $S_1', S_2', \ldots$ of ground instances of clauses, and for most cases this sequence grows exponentially. In 1965 Robinson introduced the resolution principle which can be applied directly to any set S of (not necessarily ground) clauses to test the unsatisfiability of S.

The essential idea is to check whether S contains $\square$, in which case it is unsatisfiable. If S does not contain $\square$, we check whether $\square$ can be derived from S. By Herbrand's theorem, checking for $\square$ is equivalent to counting the nodes of a closed semantic tree T (the existence of which implies S unsatisfiable). Clearly S contains $\square$ iff T consists solely of the root node. If S does not contain $\square$, T must contain more than one node. If we can reduce the number of nodes in the tree to one, eventually $\square$ can be forced to appear (this is what resolution does). We can view resolution as an inference rule used to generate new clauses from S. If we put those new clauses into S, some nodes of the original T can be forced to become failure nodes, reducing T, and eventually $\square$ can be derived.

4.2 Resolution in Prop. Logic

The resolution principle is essentially an extension of Davis & Putnam's one-literal rule, which says:

- if there is a unit ground clause L in S, obtain S' from S by deleting those ground clauses in S containing L. If S' is empty, S is satisfiable; otherwise obtain S'' from S' by

deleting $\sim L$ from S'. S'' is unsatisfiable iff S is. Note that if $\sim L$ is a unit ground clause, then the clause becomes $\square$ when $\sim L$ is deleted.

eg $C_1 \triangleq P$ and $C_2 \triangleq \sim P \vee Q$ gives $C_3 \triangleq Q$.

Extending the above rule and applying it to any pair of clauses (not necessarily unit clauses), we have the following resolution principle:

For any two clauses C_1 and C_2, if there is a literal L_1 in C_1 complementary to a literal L_2 in C_2, then delete L_1 and L_2 from C_1 and C_2 respectively, and construct the disjunction of the remaining clauses; this constructed clause is a resolvent of C_1 and C_2.

Theorem: Given two clauses C_1 and C_2, a resolvent of C_1 and C_2 is a logical consequence of C_1 and C_2.

D: Given a set S of clauses, a (resolution) deduction of C from S is a finite sequence $C_1, C_2, \ldots, C_k$ of clauses such that each C_i is either a clause in S or a resolvent of clauses preceding C_i, and $C_k = C$. A deduction of $\square$ from S is called a refutation or proof of S.

We say that a clause C can be deduced or derived from S if there is a deduction of C from S.
The resolution principle is a very powerful inference rule, which is complete in proving the unsatisfiability of a set of clauses. (ie given a set S of clauses, S is unsatisfiable iff there is a resolution deduction of $\square$ from S.

Examples: ① $S = \{ \sim P \vee Q, \sim Q, \sim P \}$

Resolving the first two clauses we get $\sim P$

Resolving this with the 3rd clause, we get $\square$

Hence S is unsatisfiable

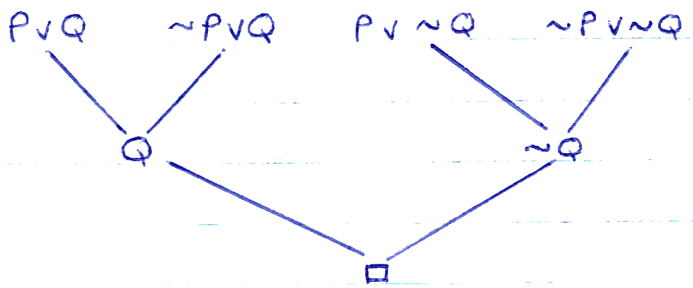
② $S = \{ \overset{①}{P \vee Q}, \overset{②}{\sim P \vee Q}, \overset{③}{P \vee \sim Q}, \overset{④}{\sim P \vee \sim Q} \}$

We have: ① & ② $\Rightarrow$ Q ⑤

③ & ④ $\Rightarrow$ $\sim Q$ ⑥

⑤ & ⑥ $\Rightarrow$ $\square$

We can represent this by a deduction tree



4.3 Substitution & Unification.

The most important part of applying the resolution principle is finding a literal in a clause complementary to a literal in another clause. For clauses containing variables, this is more difficult; we frequently have to substitute terms for variables.

D: A substitution is a finite set of the form $\{t_1/v_1, \ldots, t_n/v_n\}$ where every v_i is a variable, every t_i is a term different from v_i, and no two elements have the same variable after the /. If $t_1, \ldots, t_n$ are ground terms, the substitution is a ground substitution. The empty substitution is denoted ε.

D: Let $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ be a substitution and E be an expression. Then $E\theta$ is an expression obtained from E by replacing simultaneously each occurrence of v_i $(1 \le i \le n)$ in E by t_i. $E\theta$ is an instance of E.

D: Let $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$ and $\lambda = \{u_1/y_1, \ldots, u_m/y_m\}$ be two substitutions. Then the composition of θ and λ is the substitution $\theta \circ \lambda$ obtained from $\{t_1\lambda/x_1, \ldots, t_n\lambda/x_n, u_1/y_1, \ldots, u_m/y_m\}$ by deleting any elt $t_j\lambda/x_j$ for which $t_j\lambda = x_j$, and any elt u_i/y_i such that y_i is in $\{x_1, x_2, \ldots, x_n\}$.

$\alpha(\beta\gamma) = (\alpha\beta)\gamma$

The composition of substitutions is associative, and ε is both a left and right identity.

In the resolution proof procedure, in order to identify a complementary pair of literals, we often have to unify (match) two expressions, i.e. we must find a substitution that can make several expressions identical.

D: A substitution θ is a unifier for a set $\{E_1, \ldots, E_k\}$ iff $E_1\theta = E_2\theta = \ldots = E_k\theta$. The set $\{E_1, \ldots, E_k\}$ is unifiable if there is a unifier for it.

D: A unifier σ for a set $\{E_1, \ldots, E_k\}$ of expressions is a most general unifier iff for each unifier θ for the set, there is a substitution λ such that $\theta = \sigma \circ \lambda$.

Examples ① Let $\theta = \{t_1/x_1, t_2/x_2\} = \{f(y)/x, z/y\}$

$\lambda = \{u_1/y_1, u_2/y_2, u_3/y_3\} = \{a/x, b/y, y/z\}$

Then $\theta \circ \lambda = \{t_1\lambda/x_1, t_2\lambda/x_2, u_1/y_1, u_2/y_2, u_3/y_3\} = \{f(b)/x, y/y, a/x, b/y, y/z\}$

Since y_1 and y_2 are among $\{x_1, x_2\}$ we can delete u_1/y_1 (i.e. a/x) and u_2/y_2 (i.e. b/y).. giving $\theta \circ \lambda = \{f(b)/x, y/z\}$

② The set $\{P(a, y), P(x, f(b))\}$ is unifiable since $\theta = \{a/x, f(b)/y\}$ is a unifier for the set

THE UNIFICATION ALGORITHM

The unification algorithm will produce a most general unifier λ for any unifiable set $\{L_i\}$ of literals and will report failure if the set is not unifiable.

Consider $P(a)$ and $P(x)$. These expressions disagree in that a occurs in $P(a)$ but x in $P(x)$. In order to unify them, we must find and eliminate the disagreement (in this case, substituting a for x).

D: The <u>disagreement set</u> of a non-empty set W of expressions is obtained by locating the first symbol (from the left) at which not all the expressions in W have exactly the same symbol, and then extracting from each expression in W the subexpressions beginning at that symbol position. The set of these subexpressions forms the disagreement set.

THE ALGORITHM

① Set $k = 0$, $W_k = W$, $\sigma_k = \varepsilon$

② If W_k is a singleton, stop. σ_k is a most general unifier for W.
Else find the disagreement set D_k of W_k

③ If there are elts v_k and t_k in D_k such that v_k is a variable that does not occur in t_k, goto ④
Else stop; W is not unifiable

④ Let $\sigma_{k+1} = \sigma_k \{t_k / v_k\}$, and $W_{k+1} = W_k \{t_k / v_k\}$
(note that $W_{k+1} = W\sigma_{k+1}$)

⑤ $k = k+1$; goto ②

The algorithm will always terminate for any finite non-empty set of expressions.

Example $W = \{P(a, x, f(g(y))),\ P(z, f(z), f(u))\}$

- $W_0 = W$, $\sigma_0 = \varepsilon$
  $D_0 = \{a, z\}$
- $\sigma_1 = \sigma_0 \circ \{t_0/v_0\} = \varepsilon \circ \{a/z\} = \{a/z\}$
  $W_1 = W_0\{t_0/v_0\} = \{P(a, x, f(g(y))),\ P(a, f(a), f(u))\}$
  $D_1 = \{x, f(a)\}$

- $\sigma_2 = \sigma_1 \circ \{t_1/v_1\} = \{a/z\}, \{f(a), x\} = \{a/z, f(a)/x\}$
  $W_2 = W_1\{t_1/v_1\} = \{P(a, f(a), f(g(y))),\ P(a, f(a), f(u))\}$
  $D_2 = \{g(y), u\}$

- $\sigma_3 = \sigma_2 \circ \{g(y)/u\} = \{a/z, f(a)/x, g(y)/u\}$
  $W_3 = W_2\{t_2/v_2\} = \{P(a, f(a), f(g(y)))\}$
  Since W_3 is a singleton, σ_3 is a mgu for W.

4.5 The Resolution Principle for First-Order Logic.

D: If two or more literals (with the same sign) of a clause C have a mgu σ, then $C\sigma$ is called a factor of C. If $C\sigma$ is a unit clause, it is called a unit factor of C

D Let C_1 and C_2 be two clauses (called parent clauses) with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2 respectively. If L_1 and $\neg L_2$ have a mgu σ, then the clause $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$ is called a binary resolvent of C_1 and C_2, where L_1 and L_2 are the literals resolved upon

(NB in Sonia's notes, we choose subsets of literals with a mgu with no loss of generality. If two clauses resolve, they may have more than one resolvent as there may be more than one way of choosing the (subsets of) literals

D: A <u>resolvent</u> of (parent) clauses C_1 and C_2 is one of the following binary resolvents:

- a br of C_1 and C_2
- a br of C_1 and a factor of C_2
- a br of a factor of C_1 and C_2
- a br of a factor of C_1 and a factor of C_2.

The resolution principle is an inference rule that generates resolvents from a set of clauses. It is considerably more efficient than Herbrand's method. Furthermore, resolution is complete (i.e. it will always generate $\square$ from an unsatisfiable set of clauses.

A <u>deduction tree</u> from a set S of clauses is an (upward) tree, to each initial node of which is attached a clause in S, and to each non-initial node of which is attached a resolvent of clauses attached to its immediate predecessor nodes. We call the tree a <u>deduction tree</u> of R if R is the clause attached to the root node of the tree.

<u>Examples</u>: ① Let $C = P(x) \vee P(f(y)) \vee \sim Q(x)$. The first and second literals have a mgu $\sigma = \{f(y)/x\}$. Hence $C\sigma = P(f(y)) \vee \sim Q(f(y))$ is a factor of C.

② Let $C_1 = P(x) \vee Q(x)$ and $C_2 = \sim P(a) \vee R(x)$. Since x appears in both C_1 and C_2 we rename the variable in C_2 and let $C_2 = \sim P(a) \vee R(y)$. Choose $L_1 = P(x)$ and $L_2 = \sim P(a)$. Since $\sim L_2 = P(a)$, L_1 and $\sim L_2$ have mgu $\sigma = \{a/x\}$. Therefore $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma) = (\{P(a), Q(a)\} - \{P(a)\}) \cup (\{\sim P(a), R(y)\} - \{\sim P(a)\})$
$= \{Q(a)\} \cup \{R(y)\} = \{Q(a), R(y)\} = Q(a) \vee R(y)$, a binary resolvent of C_1 and C_2, where $P(x)$ and $\sim P(a)$ are the literals resolved upon.

③ Let $C_1 = P(x) \vee P(f(y)) \vee R(g(y))$ and $C_2 = \sim P(f(g(a))) \vee Q(b)$.
A factor of C_1 is $C_1' = P(f(y)) \vee R(g(y))$. A binary resolvent of C_1' and C_2 is $R(g(g(a))) \vee Q(b)$. Therefore, $R(g(g(a))) \vee Q(b)$ is a resolvent of C_1 and C_2.

4.6 Completeness of the Resolution Principle.

A set S of clauses is unsatisfiable iff there is a deduction of $\square$ from S. (Completeness of R.P. theorem)

4.7 Applications of the Resolution Principle.

The R.P. is a refutation process used to find proof that a wff W logically follows from a set S of wffs. This is equivalent to showing that $\{\sim W\} \cup \{S\}$ is unsatisfiable.
If a set S of clauses is unsatisfiable the semantic tree for the set is closed. We may infer new clauses from this set by resolution. These clauses, when added to S, form a new semantic tree which has fewer nodes above the failure nodes. Continuing this process results in the root node being a failure node for the empty clause.
Thus if we continue to perform resolutions on a set S of unsatisfiable clauses, $\square$ will eventually be generated. The R.P. may thus be used without explicit reference to the semantic tree.
Let $R(S)$ be the union of S with all the resolvents obtainable between pairs of clauses in S. Then by $R^2(S)$ we mean $R(R(S))$. If S is unsatisfiable, we are guaranteed that for some finite n (called the level of the refutation) the empty clause will be contained in $R^n(S)$. The generation of $R^n(S)$ corresponds to a breadth-first search.

Examples:

① The customs officials searched everyone who entered the country who was not a VIP. Some drug pushers who entered the country were only searched by drug pushers. No drug pushers was a VIP. Therefore some of the officials were drug pushers.

Let $E(x) \triangleq$ "x entered the country"
$V(x) \triangleq$ "x was a VIP"
$S(x,y) \triangleq$ "y searched x"
$C(x) \triangleq$ "x was a custom official"
$P(x) \triangleq$ "x was a drug pusher"

Then the premises are

$$(\forall x)(E(x) \wedge \sim V(x) \rightarrow (\exists y)(S(x,y) \wedge C(y)))$$
$$(\exists x)(P(x) \wedge E(x) \wedge (\forall y)(S(x,y) \rightarrow P(y)))$$
$$(\forall x)(P(x) \rightarrow \sim V(x))$$

Conclusion: $(\exists x)(P(x) \wedge C(x))$

Transforming premises into clauses we obtain:

① $\sim E(x) \vee V(x) \vee S(x, f(x))$
② $\sim E(x) \vee V(x) \vee C(f(x))$
③ $P(a)$
④ $E(a)$
⑤ $\sim S(a,y) \vee P(y)$
⑥ $\sim P(x) \vee \sim V(x)$

The negation of the conclusion:
⑦ $\sim P(x) \vee \sim C(x)$

Then

⑧ $\sim V(a)$ ③ & ⑥
⑨ $V(a) \vee C(f(a))$ ② & ④
⑩ $C(f(a))$ ⑧ & ⑨
⑪ $V(a) \vee S(a, f(a))$ ① & ④

⑫ $S(a, f(a))$ ⑧ & ⑪

⑬ $P(f(a))$ ⑫ & ⑤

⑭ $\sim C(f(a))$ ⑬ & ⑦

⑮ $\square$ ⑩ & ⑭

(2) Students are citizens. Therefore student' votes are citizens' votes.

Let $S(x) \triangleq$ "x is a student"

$C(x) \triangleq$ "x is a citizen"

$V(x,y) \triangleq$ "x is a vote of y".

Then: premise : $(\forall y)(S(y) \rightarrow C(y))$

conclusion : $(\forall x)((\exists y)(S(y) \wedge V(x,y)) \rightarrow (\exists z)(C(z) \wedge V(x,z)))$

A standard form of the premise is

① $\sim S(y) \vee C(y)$.

Since $\sim((\forall x)\ldots) = \sim((\forall x)((\forall y)(\sim S(y) \vee \sim V(x,y)) \vee (\exists z)(C(z) \wedge V(x,z))))$

$= \sim((\forall x)(\forall y)(\exists z)(\sim S(y) \vee \sim V(x,y) \vee (C(z) \wedge V(x,z))))$

$= (\exists x)(\exists y)(\forall z)(S(y) \wedge V(x,y) \wedge (\sim C(z) \vee \sim V(x,z)))$

we have three clauses for the negation of the conclusion

② $S(b)$

③ $V(a,b)$

④ $\sim C(z) \vee \sim V(a,z)$

Then

⑤ $C(b)$ ① & ②

⑥ $\sim V(a,b)$ ⑤ & ④

⑦ $\square$ ⑥ & ③.

English: "Assume that b is a student, a is b's vote, and a is not a vote of any citizen. Since b is a student, b must be a citizen. Furthermore, a must not be a vote of b because b is a citizen. This is impossible"

4.8 Deletion Strategy.

A straightforward application of the resolution principle corresponds to a simple breadth-first refutation search. Such a search would start with a set of clauses S and add to this set all resolvents between pairs of clauses in S, producing the set $R(S)$. Next $R^2(S)$ is produced, and so on. This is called the level saturation resolution method and is impractical due to the exponential expansions of the sets. Practical proof procedures depend on effective search strategies to speed up and heuristically constrain the search.

A set of clauses may be simplified by the elimination of certain clauses or certain literals in the clauses. The simplified set is unsatisfiable iff the original set is unsatisfiable

Firstly tautologies may be eliminated, since any unsatisfiable set containing a tautology is still unsatisfiable after removing it.

Secondly, in certain cases it may be possible to evaluate the truth value of literals.

- if a literal in a clause evaluates to T, the entire clause may be eliminated (cf tautologies)
- if a literal evaluates to F, the occurence of that literal in the clause can be eliminated

Thirdly, we can eliminate subsumption.

D: A clause C subsumes a clause D iff there is a substitution σ such that $C\sigma \subseteq D$; D is the subsumed clause and may be deleted without affecting unsatisfiability.

Example: Let $C = P(x)$ and $D = P(a) \vee Q(a)$. If $\sigma = \{a/x\}$ then $C\sigma = P(a)$. Since $C\sigma \in D$, C subsumes D.

The Subsumption Algorithm

Let C, D be clauses. Let $\theta = \{a_1/x_1, \ldots, a_n/x_n\}$ where $x_1, \ldots, x_n$ are variables occuring in D, and $a_1, \ldots, a_n$ are new distinct constants not occuring in C or D. Suppose $D = L_1 \vee L_2 \vee \ldots \vee L_m$

Then $D\theta = L_1\theta \vee L_2\theta \vee \ldots \vee L_m\theta$, a ground clause.

① Let $W = \{\sim L_1\theta, \sim L_2\theta, \ldots, \sim L_m\theta\}$ (the clauses of $\sim D\theta$)

② Let $k = 0$ and $\mathcal{U}^0 = \{C\}$

③ If $\mathcal{U}^k$ contains $\square$ exit (C subsumes D)
Else set $\mathcal{U}^{k+1} = \{$resolvents of C_1 and $C_2 : C_1 \in \mathcal{U}^k$ and $C_2 \in W\}$

④ If $\mathcal{U}^{k+1} = \phi$ exit (C does not subsume D)
Else $k = k+1$; goto ③.

Example Let $C = \sim P(x) \vee Q(f(x), a)$
$D = \sim P(h(y)) \vee Q(f(h(y), a) \vee \sim P(z)$

Then $\theta = \{b/y, c/z\}$
$W = \{\sim P(h(b)), \sim Q(f(h(b)), a), P(c)\}$

$\mathcal{U}^0 = \{C\}$
$\mathcal{U}^1 = \{Q(f(h(b)), a), \sim P(h(b)), Q(f(c), a)\}$
$\mathcal{U}^2 = \{\square\}$ ∴ C subsumes D.

Exercises ~~Section 4.2~~

(4.2) ① Prove $\{P \vee Q \vee R, \sim P \vee R, \sim Q, \sim R\}$ is unsatisfiable by resolution
② Prove $\{P \vee Q, \sim Q \vee R, \sim P \vee Q, \sim R\}$ " "

(4.3) ③ Let $\theta = \{a/x, b/y, g(x,y)/z\}$ and $E = P(h(x), z)$. Find $E\theta$.
④ Let $\theta_1 = \{a/x, f(z)/y, y/z\}$ and $\theta_2 = \{b/x, z/y, g(x)/z\}$. Find $\theta_1 \circ \theta_2$.

(4.4) ⑤ Determine whether each of the following sets is unifiable and find an mgu if so
$\{Q(a), Q(b)\}$ $\{Q(a,x), Q(a,a)\}$ $\{Q(a,x,f(x)), Q(a,y,y)\}$

(4.5) ⑥ Give the factors, if any, of $P(x) \vee Q(y) \vee P(f(x))$; $P(x) \vee P(a) \vee Q(f(x))$;
$P(x,y) \vee P(a, f(a))$; $P(a) \vee P(b) \vee P(x)$; $P(x) \vee P(f(y)) \vee Q(x,y)$

⑦ Find all possible resolvents, if any, of $\{\sim P(x) \vee Q(x,b), P(a) \vee Q(a,b)\}$;

(4.7) ⑧ Prove by resolution that $(P \to Q) \to (\sim Q \to \sim P)$

(4.8) ⑨ Let $C = P(x,y) \vee Q(z)$ and $D = Q(a) \vee P(b,b) \vee R(u)$. Does C subsume D?

⑩ Let $C = P(x,y) \vee R(y,x)$ and $D = P(a,y) \vee R(z,b)$. Prove that C does not subsume D.

⑪ Let $C = \sim P(x) \vee P(f(x))$ and $D = \sim P(x) \vee P(f(f(x)))$. Show that $C \to D$ but C does not subsume D.

5 SEMANTIC & LOCK RESOLUTION. (Lock methods complete)

5.1 Although the deletion strategy can be used to delete irrelevant and redundant clauses, time has already been wasted generating them. In order to have efficient theorem proving procedures, we must prevent large numbers of useless clauses from being generated.

5.2 INTRODUCTION TO SEMANTIC RESOLUTION.

The basic strategy is to divide S into 2 subsets, not allowing resolution between clauses in the same set. We use an interpretation I, dividing S into sets

$S_1 = \{$ clauses falsified by I $\}$

$S_2 = \{$ clauses satisfied by I $\}$

We will always get 2 non-empty subsets if S is unsatisfiable, as if $S_1 = \Phi$, S is satisfiable by I, or if $S_2 = \Phi$, S is satisfiable by $\sim I$.

We further restrict the number of resolutions by ordering predicates (eg $P > Q > R$) and requiring that the resolved literal in the clause from S_1 is the largest in the clause.

One other method we use to restrict our search is the clash

5.3 Formal Defns and Examples of S.R.

D: Let I be an interpretation. Let P be an ordering of predicate symbols. A finite set of clauses $\{E_1, \ldots, E_q, N\}$ where $q \geq 1$, is called a semantic clash w.r.t. P and I (or PI-clash) iff $E_1, \ldots, E_q$ (the electrons) and N (the nucleus) satisfy:

① $E_1, \ldots, E_q$ are false in I

② Let $R_1 = N$. For each $i = 1, \ldots, q$, there is a resolvent R_{i+1} of R_i and E_i

③ The literal in E_i which is resolved upon contains the largest predicate symbol in E_i, $i = 1, \ldots, q$

④ R_{q+1} is false in I.

R_{q+1} is called a PI-resolvent (of the PI-clash $\{E_1, \ldots, E_q, N\}$

In a PI-clash $\{E_1, E_2, \ldots, E_n, N\}$ E_1 is considered as the first electron, E_2 the second, and so on. Regardless of the order, we always get the same PI-resolvent of this clash. Therefore, we can avoid generating too many deductions of the same resolvent. This is one of the reasons for using clashes.

D: Let I be an interpretation for a set S of clauses, and P be an ordering of predicate symbols appearing in S. A deduction from S is called a PI-deduction iff each clause in the deduction is either a clause in S, or a PI-resolvent.

Examples

① Let $E_1 = A_1 \vee A_3$, $E_2 = A_2 \vee A_3$; $N = \sim A_1 \vee \sim A_2 \vee A_3$

Let $I = \{\sim A_1, \sim A_2, \sim A_3\}$ and P be an ordering in which $A_1 > A_2 > A_3$. Then $\{E_1, E_2, N\}$ is a PI-clash, the PI-resolvent of which is A_3 (note A_3 is false in I). Neither $\{E_1, N\}$ nor

{E2,N} is a PI-clash; the resolvent of {E1,N} is ~A2 v A3 which is true in I, for eg.

5.4 Completeness of SR

Lemma: If P is an ordering of predicate symbols in a finite unsatisfiable set S of ground clauses, and if I is an interpretation of S, then there is a PI-deduction of □ from S.

Theorem (completeness of PI-R): If P is an ordering of predicate symbols in a finite and unsatisfiable set S of clauses, and if I is an interpretation of S, then there is a PI-deduction of □ from S.

5.5 Special cases of SR: Hyperresolution & the Set-of-Support strategy.

Hyperresolution

Consider an interpretation I in which every literal is the negation of an atom. Then, every electron and every PI-resolvent must contain only atoms. Similarly, if every literal in I is an atom, then every electron & PI-resolvent must contain only negations of atoms.

D: A clause is positive if it contains no negation signs. A clause is negative if every literal of it contains the negation sign. A clause is mixed if it is neither positive nor negative.

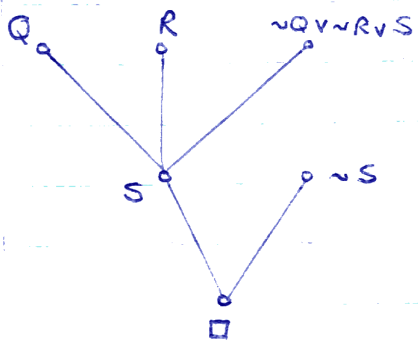
D: A positive hyperresolution is a special case of PI-R in which every literal in the interpretation I contains the negation sign (ie all electrons & PI-R's are positive)

D: A negative hyperresolution is a special case of PI-R in which the interpretation I does not contain any negation signs.

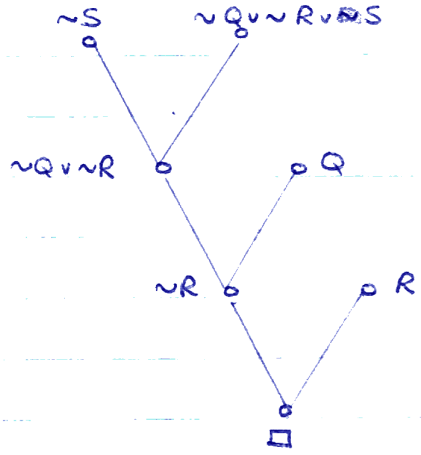
Both positive and negative hyperresolution are complete. It is often the case that the axioms of a theorem are represented by some positive and mixed clauses, and the negation of the conclusion by a negative clause. In this case, positive HR roughly corresponds to "thinking forward", while negative HR corresponds to "thinking backward".

Example: Axioms Q, R, $\neg Q \vee \neg R \vee S$. Conclusion S.
This gives the clauses $\{Q, R, \neg Q \vee \neg R \vee S, \neg S\}$
Let the ordering be $Q > R > S$. Then we have

POSITIVE HYPERDEDUCTION OF □
(start with axioms and derive a contradiction with neg. of conc)



NEGATIVE HYPERDEDUCTION OF □
(Start with neg of conc & try to derive a contradiction)



The Set-of-Support Strategy

A theorem consists of a set of axioms $A_1, A_2, \ldots, A_n$ and a conclusion B. To prove the theorem, we are essentially proving that $A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge \sim B$ is unsatisfiable. Since $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ is usually satisfiable, it might be wise to avoid resolving clauses in $A_1 \wedge A_2 \wedge \ldots \wedge A_n$.

D: A subset T of a set S of clauses is called a <u>set-of-support</u> of S if $S \setminus T$ is satisfiable. A <u>set-of-support resolution</u> is a resolution of two clauses that are not both from $S \setminus T$. A <u>set-of-support deduction</u> is a deduction in which every resolution is a s-o-s resolution.

A logical choice for T is the set of clauses originating from the negation of the theorem to be proved. The set $S \setminus T$ should then be the basic axioms of some theory which are assumed to form a satisfiable set. The set-of-support strategy avoids seeking a proof within that subset which is assumed to be itself satisfiable.

<u>Theorem (Completeness)</u>: If S is a finite unsatisfiable set of clauses and T is a subset of S such that $S \setminus T$ is satisfiable, then there is a set-of-support deduction of $\square$ from S with T as a set-of-support.

<u>Example</u>: Let S be a set containing the clauses

① $P(g(x_1, y_1), x_1, y_1)$

② $\sim P(x_2, h(x_2, y_2), y_2)$

③ $\sim P(x_3, y_3, u_3) \vee P(y_3, z_3, v_3) \vee \sim P(x_3, v_3, w_3) \vee P(u_3, z_3, w_3)$

④ $\sim P(k(x_4), x_4, k(x_4))$

Let T be a set consisting of clause ④. Then the following deduction is a set-of-support deduction with T as the set of support. Note that no resolution is performed among ①, ② and ③.

⑤ $\sim P(x_3, y_3, k(z_3)) \vee P(y_3, z_3, v_3) \vee \sim P(x_3, v_3, k(z_3))$ (④ & ③)

⑥ $\sim P(x_3, y_3, k(h(y_3, v_3))) \vee \sim P(x_3, v_3, k(h(y_3, v_3)))$ (⑤ & ②)

⑦ $\square$ (⑥ & ①).

5.6 Semantic Resolution using Ordered Clauses

We now consider orderings other than orderings of predicate symbols. In the FOL where variables are involved, in an clause E there may be more than one literal that contains the same largest predicate symbol in E, and any one of those is a candidate to be resolved upon.

Eg. Consider $E = Q(a) \vee Q(c) \vee Q(d)$

$N = \sim Q(x)$

Let I be an interpretation in which every literal is negative, and P be any ordering of predicate symbols. Then $\{E, N\}$ is a PI-clash. Since E contains four literals with the same predicate symbol, every one of them can be resolved with N.

Thus using orderings of predicate symbols, we may not be able to single out uniquely a literal in an clause. To remedy this, we consider ordered clauses. We consider a clause as a sequence of literals rather than a set. The rightmost literal is considered the largest.

D: An ordered clause is a sequence of distinct literals.

D: A literal L_2 is greater than a literal L_1 in an ordered clause (or L_1 is smaller than L_2) iff L_2 follows L_1 in the sequence specified by the ordered clause.

D: If two or more (unframed) literals (with the same sign) of an ordered clause C have a mgu σ, then the ordered clause obtained from the sequence $C\sigma$ by deleting any (unframed) literal that is identical to a smaller literal in the sequence is an ordered factor of C.

D: Let C_1 and C_2 be ordered clauses with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2 respectively. If L_1 and $\sim L_2$ have a mgu σ, and if C is the ordered clause obtained by concatenating the sequences $C_1\sigma$ and $C_2\sigma$, removing $L_1\sigma$ and $L_2\sigma$, and deleting any literal that is identical to a smaller literal in the remaining sequence, then C is called an ordered binary resolvent of C_1 against C_2. The literals L_1 and L_2 are the literals resolved upon.

D: An ordered resolvent of an ordered clause C_1 against an ordered clause C_2 is one of the following ordered binary resolvents:

an ordered binary of (an ordered factor of) C_1 against (an ordered factor of) C_2.

D: Let I be an interpretation. A finite sequence of ordered clauses $(E_1, E_2, \ldots, E_q, N)$ $q \geq 1$, is an ordered semantic clash with respect to I (or OI-clash) iff $E_1, \ldots E_q$ (the ordered electrons) and N (the ordered nucleus) satisfy the following:

- $E_1, E_2, \ldots, E_q$ are false in I
- Let $R_q = N$. For each $i = q, q-1, \ldots, 1$ there is an ordered resolvent R_{i-1} of E_i against R_i
- The literal in E_i resolved upon is the "last" literal in E_i, $i = 1, \ldots, q$; the literal in R_i that is resolved upon is the "largest literal" that has an instance true in I.
- R_0 is false in I.

R_0 is an <u>OI-resolvent</u> of the OI-clash $(E_1, E_2, \ldots, E_q, N)$

D: Let I be an interpretation for a set S of ordered clauses. A deduction from S is an <u>OI-deduction</u> iff each ordered clause in the deduction is either an ordered clause in S or an OI-resolvent

<u>Example</u> Consider the following ordered clauses

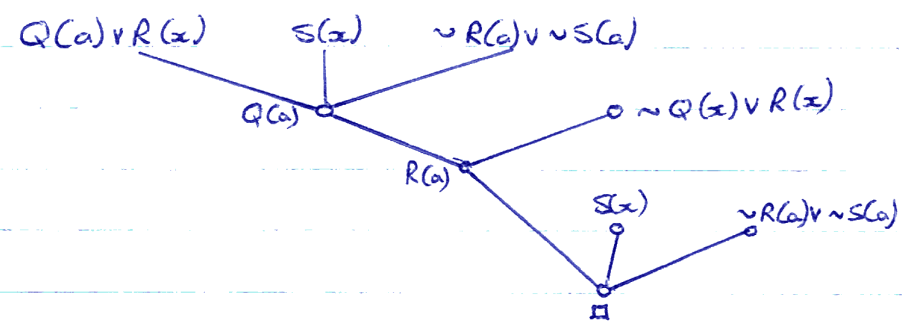
① $Q(a) \vee R(x)$

② $\sim Q(x) \vee R(x)$

③ $\sim R(x) \vee \sim S(a)$

④ $S(x)$

Let I be an interpretation in which every literal is negative. Then an OI-deduction of $\square$ from S is



Example. Consider the following set S of ordered clauses

① $P \vee Q$ ④ $\sim R \vee \sim P$

② $Q \vee R$ ⑤ $\sim W \vee \sim Q$

③ $R \vee W$ ⑥ $\sim Q \vee \sim R$

Let I be an interpretation in which every literal is negative. Thus, clauses ① - ③ can be used as ordered electrons, and clauses ④ - ⑥ can be used as an ordered nucleus. We obtain the following OI-resolvents

⑦ $R \vee P$ from OI-clash (⑤, ①, ⑤)

⑧ $P \vee Q$ from OI-clash (①, ②, ⑥)

From clauses ① - ⑧, we obtain the following OI-resolvent:

⑨ $Q \vee R$ from OI-clash (②, ⑦, ④)

Note that clauses ⑧ and ⑨ are in S. Therefore, from clauses ① - ⑨ we cannot produce any new OI-resolvents, i.e. $\square$ cannot be produced by OI-resolution. However, S is unsatisfiable. Hence OI-resolution is <u>not complete.</u>

57 IMPLEMENTATION OF SEMANTIC RESOLUTION.

A <u>positive ordered clause</u> is an ordered clause that does not contain any negation sign, and a <u>negative ordered clause</u> is an ordered clause in which every literal contains the negation sign. A <u>nonpositive (nonnegative) ordered clause</u> is an ordered clause that is not positive (negative). We consider positive hyperresolution (negative can be treated similarly). Thus we need consider only positive ordered clauses as candidates for electrons, and non-positive ones for the nucleus. We agree that for any nonpositive ordered clauses, negative literals are put after positive literals. Let S be a set of ordered clauses; P an ordering of predicate symbols in S.

GENERATION OF POSITIVE HYPERRESOLUENTS

Step ⓪ : Let M and N be the sets of all positive and nonpositive ordered clauses in S, respectively.

Step 1 : $j := 1$

Step 2 : $A_0 = \Phi$; $B_0 := N$

Step 3 : $i := 0$

Step 4 : If A_i contains $\square$ terminate (contradiction found)

Step 5 : If $B_i = \Phi$ then goto ⑧

Step 6 : $W_{i+1} := \{$ ordered resolvents of C_1 against C_2, where C_1 is an ordered clause or an ordered factor of an ordered clause in M, C_2 is an ordered clause in B_i, the resolved literal of C_1 contains the "largest" predicate symbol in C_1, and the resolved literal of C_2 is the "last" literal of C_2 $\}$

$A_{i+1} := \{$ all positive clauses in $W_{i+1}\}$

$B_{i+1} := \{$ all nonpositive clauses in $W_{i+1}\}$

Step 7 : $i = i + 1$; goto ④

Step 8 : $T := A_0 \cup \ldots \cup A_i$

$M := M \cup T$

Step 9 : $j := j + 1$

Step 10 : $R := \{$ ordered resolvents of C_1 against C_2, where C_1 is an ordered (factor of an ordered) clause in T, C_2 is an ordered clause in N, and the resolved literal of C_1 contains the "largest" predicate symbol in $C_1\}$

$A_0 := \{$ all positive ordered clauses in $R\}$

$B_0 = \{$ all nonpositive ordered clauses in $R\}$

goto ③

(example: Chang & Lee p 118)

For each j, B_i will eventually be empty since the maximum number of negative literals in any clause of B_i decreases by one as i increases by one. All the clauses in each A_i are positive hyperresolvents. The deletion strategy can be incorporated for T and M.

5.8 LOCK RESOLUTION

D: Let C be a clause that has every one of its literals indexed with integers. If two or more literals (with the same sign) of C have a mgu σ, then the clause obtained from $C\sigma$ by deleting any literal that is identical to a literal of lower index* is a lock factor of C.

D: Let C_1 and C_2 be two clauses with no variables in common and with every literal in them indexed. Let L_1 and L_2 be two literals of lowest index in C_1 and C_2 respectively. If L_1 and $\neg L_2$ have a mgu σ, and if C is the clause obtained from $(C_1\sigma \vee C_2\sigma)$ by removing $L_1\sigma$ and $L_2\sigma$ and by merging low for any identical literals in the remaining clause, then C is called a binary lock resolvent of C_1 and C_2. L_1 and L_2 are the literals resolved upon.

D: Let C_1 and C_2 be two clauses with every literal in them indexed. A lock resolvent of C_1 and C_2 is one of the following: binary lock resolvents: a binary lock resolvent of (a lock factor of) C_1 and (a lock factor of) C_2.

D: Let S be a set of clauses, where every literal in S is indexed with an integer. A deduction from S is a lock deduction iff every clause in the deduction is either a clause in S or a lock resolvent.

* called 'merging low'

Example: Consider the following set S of clauses:

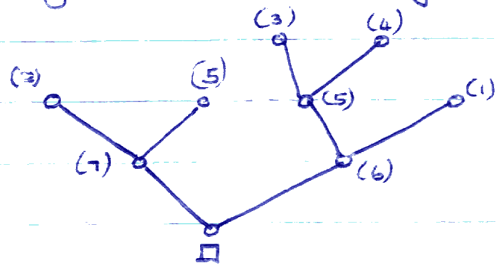
① $P \vee {}_2Q$
② ${}_3\sim P \vee {}_4\sim Q$
③ ${}_6\sim P \vee {}_5Q$
④ ${}_8\sim P \vee {}_7\sim Q$

} S

We get the Lock resolvents:

⑤ ${}_6\sim P$ from ③ & ④
⑥ ${}_2Q$ " ① & ⑤
⑦ ${}_4\sim Q$ " ② & ⑤
⑧ $\square$ " ⑥ & ⑦

Only three (non-empty) Lock resolvents were generated. If we had used Level saturation, 37 resolvents would have been necessary before deducing $\square$



(Note: indexing is arbitrary)

Example: Consider

① ${}_5P(y,a) \vee {}_1P(f(y),y)$
② ${}_6P(y,a) \vee {}_2P(y,f(y))$
③ ${}_8\sim P(x,y) \vee {}_3P(f(y),y)$
④ ${}_9\sim P(x,y) \vee {}_4P(y,f(y))$
⑤ ${}_{10}\sim P(x,y) \vee {}_7\sim P(y,a)$

} S

From S, we can obtain the following Lock deduction of $\square$

⑥ ${}_5P(a,a) \vee {}_{10}\sim P(x,f(a))$ Lock resolvent of ① & ⑤
⑦ ${}_8\sim P(x,a) \vee {}_{10}\sim P(y,f(a))$ " " ③ & ⑤
⑧ ${}_{10}\sim P(x,f(a)) \vee {}_{10}\sim P(y,f(a))$ " " ⑥ & ⑦
⑨ ${}_6P(a,a)$ " " ⑧ & ②
⑩ ${}_9\sim P(x,a)$ " " ⑧ & ④
⑪ $\square$ " " ⑨ & ⑩

5.9 Completeness of Lock Resolution

Lemma: Let C_1 and C_2 be two clauses with every literal of them indexed. If C_1' and C_2' are instances of C_1 and C_2 respectively, and if C' is a lock resolvent of C_1' and C_2'; then there is a lock resolvent C of C_1 and C_2 such that C' is an instance of C.

Lemma: Let S be a set of ground clauses, where every literal in S is indexed with an integer. If S is unsatisfiable, then there is a lock deduction of $\square$ from S.

Theorem (Completeness) Let S be a set of clauses, where every literal in S is indexed with an integer. If S is unsatisfiable, then there is a lock deduction of $\square$ from S

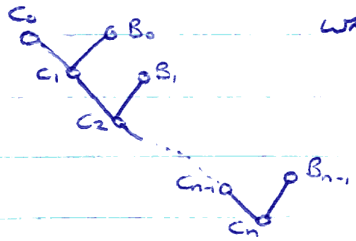
Although lock resolution itself is complete, it is not compatible with most other resolution strategies. For example, the combination of lock resolution and the deletion strategy is not complete (similarly with set-of-support strategy). It is thus very restrictive, but highly efficient.

(Exercises Chang & Lee p126 - 129)

6. Linear Resolution

6.1 When proving an identity, we often start with the left hand side of the identity, apply an inference rule to it to obtain some other expression, then apply some inference rule again to the expression we have just obtained until the left-side expression is identical to the right-side expression. The idea of linear resolution is similar: start with a clause, resolve it against a clause to obtain a resolvent, and resolve this resolvent against some clause until $\square$ is obtained.
Linear resolution is simple, complete, and compatible with the set of support strategy.

6.2 Linear Resolution

D: Given a set S of clauses and a clause C_0 in S, a linear deduction of C_n from S with top clause C_0 is a deduction of the form  where:

① for $i = 0, 1, \ldots, n-1$, C_{i+1} is a resolvent of C_i (the centre clause) and B_i (side clause)

② each B_i is either in S, or is a C_j for some $j < i$.

This is a primitive form of linear deduction, and is later modified to CL-deduction (§6.4).

6.3 Input Resolution and Unit Resolution

These are two refinements of linear resolution which are much more efficient, but are not complete.
Given a set S of clauses, since S is the original input set, we shall call each member of S an input clause.

D An input resolution (vine form) is a resolution in which one of the two parent clauses is an input clause. An input deduction is a deduction in which every resolution is an input resolution. An input refutation is an input deduction of $\square$ from S.

An input deduction is actually a linear deduction in which every side clause is an input clause, and hence is a subcase of linear deduction.

D: A unit resolution is a resolution in which a resolvent is obtained by using at least one unit parent clause, or a unit factor of a parent clause. A unit deduction is a deduction in which every resolution is a unit resolution. A unit refutation is a unit deduction of $\square$.

Unit resolution is essentially an extension of the one-literal rule (cf § 3.6) to FOL. In order to deduce $\square$ from a given set of clauses, one must obtain successively shorter clauses, and unit resolution provides a means for progressing rapidly toward shorter clauses. Unit resolution and input resolution are equivalent.

6.4 Linear Resolution using Ordered Clauses and the Information of Resolved Literals.

Unlike semantic resolution, we can introduce ordered clauses into linear resolution without destroying completeness. Furthermore, the information provided by literals resolved upon (which is usually lost when the literals are deleted) can be used to improve linear resolution. Literals resolved upon are retained as framed literals in the ordered clause; these literals are purely for recording information, and do not participate in the resolution.

D: An ordered clause C is a reducible ordered clause iff the last literal of C is unifiable with the negation of a framed literal of C.

Framed literals not followed by unframed literals may also be deleted. Thus $\boxed{P} \vee \boxed{Q} \vee \sim P$ is reducible, reducing to $\boxed{P} \vee \boxed{Q}$ and then $\square$. This kind of operation is called the reduction of a reducible ordered clause.

D: Let C be a reducible ordered clause. Let the last literal L be unifiable with some framed literal with a mgu σ. The reduced ordered clause of C is the ordered clause obtained from $C\sigma$ by deleting $L\sigma$ and every subsequent framed literal not followed by any unframed literal.

In an ordered clause, if there is more than one occurrence of the same unframed literal, we always keep only the left most one and delete the other identical unframed literals. This operation is called merging left for identical unframed literals

D: If two or more unframed literals (with the same sign) of an ordered clause C have a mgu σ, the ordered clause obtained from the sequence Cσ by merging left for any identical literals in Cσ and by deleting every framed literal not followed by an unframed literal in the remaining clause is called an ordered factor of C.

D: Let C_1 and C_2 be two ordered clauses with no variables in common and L_1 and L_2 be two unframed literals in C_1 and C_2 respectively. Let L_1 and $\neg L_2$ have a mgu σ. Let C^* be the ordered clause obtained by concatenating the sequence $C_1\sigma$ and $C_2\sigma$, framing $L_1\sigma$, deleting $L_2\sigma$, and merging left for any unframed literals in the remaining sequence. Let C be obtained from C^* by removing every framed literal not followed by any unframed literal in C^*. C is called an ordered binary resolvent of C_1 against C_2. The literals L_1 and L_2 are the literals resolved upon.

D: An ordered resolvent of an ordered clause C_1 against an ordered clause C_2 is any of the following:
an ordered binary resolvent of (an ordered factor of) C_1 against (an ordered factor of) C_2.

D: Given a set S of ordered clauses and an ordered clause C_0 in S, an OL-deduction of C_n from S with top ordered clause C_0 is a deduction of the form (as before) satisfying:

- for $i = 0, 1, \ldots, n-1$, C_{i+1} is an ordered resolvent of C_i (a centre ordered clause) against B_i (a side ordered clause); the literal resolved upon in C_i (or an ordered factor of C_i) is the last literal.

- each B_i is either an ordered clause in S or an instance of some C_j, $j<i$. B_i is an instance of some C_j, $j<i$ iff C_i is a reducible ordered clause. In this case, C_{i+1} is the reduced ordered clause of C_i.
- no tautology is in the deduction.

D: An <u>OL-refutation</u> is an OL-deduction of $\square$.

6.5 Completeness of Linear Resolution.

<u>Theorem</u>: If C is an ordered clause in an unsatisfiable set S of ordered clauses, and if $S - \{C\}$ is satisfiable, then there is an OL-refutation from S with top ordered clause C.

7. The Equality Relation

D Given a set S of clauses, let W be the set of all interpretations of S. Let Q be a nonempty subset of W. Then S is Q-unsatisfiable iff S is false in every elt of Q.

D An E-interpretation I of a set S of clauses is an interpretation of S satisfying the following conditions. Let α, β and γ be any terms in the Herbrand universe of S and let L be a literal in I.

Then
- $(\alpha = \alpha) \in I$
- if $(\alpha = \beta) \in I$, then $(\beta = \alpha) \in I$
- if $(\alpha = \beta) \in I$ and $(\beta = \gamma) \in I$, then $(\alpha = \gamma) \in I$
- if $(\alpha = \beta) \in I$ and L' is the result of replacing some one occurrence of α in L by β, then $L' \in I$.

An E-interpretation is just an interpretation which satisfies the reflexive, symmetric, transitive and substitutive axioms of equality

D A set S of clauses is E-satisfiable iff there is an E-interpretation satisfying all clauses in S; otherwise S is E-unsatisfiable

Example: Consider $S \triangleq \{P(a), \sim P(b), a = b\}$. There are 64 interpretations of S, of which the following 6 are E-interpretations:

$\{P(a), P(b), a=a, b=b, a=b, b=a\}$
$\{\sim P(a), \sim P(b), a=a, b=b, a=b, b=a\}$
$\{P(a), P(b), a=a, b=b, a \neq b, b \neq a\}$
$\{P(a), \sim P(b), a=a, b=b, a \neq b, b \neq a\}$
$\{\sim P(a), P(b), a=a, b=b, a \neq b, b \neq a\}$
$\{\sim P(a), \sim P(b), a=a, b=b, a \neq b, b \neq a\}$

Clearly S is false in each of these, and hence E-unsatisfiable (but not unsatisfiable)

D: Let S be a set of clauses. The set F of the functionally reflexive atoms for S is the set

$$F \triangleq \{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)\}$$

for all n-place function symbols f occurring in S.

D: Let S be a set of clauses. Then the set of equality axioms for S is the set consisting of the following clauses

① $x = x$

② $x \neq y \vee y = x$

③ $x \neq y \vee y \neq z \vee x = z$

④ $x_j \neq x_0 \vee \sim P(x_1, \ldots, x_j, \ldots, x_n) \vee P(x_1, \ldots, x_0, \ldots, x_n)$ for $j = 1, \ldots, n$, for every n-place predicate symbol P in S.

⑤ $x_j \neq x_0 \vee f(x_1, \ldots, x_j, \ldots, x_n) = f(x_1, \ldots, x_0, \ldots, x_n)$ for $j = 1, \ldots, n$ for every n-place function symbol f in S.

Theorem: Let S be a set of clauses and K be the set of equality axioms for S. Then S is E-unsatisfiable iff $(S \cup K)$ is unsatisfiable

Proof: ($\Rightarrow$) Suppose S is E-unsatisfiable, but $(S \cup K)$ is satisfiable. Then there is an interpretation I satisfying $(S \cup K)$. Since I satisfies K, I is an E-interpretation. Contradiction.

($\Leftarrow$) Suppose $(S \cup K)$ is unsatisfiable, but S is E-satisfiable. Then there is an E-interpretation I_E such that I_E satisfies S. Clearly, I_E satisfies K, and hence $(S \cup K)$. Contradiction

Theorem: A finite set S of clauses is E-unsatisfiable iff there is a finite set S' of ground instances of clauses in S such that S' is E-unsatisfiable

Proof ($\Rightarrow$) S unsatisfiable $\Rightarrow$ $(S \cup K)$ unsatisfiable (by last theorem). Herbrand's theorem says that there is a finite set of ground clauses in S such that $(S' \cup K)$ is unsatisfiable. Hence S' is E-unsatisfiable.

($\Leftarrow$) Since S' is E-unsatisfiable, every E-interpretation falsifies S' and thus S. Therefore S is E-unsat.

7.3 PARAMODULATION

Using both resolution and paramodulation, we can always deduce $\square$ from an E-unsatisfiable set of clauses. P is essentially an extension of the equality substitution rule and can be applied to any pair of clauses (not necessarily unit clauses)

Notation: $E[t]$ = an expression E (eg clause, literal, term) containing a term t. If we substitute s for one single occurrence of t the result is denoted $E[s]$

Paramod - for ground clauses can be stated as follows: If C_1 is $L[t] \vee C_1'$, where $L[t]$ is a literal containing term t, and C_1' is a clause, and if C_2 is $t = s \vee C_2'$ where C_2' is a clause, then we can infer the paramodulant (clause)

$$L[s] \cup C_1' \cup C_2'$$

Example: Consider $C_1 : P(a) \vee Q(b)$

$C_2 : a = b \vee R(b)$

Let L be $P(a)$, C_1' be $Q(b)$ and C_2 be $R(b)$. The paramodulant is $P(b) \vee Q(b) \vee R(b)$

D Let C_1 and C_2 be two clauses (parent clauses) with no variables in common. If C_1 is $L[t] \vee C_1'$ and C_2 is $r = s \vee C_2'$, where $L[t]$ is a literal containing term t and C_1' and C_2' are clauses, and if t and r have a most general unifier σ, then infer

$$L\sigma[s\sigma] \cup C_1'\sigma \cup C_2'\sigma$$

where $L\sigma[s\sigma]$ denotes the result obtained by replacing one single occurrence of $t\sigma$ in $L\sigma$ by $s\sigma$. The above inferred clause is called a binary paramodulant of C_1 and C_2; the literals L and $r = s$ the literals paramodulated upon (else we apply paramodulation from C_2 into C_1)

D A paramodulant of clause C_1 and C_2 (parent clauses) is a binary paramodulant of (a factor of) C_1 and (a factor of) C_2.

Paramodulation is an inference rule for the equality relation. The combination of paramod. and resolution is complete for E-unsatisfiable sets of clauses.

Example C_1: $P(g(f(x))) \vee Q(x)$ } $L \triangleq P(g(f(x)))$; $C_1' \triangleq Q(x)$

C_2: $f(g(b)) = a \vee R(g(c))$ } $r \triangleq f(g(b))$, $s \triangleq a$; $C_2' \triangleq R(g(c))$

L contains $f(x)$ which can be unified with r. Let t be $f(x)$. A mgu of t and r is $\sigma = \{g(b)/x\}$. Therefore $L\sigma[t\sigma]$ is $P(g(f(g(b))))$. Hence $L\sigma[s\sigma]$ is $P(g(a))$. Since $C_1'\sigma$ is $Q(g(b))$ and $C_2'\sigma$ is $R(g(c))$, we obtain a binary paramodulant:

$$P(g(a)) \vee Q(g(b)) \vee R(g(c))$$

literals P'd upon: $P(g(f(x)))$, $f(g(b)) = a$

Ex Find all paramodulants of:

C_1: $P(f(x, g(x))) \vee Q(x)$

C_2: $a = b \vee g(a) = a \vee f(a, g(a)) = b$.

7.4 HYPERPARAMODULATION (for E-unsat. complete)

The refinements of paramodulation are somewhat weaker than those of resolution. There are no complete counterparts to PI-resolution and OL-resolution. However, hyperresolution and linear resolution can be extended to paramodulation

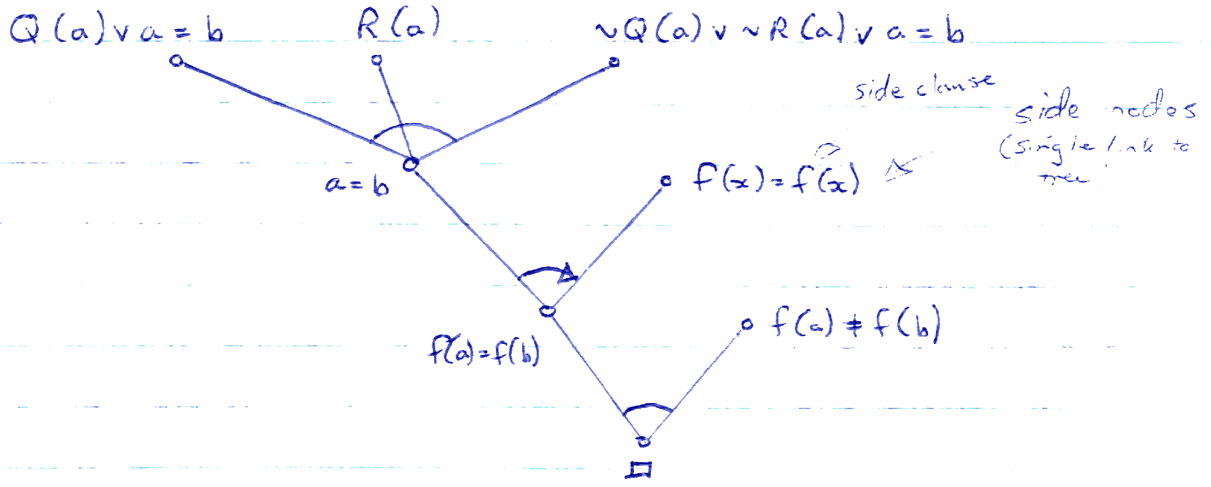
D: Let P be an ordering of the predicate symbols that includes the predicate symbols in clauses C_1 and C_2. Then a paramodulant of C_1 and C_2 is called a <u>P-hyperparamodulant</u> iff
- C_1 and C_2 are positive clauses
- The literals paramodulated upon in C_1 and C_2 contain the largest predicate symbols in C_1 and C_2 respectively.

(On the other hand, a P-hyperresolvent is a PI-resolvent, where every literal in I contains a negation sign $\neg$ in this case electrons and P-hyperresolvents are positive.)

D: Let P be an ordering of the predicate symbols in a set S of clauses. Then a <u>P-hyperdeduction with resolution & paramodulation</u> is a deduction in which every clause is a clause in S, or a P-hyperresolvent, or a P-hyperparamodulant. A <u>P-hyperrefutation with resolution & paramodulation</u> is a P-hyperdeduction of $\square$ with resolution & paramod.

<u>Example</u>: $S \triangleq \{\neg Q(a) \vee \neg R(a) \vee a \qquad \vee a = b, R(a), f(a) \neq f(b)\}^{f(x)=f(x)}$

Let P be $Q > R > =$. A P-hyperrefutation with r & p is shown opposite, where $\curvearrowright$ indicates that a P-hyperpara. is applied from left to right, and $\frown$ indicates that P-hyperres. is applied



7.5 INPUT & UNIT PARAMODULATIONS

Let S be the original input set of clauses. Every member of S is an input clause.

An input paramodulation is a paramod. in which one of the two parent clauses is an input clause.

A unit paramodulation is a paramod. in which a paramodulant is obtained by using at least a unit parent clause or a unit factor of a parent clause.

An input (unit) deduction by paramodulation is a deduction obtained by employing input (unit) paramod. only.

An input (unit) deduction by resolution and paramodulation is a deduction obtained by employing both input (unit) resolution and input (unit) paramodulation.

An input (unit) refutation is an input (unit) deduction of the empty clause.

Lemma: If a set S of ground clauses has an input refutation by R & P, then it has a unit refutation by R & P.

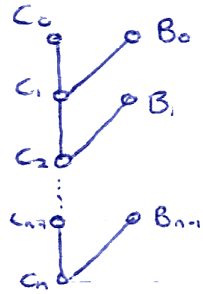
Theorem: If a set S of clauses has an input refutation by R & P, then S, together with its functionally reflexive axioms, has a unit refutation by R & P.

Lemma: If C is a clause in an E-unsatisfiable set S of ground unit clauses that contains all the ground instances of $x = x$, and if $S - \{C\}$ is E-satisfiable, then S has an input refutation by R. and P. with top clause C.

Theorem: If C is a clause in an E-unsatisfiable set S of unit clauses including $x = x$ and the functionally reflexive axioms, and if $S - \{C\}$ is E-satisfiable, then S has an input refutation by R. & P. with top clause C.

7.6 LINEAR PARAMODULATION (complete)

D: Given a set S of clauses and a clause C_0 in S, a linear deduction of C_n from S with top clause C_0 by R & P is a deduction of the form

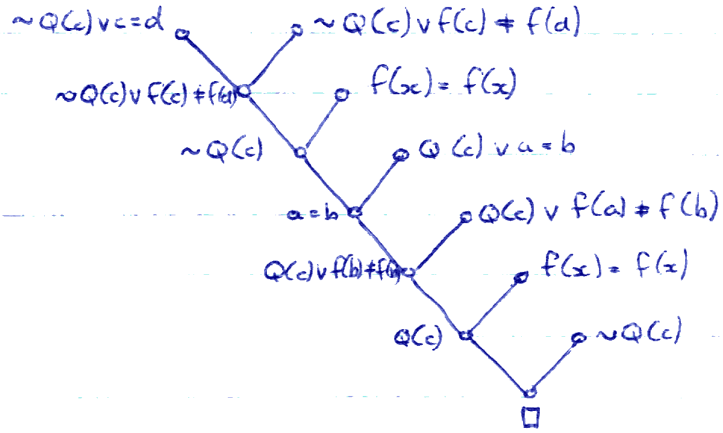


Where:

- for $i = 0, \ldots, n-1$, C_{i+1} is a resolvent or paramodulant of C_i (centre clause) and B_i (side-clause)
- each B_i is either in S, or is a C_j for some $j < i$.

Note that a linear deduction by R&P is an input deduction by R and P iff each B_i is in S. A linear refutation by R. and P. is a linear deduction of $\square$ by R. and P.

Example Let $S \triangleq \{\sim Q(c) \vee c = d,\ \sim Q(c) \vee f(c) \neq f(d),\ Q(c) \vee a = b,$
$Q(c) \vee f(a) \neq f(b),\ f(x) = f(x)\}$ Then a linear refutation from S with top clause $\sim Q(c) \vee c = d$ is shown below. There are six side clauses, five of which are from S.



Exercise $S \triangleq \{P(b) \vee Q(a),\ P(a) \vee \sim Q(b),\ \sim P(a) \vee Q(b),$
$\sim P(b) \vee \sim Q(a),\ a = b\}$
Find linear refutations by R & P from S using (i) clause 5 (ii) clause 1, as the top clause.

ADDENDA: BREADTH-FIRST LEVEL-SATURATION DEDUCTION WITH PARAMODULATION

$S_0 = S$; $i = 0$

repeat

$i = i + 1$

if odd (i), $S_i = S_{i-1} \cup \{$paramods of 2 clauses in $S_{i-1}\}$

if even (i), $S_i = S_i \cup \{$resolvents of 2 clauses in $S_{i-1}\}$

if $\square \in S_i$ exit

end repeat

8. Question Answering.

8.1 Introduction

Mechanical theorem proving techniques can be applied to design question-answering and problem-solving systems. The major task in designing such systems is optimal storage and execution efficiency.

We can divide questions into four classes according to the form of the answer.

Class A: yes/no answers
Class B: 'associative' (any term) answers (eg x is y, x if y)
Class C: action sequence answer (eg do x then y)
Class D: conditional answers (eg if x then y else z)

Generally speaking, in practical applications, a question answering system concerns class A & B questions, a problem-solving system concerns class C questions, and a program-synthesising system concerns class D questions.

To implement such systems, we need:
- a language for facts and questions
- search technique to get answer
- efficient technique
- ability to derive an answer not explicitly stored.

8.2 Class A Questions.

Since the answer to a class A question is simply "yes" or "no", the question answering problem is merely a theorem-proving problem where the given facts are considered as axioms of a theorem, and the question is presented as the conclusion of the theorem. Sometimes we cannot prove the theorem corresponding to the question. In this case, we attempt to disprove it. If we can do neither, we should answer "Insufficient information".

8.3 Class B Questions.

Steps involved

1. Obtain clause form for facts and question; use different variables in each clause.
2. Substitute new variables for any Skolem constants / functions in the negation of the theorem (question).
3. Append $\vee$ ans(x) to the clause arising from the negation of the theorem (question).
4. Perform resolution as usual
5. Answer is at the root, ie $\square$ (get $\square \vee$ ans(x))

Example given "John is Mary's husband". Asked "Who is Mary's husband?"
Let P(x,y) denote "x is the husband of y"
We are given P(John, Mary) ① We establish the following conclusion to answer our question: $(\exists x)\, P(x, \text{Mary})$ ②
If we can prove that ① ⇒ ②, at least we know an answer exists.
By Tracing the variable, we can also see what x is.
We negate ②, getting ③ : $\neg P(x, \text{Mary})$ Resolving ① & ③ we get a contradiction and the theorem is proved. In the process of resolving, x is replaced by "John". To trace x we add the ANS predicate

to clause ③, getting ④ $\sim P(x, Mary) \vee ANS(x)$.
Note that this is equivalent to $(\forall x)(P(x, Mary) \rightarrow ANS(x))$
Resolving ① and ④, we get $ANS(John)$.

Similar to a halting clause, a clause containing only the ANS predicate is an <u>answering clause</u>.

<u>Example</u>: Consider the facts:

F_1: For all x, y and z, if x is the father of y and z is the father of x, then z is the grandfather of y.

F_2: Everyone has a father.

and the question: For all x, who is the grandfather of x.

Let $P(x,y) \triangleq$ "x is the father of y"
$Q(x,y) \triangleq$ "x is the grandfather of y"

Then the above facts are:

① $\sim P(x,y) \vee \sim P(z,x) \vee Q(z,y)$

② $P(f(x), x)$ where $f(x) \triangleq$ the father of x.

Our question is:

③ $\sim Q(y,x) \vee ANS(y)$

We generate the resolvents:

④ $\sim P(z, f(y)) \vee Q(z,y)$ from ①, ②

⑤ $Q(f(f(y)), y)$ from ④, ②

⑥ $ANS(f(f(x)))$ from ⑤, ③

In maths, it is common practice to find a solution through proving the existence of such a solution.
See C & L p. 239 for further eg's.

8.4 Class C Questions.

For this kind of question, our task is to find a sequence of actions that will achieve some goal. The most important concept involved is the "state and state transformation" method. At a given moment, every object under consideration is said to be in a certain state. To achieve a goal, we have to change the present state of the object to a desired state - this requires action.

Example Consider a ——f_1—— b. Suppose the initial state of subject d is s_1 and d is initially at a. Let $P(x, y, z)$ denote " x is located at y in state z".
Then $P(d, a, s_1)$ ①.
Suppose every subject x in state z can be moved from location y_1 to location y_2 by action f_1. The range of $f_1(x, y_1, y_2, z)$ is the new state achieved after x, which is initially in state z, is moved from y_1 to y_2 by action f_1, i.e.
$(\forall x)(\forall y_2)(\forall z)\ (P(x, y_1, z)) \rightarrow P(x, y_2, f_1(x, y_1, y_2, z))$
We can now answer the question "How can we move object d from location a to location b?"
Our facts are
③ $P(d, a, s_1)$
④ $\sim P(x, y_1, z) \vee P(x, y_2, f_1(x, y_1, y_2, z))$
The question is
⑤ $\sim P(d, b, z) \vee ANS(z)$
We generate the resolvents:
⑥ $P(d, y_2, f_1(d, a, y_2, s_1))$ from ③ & ④
⑦ $ANS(f_1(d, a, b, s_1))$ from ⑤ & ⑥.
Thus our answer involves one action, namely applying f_1 to move d from a to b.

Example (11.8). A monkey wants to eat a banana suspended from the ceiling of a room. The monkey is too short to reach the banana. There is a chair in the room. The monkey can stand on the chair (and reach the banana) move the chair, and walk around.

Predicates:

$P(x, y, z, s) \equiv$ "In state s, the monkey is at x, the banana is at y, and the chair is at z"

$R(s) \equiv$ "In state s the monkey can reach the banana"

Functions (assumed self-explanatory)

walk (y, z, s) : startposition × endposition × current state → next state

carry (y, z, s) : as for walk, but with the monkey carrying the chair.

climb (s) : oldstate → newstate

We assume that initially, the monkey is at location a, the banana at location b, the chair at location c, and the monkey in state s_0.

Our axioms are:

① $\sim P(x, y, z, s) \vee P(z, y, z, \text{walk}(x, z, s))$

② $\sim P(x, y, x, s) \vee P(y, y, y, \text{carry}(x, y, s))$

③ $\sim P(b, b, b, s) \vee R(\text{climb}(s))$

④ $P(a, b, c, s_0)$

The "meanings" of these are:

① : in any state, the monkey can walk from location x to location z

② : if the monkey and chair are at x, they can move to y.

③ : if the chair and the monkey are both under the banana, the monkey can climb the chair and reach the banana.

④ initial situation

The question is

⑤ $\sim R(s) \vee ANS(s)$

The resolvents obtained are

⑥ $\sim P(b,b,b,s) \vee ANS(climb(s))$ ⑤ & ③

⑦ $\sim P(x,b,x,s) \vee ANS(climb(carry(x,b,s)))$ ⑥ & ②

⑧ $\sim P(z,b,z,s) \vee ANS(climb(carry(z,b,walk(x,z,s))))$ ⑦ & ①

⑨ $ANS(climb(carry(c,b,walk(a,c,s.))))$ ⑧ & ④

(i.e. walk from a to c, then carry chair from c to b, climb chair (and reach the banana.))

In general, it is best to start with the ANS clause (cf set-of-support strategy).

8.5 CLASS D QUESTIONS.

Assume we have the following facts

F_1: If John is under five years old, he should take drug a.

F_2: If John is not under five years old, he should take drug b.

Our question is "What drug should John take?"

Let $P(x) \triangleq$ "x is under five years old"

$R(x,y) \triangleq$ "x should take y"

Then we have

① $\sim P(John) \vee R(John, a)$

② $P(John) \vee R(John, b)$

③ $\sim R(John, x) \vee ANS(x)$

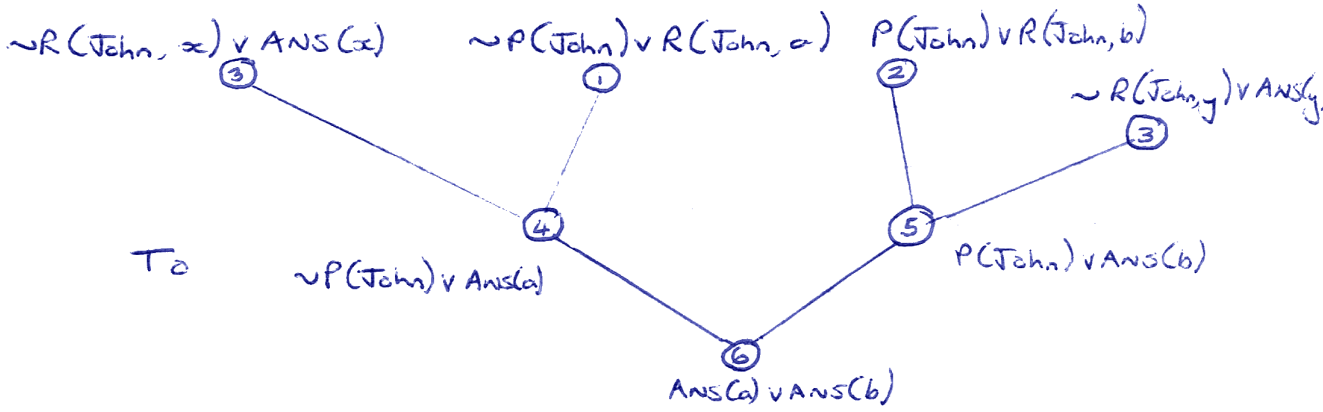
We generate the resolvents:

④ $\sim P(John) \vee ANS(a)$ ① & ③

⑤ $P(John) \vee ANS(b)$ ② & ③

⑥ $ANS(a) \vee ANS(b)$ ④ & ⑤

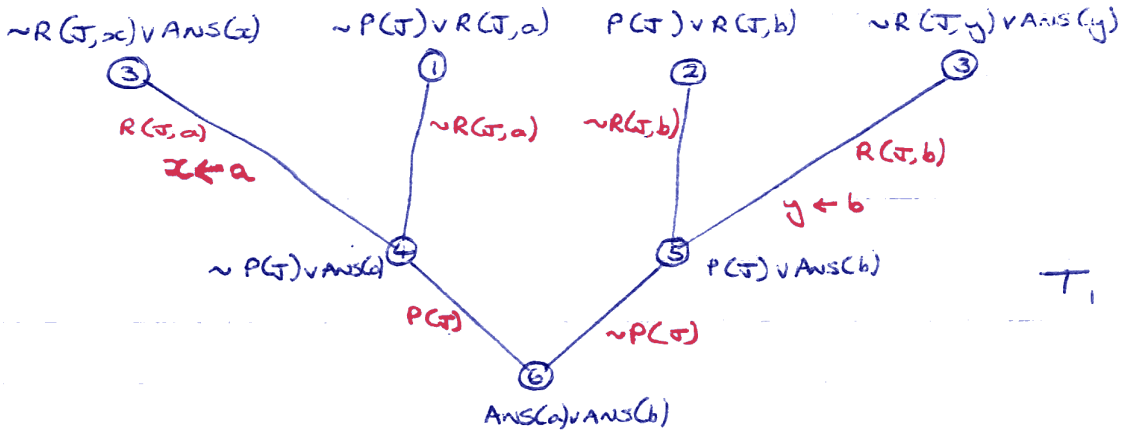
Clause (6) tells us that John should take drug a or drug b. This is far from satisfying - we want to know under what condition John should take drug a and under what condition he should take drug b, i.e. under what condition will ANS(a) (ANS(b)) be true?
The deduction of clause (6) is:



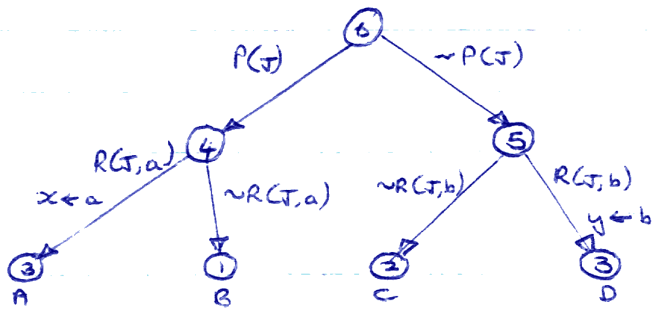
Call this deduction tree T_0. We now analyse T_0 according to the following algorithm.

An Information-Extraction Algorithm

(1) In T_0, if a clause C is a resolvent of clauses C_1 and C_2 with resolved literals L_1 and L_2 in C_1 and C_2 respectively, and that θ is a most general unifier of L_1 and $\sim L_2$, then on the arc from C_i to C, $i = 1, 2$, write down the negation of $L_i\theta$ and the substitution θ. Our tree becomes T_1.

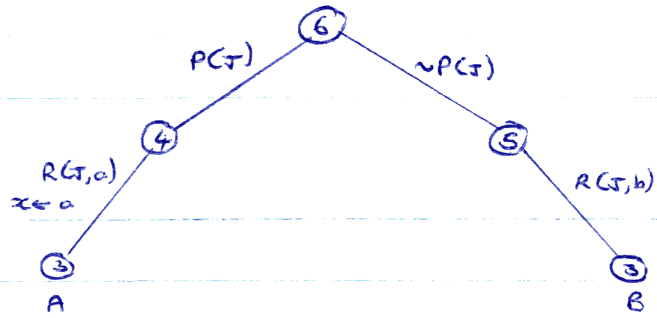


② Invert tree T_1, add arrows to the arcs, and delete all the clauses attached to the nodes, obtaining tree T_2. Label each tip node.



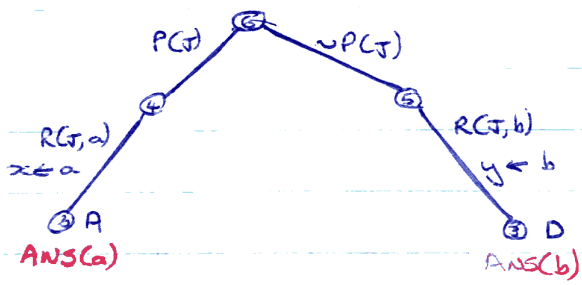
T_2

③ Delete every node (and its associated arcs) that correspond to a clause which does not contain an ANS predicate. In our example, we delete nodes ① and ②:



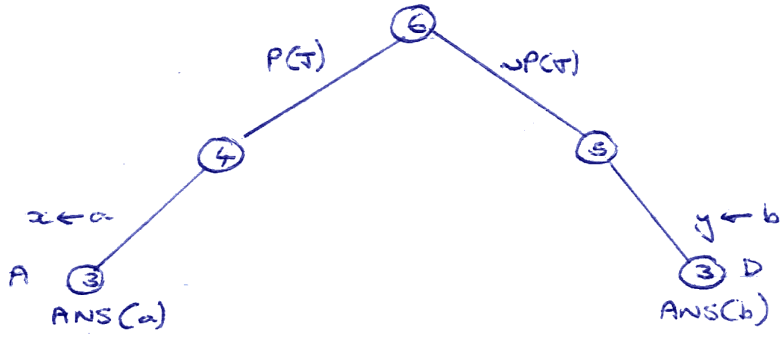
T_3

④ Let $N_1, N_2, \ldots, N_m$ be the tip nodes of T_3. For each N_i, let $I(N_i)$ denote the conjunction of literals attached to the path from the top node to N_i. Let $C(N_i)$ be the clause that corresponds to N_i. Find a literal $L(N_i)$ in the answering clause such that $L(N_i)$ is a logical consequence of $I(N_i) \wedge C(N_i)$ (such a literal always exists). Attach $L(N_i)$ to node N_i.



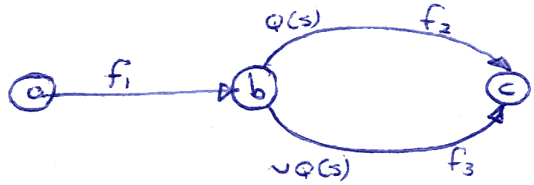
T_4

⑤ In T_4, let $N_1, N_2, \ldots, N_q$ be nodes where there is only one arc a_i leading out of N_i, $1 \le i \le q$. Let $L(a_i)$ be the literal attached to a_i. Delete $L(a_i)$, $1 \le i \le q$ from T_4 to get T_5, the decision tree.



(Validation C & L p 247 ff)

Example:



Assume that d wants to go to location c from location a. She can go to c from b either through f_2 or f_3, depending on whether some condition Q is satisfied. Assuming that d starts at a, we want to know how she can get to c.

Let $P(x, y, s)$ denote "x is at y in state s". Then the facts are:

① $\sim P(x, a, s) \vee P(x, b, f_1(x, a, b, s))$

② $\sim P(x, b, s) \vee \sim Q(s) \vee P(x, c, f_2(x, b, c, s))$

③ $\sim P(x, b, s) \vee Q(s) \vee P(x, c, f_3(x, b, c, s))$

④ $P(d, a, S_0)$

The question:

⑤ $\sim P(d, c, s) \vee ANS(s)$

The resolvents:

⑥ $\sim P(d, b, u) \vee Q(u) \vee ANS(f_3(d, b, c, u))$ ⑤ & ③

⑦ $\sim P(d, b, v) \vee Q(v) \vee ANS(f_2(d, b, c, v))$ ⑤ & ②

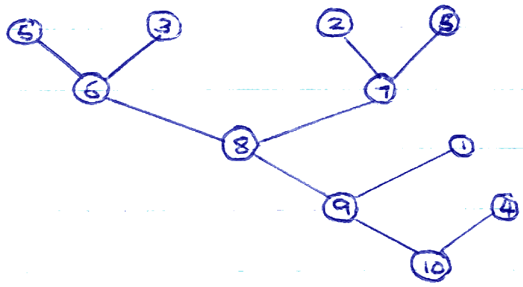
⑧ $\sim P(d, b, v) \vee ANS(f_3(d, b, c, v)) \vee ANS(f_2(d, b, c, v))$ ⑥ & ⑦

⑨ ⑧ & ①

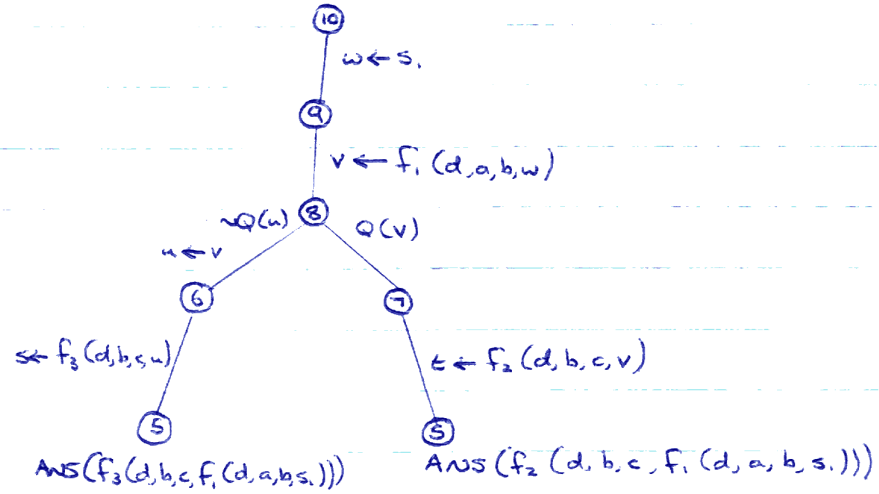
⑨ $\sim P(d,a,w) \vee ANS(f_3(d,b,c,f_1(d,a,b,w))) \vee ANS(f_2(d,b,c,f_1(d,a,b,w)))$

⑩ $ANS(f_3(d,b,c,f_1(d,a,b,s_1))) \vee ANS(f_2(d,b,c,f_1(d,a,b,s_1)))$ ⑨ & ④

Clause ⑩ is an answering clause. Its deduction is now represented by the tree below



Applying the algorithm to this tree, we get the decision tree:



8.6 Completeness of Resolution for Deriving Answers

Let us assume that the facts from which an answer is to be derived are represented by a set S_1 of clauses. Let a question be formulated as follows:

"Find values of $x_1, x_2, \ldots, x_n$ such that the relation $Q(x_1, \ldots, x_n)$ is true".

We shall say a question is answerable iff such values exist.

Let $S_2 \triangleq \{\sim Q(x_1, x_2, \ldots, x_n) \vee ANS(x_1, x_2, \ldots, x_n)\}$

Theorem: Let S be $S_1 \cup S_2$ where S_1 and S_2 are as above. The question is answerable iff there is a deduction of an answering clause from S.

9. Resolution Strategies. (cf pp 34-5)

We reconsider ways of conducting searches other than level saturation (breadth-first search)

9.1 Simplification Strategies (cf pp 34/5)

- elimination of tautologies
- elimination by evaluating predicates where possible
- elimination of subsumption

9.2 Ordering Strategies (heuristic search)

These strategies do not prohibit any particular type of resolutions but merely provide guidance about which ones should be performed first.

9.2.1 Unit-Preference Strategy

This attempts first to resolve units against units. If this succeeds we get an immediate refutation. If no units resolve it next attempts to resolve units against doubletons, etc. Whenever a pair of clauses resolve, the resolvent is checked against the units for a possible resolution before continuing. To prevent the search from continuing along an unprofitable chain of unit resolutions, a level bound is usually set. When the level bound allows no more unit resolution, then some other ordering scheme, such as doubletons against doubletons, etc. can be employed until additional units are generated. This strategy is motivated by the guaranteed shortening in the length of clauses caused by unit resolution.

9.2.2 Fewest-Components Strategy.

This strategy orders resolutions according to the length of the resolvents produced. Those two clauses that will produce the shortest resolvent are resolved first.

9.3 Refinement Strategies

These are based on the fact that not all possible resolutions need to be performed in order to find a refutation. Only resolutions between clauses meeting certain criteria need to be performed. Thus fewer resolutions need to be performed at each level.

9.3.1 Set-of-Support Strategy (cf p40)

A subset T of a set of clauses is called a set-of-support of S if S\T is satisfiable. The clauses in T are said to have support. A set-of-support resolution is a resolution of two clauses not both from S\T. A set-of-support deduction is a deduction in which every resolution is a set-of-support resolution.

A logical choice for T is the set of clauses originating from the negation of the theorem to be proved. The set S\T should then be the basic axioms of some theory that are assumed to form a satisfiable set. The set-of-support strategy avoids seeking a proof within that subset which is assumed to be itself satisfiable.

ANCESTRY-FILTERED FORM PROOFS (linear OL)

A resolution proof graph is a structure of nodes with each node corresponding to a clause. Those nodes in the graph having no ancestor are called tip nodes. If a graph displays a resolution proof of some clause from a set S of clauses, then the tip nodes correspond to clauses in S. (the base clauses of the proof).

A proof graph is in Vine form if each of its nodes is either a base clause or the immediate descendant of a base clause. If a refutation graph in Vine form exists, then in searching for it we need only perform resolutions between pairs of clauses such that at least one member is in S. Although a refinement strategy based on Vine-form refutation graphs, it is unfortunately not complete. The concept is therefore extended to graphs in ancestry-filtered (AF) form.

A refutation graph is in AF form if each node in the graph corresponds to either:

- a base clause
- an immediate descendant of a base clause
- an immediate descendant of two non-base clauses A and B such that B is an ancestor of A.

An AF-form refutation graph always exists for any unsatisfiable set of clauses. Therefore, a refinement strategy based on searching for AF-graphs is complete.

The AF-form strategy uses resolution relative to an ancestry-filter criterion as follows:

A top node is selected for the AF-graph. This node must be one that occurs in some refutation graph, so it is selected from some subset $K \subseteq S$ that contains only

clauses occurring in some repetation eg K might be those clauses originating from the negation of the theorem.
The criterion that must be satisfied by a pair of clauses (A, B) in order that they be resolved relative to the AF strategy is then

- one member of the pair (A, B) belongs to S and the other is either a clause in K or a descendant of a clause in K.
- one member of (A, B) is an ancestor of the other.

10 PROLOG PROGRAMMING

First-order logic can be used with :
- a procedural interpretation (eg PROLOG)
- a database interpretation (eg deductive db's)
- a process interpretation (eg concurrent PROLOG).

Its application in programs, data and concurrent processes could make logical inference the fundamental unit of computation (cf Japanese 5th Gen. project)

10.1 INTRODUCTION TO PROLOG.

PROLOG is used for solving problems involving objects and their relations. It is a declarative language: the user supplies *facts* and *rules* and asks *questions*. The PROLOG interpreter uses resolution and unification to compute answers. PROLOG is interactive

10.1.1 FACTS

Facts in PROLOG take the form: (args seperated by commas)

<predicate> (<argument list>).

The names of all predicates and objects must begin with lower case letters. Obviously the name given a predicate is arbitrary: we must decide on their interpretations and ensure they are used correctly. (ie facts express arbitrary relationships between objects)
A collection of facts is a *database*.

eg
```
likes (mary, john).
valuable ('life').
female (mary).
```

10.1.2 QUESTIONS.

Questions are like facts except preceded by '?-'. (questions can be lists of facts' - see later). PROLOG attempts to find a match directly a indirectly in the database. If a match is found PROLOG responds yes, otherwise PROLOG responds no (not provable, rather than not true).

eg: Given the database:
```
human (socrates).
human (aristotle).
athenian (socrates).
```

We could have the following session:
```
?- athenian (socrates).
yes
?- athenian (aristotle).
no
?- greek (socrates).
no
```

10.1.3 VARIABLES

Any name beginning with an upper case letter is taken to be a variable. Variables behave something like wild cards. Eg, to find the name of an athenian, we could type:
```
?- athenian (Ath).
Ath = socrates;
no
```

By typing a semicolon and CR, we instruct PROLOG to continue searching for further matches. Eg:
```
?- human (X).
X = socrates;
X = aristotle;
no
```

Any question can contain objects and variables freely mixed. The same variable can be used more than once in a question

If we have the database:

```
likes (john, mary).
likes (john, jane).
likes (jane, john).
likes (john, john).
```

We could ask questions like:

```
?- likes (john, x).
x = mary;
x = jane;
x = john;
no
```

/* Who does john like? */

NB Prolog comments structure

```
?- likes (x, y).
x = john, y = mary;
x = john, y = jane;
x = jane, y = john;
x = john, y = john;
no
```

/* Who likes who? */

```
?- likes (x, x).
x = john
no
```

/* Who likes themselves? */

```
?- likes (john, _ )
yes
```

/* Does john like anyone? */

```
?- likes (x, y), likes (y, x).
x = john, y = jane
```

/* What pair of "people" (objects) like each other? */

A comma seperating clauses is treated as a logical "and". The underscore is used as an "anonymous" variable, and is used when we just want to know if a match exists, without needing to know what it is.

10.1.4 RULES

A rule is a general statement about objects and their relationships. A rule consists of a head and a body, connected by ':-'. (":-" "if")

Eg:

```
sister_of (x, y) :-
        female (x),
        parents (x, Mother, Father),
        parents (y, Mother, Father).
```

We shall use the word clause wherever we refer to a fact or a rule.

10.2 A CLOSER LOOK.

10.2.1 SYNTAX.

PROLOG programs are built from terms (constants, variables and structures)

CONSTANTS

Constants name specific objects or specific relationships. There are two types: atoms and integers. Atoms are lowercase strings or strings of symbols (as well as special symbols eg $). Quoted atoms can have any characters.

Eg. Atoms	Not atoms
a	2dogs
a_void_ed	a-void
=	A_void
--->	_me
'GOD'	

Integers are whole numbers consisting only of digits and may not contain a decimal point.

VARIABLES

Variables are like atoms except they begin with upper case letters or the underscore. Unlike normal variables, the anonomous variable cannot have multiple instantiations ie likes(_,_) will return yes if any likes fact occurs in the database; not just if there is someone who likes themselves.

STRUCTURES

A structure is a collection of component objects into a single object. A structure is written in prolog by specifying its functor and components

<functor> (<component list>)

Note the correlation between predicates - a predicate is a functor of a structure (This is why all Prolog progs are made up of constants, variables and structures)

10.2.2 BASIC OPERATORS (INFIX)

Arithmetic	Comparisons
	X = Y
X + Y	X \= Y
X - Y	X < Y
X * Y	X =< Y
X / Y	X > Y
X mod Y	X >= Y

10.2.3 SUMMARY OF GOAL SATISFYING.

A question provides a conjunction of goals to be satisfied. Prolog uses the known clauses to satisfy the goals. A fact can cause a goal to be satisfied immediately, whereas a rule can only reduce the task to that of satisfying a conjunction of subgoals. A clause can only be used if it matches the goal under consideration. If a goal cannot be satisfied, backtracking occurs - this consists of reviewing what has been done, attempting to resatisfy the goals by finding an alternative way to satisfy them. Typing a semicolon forces Prolog to backtrack.

SATISFYING A CONJUNCTION OF GOALS.

Prolog attempts to satisfy goals from left to right (it will not attempt to satisfy a goal until its neighbour on the left has been satisfied; when it has been satisfied, P. will attempt to satisfy its right neighbour). As each goal is satisfied, a marker is placed in the database at the place where a satisfying clause was found.
When a failure is generated or forced (;), the current goal is abandoned, and Prolog attempts to satisfy the previous goal with a different clause, starting from the marked position. This backtracking can extend through all levels of the search. When a goal fails, it reports failure to its left neighbour; if it has no left neighbour, it reports to the goal which caused its rule to be used.
The rules for deciding whether a goal matches the head of a use of a clause are as follows. Note that in the use of a clause, all variables are originally uninstantiated:

- an uninstantiated variable will match any object.
  As a result, the variable will be instantiated to that object
- otherwise, an integer or atom will match only itself
- otherwise a structure will match another structure with the same functor and number of arguments, and all the corresponding arguments must match.

A noteworthy case in matching is one in which two uninstantiated variables are matched together. In this case we say that the variables share. Two sharing variables are such that as soon as one is instantiated, so is the other (with the same value)

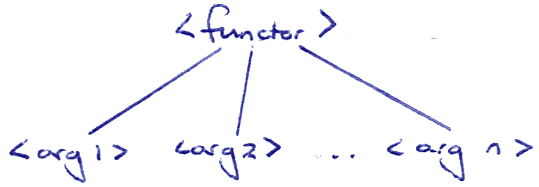
10.3 Using Data Structures

10.3.1 Structures and Trees

A structure in Prolog represents a tree

ie <functor> (<arg1>, <arg2>, ..., <arg n>)

is effectively



where each of the arguments could also be a structure (tree).

10.3.2 Lists

Lists can be represented as trees. Lists are either empty or a structures having two components, the head and tail. The empty list is written as [] . The head and tail are components of the functor . . Thus the list consisting of one elt 'a' is . (a, []), with the tree a ; []

The list consisting of the atoms a, b and c could be written . (a, . (b, . (c, [])))

The dot functor is also defined as an right-associative operator, so we could have written a . b . c . [] . or [a, b, c] where the latter is a list notation.

Additional notation : [X|Y] "the list with head X and tail Y"

eg
```
p([1,2,3]).
p([the, cat, sat, [on, the, mat]]).
?- p([X|Y]).
X = 1 , Y = [2,3] ;
X = the, Y = [cat, sat, [on, the, mat]]
?- p([_,_,_,[_|X]]).
X = [the, mat]
```

Lists are used by Prolog for string storage. For example, Prolog stores the string "system" as [115,121,115,116,101,109]
We could write a Prolog predicate member(X,Y) if the object X is a member of the list Y as follows

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y)
```

Example: A program to "reply" to input.

Database

```
change(you,i).
change(are,[am,not]).
change(french,german).
change(do,no).
change(X,X)          /* this is the "catchall" */
alter([],[]).
alter([H|T],[X|Y]) :- change(H,X),
                      alter(T,Y).
```

```
?- alter([you,are,a,computer],Z).
Z = [i,[am,not],a,computer]
?- alter([do,you,know,french],X).
X = [no,i,know,german]
```

10.4 BACKTRACKING AND CUT.

If we had used member (above) on the following:

```
?- member(a,[a,b,r,a,c,a,d,a,b,r,a]).
```

five matches would have been returned. The "cut" can be used. A cut is used as a goal clause (written !) and represents a goal which is immediately satisfied and cannot be resatisfied (ie no backtracking can occur)
In other words, when a cut is encountered as a goal, the system becomes committed to all choices made since

the parent goal was invoked. All other alternatives are discarded. Hence an attempt to resatisfy any goal between the parent goal and the cut goal will fail.

The three main uses for cuts are:

- to tell P. that it has found the right rule for a particular goal (search)
- to tell P. to fail a particular goal immediately without trying for alternatives (search)
- to tell P. to stop generating alternatives (committing search).

10.4.1 CONFIRMING THE CHOICE OF A RULE.

Consider a P. prog. to sum integers, so that ?- sum_to(5,X) returns X=15. The program is:

```
sum_to(1,1) :- !.        terminates recursion & initialises result to 1.
sum_to(N,Res) :-
        N1 is N-1,
        sum_to(N1,Res1),
        Res is Res1+N.
```

Here the cut is used to terminate the recursion. Prolog always tries to satisfy the first rule first. In general, the predicate not can replace cut, eg

```
sum_to(1,1).
sum_to(N,R) :- not(N=1), N1 is N-1, sum_to(N1,R1), R is N1+R.
```

10.4.2 THE 'CUT-FAIL' COMBINATION.

fail is a predicate that takes no arguments, always fails, and causes backtracking to occur. A cut-fail combination can be used to force failure. We could

this defines: not(P) :- call(P), !, fail.
not(P).

10.4.3 TERMINATING A 'GENERATE & TEST'

It can happen that if we find we can satisfy a goal in one way, we need never have to satisfy it again. We would thus like the first solution found to be fixed, so that failures later on that cause backtracking will not attempt to resatisfy this goal. We can do this by using a cut immediately after the goal.

Note that using cut can be dangerous - it should only be used if there is a clear policy of how the rules (i.e. what context) are going to be used.

10.5 INPUT & OUTPUT

nl. - forces a new line
tab(x) - moves right by x spaces
write(x) - if x is instantiated, it is printed out,
else a unique number (the address?) is printed.
display(x) - as for write but prints in structure form

Eg: ?- write(a+b*c*c), nl, display(a+b*c*c), nl.

```
a+b*c*c
+(a,*(*(b,c),c))
yes
```

These predicates cannot be resatisfied (i.e. ; causes them to fail)

read(X) - reads the next term typed in, terminated by a dot . and a non-printing character (eg space or CR). If X is already instantiated, the next term will be read, and matched against X.

Example: Database: event(1505, ['Euclid', translated, into, 'Latin']).
event(1523, ['Christian', 'II', flees, from, 'Denmark']).

Rules:

```
phh([]):-nl.
phh([H|T]):- write(H), tab(1), phh(T).
hello :-
        phh(['What', date, do, you, 'desire?']),
        read(D),
        event(D,S),
        phh(S).
```

```
?- hello.
What date do you desire?
1523.
Christian II flees from Denmark
```

put(X) writes out the character whose ascii code is in X.

Example: Write & display cannot be used for printing strings due to the representation of strings. We could define:

```
printstring([]).
printstring([H|T]):- put(H), printstring(T).
```

to do this.

get0(x) - instantiates x to the next character typed in.
get(x) - " " " " " printing character " ".
(both match if x is already instantiated)

File I/O

Output

tell(x) - selects x as the current output file.
telling(x) - returns yes if x is the current output file
told - closes a file, reselects standard output.

Input

see(x)
seeing(x)
seen
} see output.

consult(x) - reads clauses from file x into database.

Declaring Operators.

If we wish to declare an operator with a given position, precedence and associativity, we use the built in predicate op:

?- op(<precedence>, <specifier>, <name>)
(numeric)

returns yes if the declaration is legal. The specifier is an atom made up of up to three non-repeating letters, x, f and y. f specifies the function (operator), and y the higher-precedence arg.

eg
- yfx — left associative infix
- xfy — right associative infix
- fyx — left associative prefix
- xyf — right associative postfix
- fx — unary prefix with f taking precedence (eg not, -)
- fy — unary prefix with y taking precedence

10.6 Built-in Predicates

10.6.1 Loading Clauses

consult (x) - augments the database with clauses from file x
reconsult (x) - supersedes the database defns "

These can be done simply as a list of file names with
- preceding any file that is to be reconsulted

eg
```
?- [file1, -file2, 'fred.1'].
```
is equivalent to:
```
?- consult(file1), reconsult(file2), consult('fred.1').
```

10.6.2 Success & Failure

true - this goal always succeeds
fail - this goal always fails

10.6.3 Classifying Terms

var (x) succeeds if X is currently an uninstantiated variable
nonvar (x) opposite of var (x)
atom (x) succeeds if X is an atom
integer (x) " " " " integer
atomic (x) - succeeds if X is either an atom or an integer

10.6.4 Treating Clauses as Terms

listing (A) - lists all clauses in db containing atom A as predicate
clause (X,Y) - matches against clauses with head X and body Y. A clause without a body has the implicit body true.

asserta (X) - adds clause X to the db at the beginning

assertz (X) - " " " " " " " end.

retract (X) - removes a clause from the db.

10.6.5 Creating & Accessing Components of General Structures

functor (T, F, N) "T is a structure with functor F and arity N"
can be used as functor (t, F, N) matches structure
or functor (T, f, n) creates a structure

arg (n, t, A) returns nth argument of structure t in A.

X =.. L "L is the list consisting of the functor of X followed by the arguments of X. If X is instantiated, P. creates the appropriate list and matches it with L. If X is uninstantiated, the list is used to construct an appropriate structure for X to stand for.

name (A, L) "L is the list of characters making up the name of atom A".

10.6.6 Affecting Backtracking

! Cut. See 10.4

repeat repeat always succeeds, even on backtracking. It is defined as

```
repeat.
repeat :- repeat.
```

In backtracking, repeat causes all goals after it to be repeated, but none before. It thus can act as a looping construct, particularly when used with cut.

10.6.7 Creating Complex Goals.

,	conjunction
;	disjunction (can be dangerous to use with cuts & backtracking)
call(X)	attempts to satisfy goal X (cf eval in Lisp)
not(X)	

10.6.8 Equality

=	equals
\=	not equals
==	strong equals (ie uninstantiated variables match other only if they are sharing)
\==	weak not equals (not(X == Y))

10.6.9 Input & Output *

get0(X)	nl
get(X)	tab(X)
skip(X)	skips till an X is read.
read(X)	write(X)
put(X)	display(X)
op(X,Y,Z)	

10.6.10 Handling Files

see(X)	tell(X)
seeing(X)	telling(X)
seen	told

* remember these are goals, and act as functions only if their arguments are uninstantiated variables (and hence match anything).

10.6.11 EVALUATING ARITHMETIC EXPRESSIONS.

X is Y	Y must be instantiated to an arith. expr. structure.
X + Y	
X - Y	
X * Y	
X / Y	
X mod Y	

10.6.12 COMPARISONS

X = Y	X < Y	X > Y
X \= Y	X =< Y	X >= Y

10.6.13 DEBUGGING

trace	
notrace	
spy P	P a structure, atom or list. Sets a spy point at that predicate (atom, functor, or all list elts)
debugging	prints a list of spy points
nodebug	removes spy points
nospy P	removes specific spy points

10.7 Logic Programming with Prolog.

(cont.) Usual form consists of a collection (conjunction) of clauses each of which is a collection (disjunction) of literals, each of which is an atomic formula or the negation of an atomic formula.

As a notation for clauses, we can use the following. We write unnegated literals first seperated by ';'s, followed by the sign ':-', followed by the negated literals without their '~'s and seperated by ','s. In this notation, a clause with negated literals K, L, ... and unnegated literals A, B, ... would come out as:

A ; B ; ... :- K, L, ...

The reasoning behind this notation (using the usual Prolog interpretations of ; , and :-) is:

A ; B ; ... :- K, L, ...
(or; is implied by; and)

≡ K₁ ∧ L ∧ ... → A ∨ B ∨ ... (using FOL notation)

≡ (A ∨ B ∨ ...) ∨ ~(K ∧ L ∧ ...)

≡ A ∨ B ∨ ... ∨ ~K ∨ ~L ∨ ...

A Horn clause is a clause with at most one unnegated literal. There are thus two types of Horn clauses - those with one unnegated literal (headed; of the form 'A :- ...') and those with none (headless; of the form ':- ...').

As can be seen, all clauses in Prolog are Horn clauses. Headed clauses can be expressed directly in Prolog.

A headless clause must be a goal, and corresponds to a question (ie of the form '?- ...')

When we consider sets of Horn clauses (including goal statements), we need only consider those sets where all but one of the clauses are headed. In other words, any

solveable problem (theorem-proving task) that can be expressed in Horn clauses can be expressed in such a way that

- there is one headless clause
- all other clauses are headed.

Since it is arbitrary how we decide which clauses are actually the goals, the headless clause is used (cf question representation above).

It is easy to see that at least one headless clause must be present for a problem to be solveable, as the result of resolving two headed Horn clauses is itself a headed Horn clause, while the empty clause (what we wish to derive) is headless. Furthermore, it can be shown that if there are several headless clauses among our assumptions, any Resolution proof of a new clause can be converted into a proof using at most one of them.

Prolog attempts to resolve Horn clauses using linear input resolution, where the latest clause derived can be seen as the conjunction of goals yet to be satisfied. Prolog's strategy is a kind of SL resolution - the literal to be matched is always the first one in the goal clause. Prolog places new derived goals at the front of the goal clause (i.e. Prolog finishes satisfying subgoals first). For space and implementation reasons, Prolog uses a depth first search. Prolog matching is also slightly different to unification, in that Prolog will allow a variable to be instantiated to something containing itself. eg:

```
equal(X, X)
?- equal(foo(Y), Y)
```

can be satisfied by Prolog but not by resolution (provided Prolog does not attempt to print the value of Y)

Prolog succeeds partially as a logic programming (purely declarative) language. The problems arise with the use of built in predicates (particularly cut) which express control information. Thus there are problem areas - however, careful use of Prolog can result in programs which are almost entirely declarative.