AUTOMATED REASONING

SLIDES 3:

RESOLUTION
- **The resolution rule**
- **Unification**
- **Refutation by resolution**
- **Factoring**
- **Clausal Form and Skolemisation**

KB - AR - 09

Resolution

3ai

Resolution is a *clausal refutation system* (it tries to derive False from Givens:)

Some Notations for Clauses

A clause has the form A1 ∨ A2 ∨... ∨An, where each Ai is a literal.
A literal is either an atom or a negated atom.
All variables in a clause are implicitly universally quantified.
e.g. $P(x,f(x)) \vee \neg R(x) \equiv \forall x(P(x,f(x)) \vee \neg R(x)) \equiv \forall x(R(x) \to P(x,f(x)))$
(Variables will start x,y,..,z; other terms are constants. e.g. a,b, f(...), etc.)

A clause with no literals is called the empty clause and often denoted [] .
The empty clause is always false. (e.g. it is derived from A and ¬A.)

(Clauses will sometimes be represented as sets
e.g. {A,C, B} ≡ A ∨ C ∨ B or more simply as ACB)

Resolution is "Modus Ponens" or (→E) generalised to first order logic:

e.g. without variables first (ie propositional logic)

A, A → B ==> B	A, ¬A ∨ B ==> B
¬B, A → B ==> ¬A	¬B, ¬A ∨ B ==> ¬A
A → B, B →C ==> A →C	¬A ∨ B, ¬B ∨ C ==> ¬A ∨ C

A ∧ D → B, B→ C ∨ E ==> A ∧ D → C∨ E,
¬A ∨ ¬D ∨ B, ¬B ∨ C∨ E ==> ¬A∨ ¬D ∨ C ∨ E

<u>**Resolution:**</u>

3aii

These slides detail the *Resolution rule*, which was proposed by Alan Robinson in 1963. Resolution is the backbone of the Otter family of theorem provers and many others besides. It is also, in a restricted form, the principal rule used in Prolog. In order to form a resolvent, it is necessary to be able to unify two (or more) literals. The unification algorithm is shown on 3aiii and is used in Prolog, so you should already be familiar with it.

Resolution can be thought of as a generalisation of the transitivity property of →. That is, from A → B and B → C derive A→ C.

The rule on slide 3aiv is called *Binary Resolution*. Robinson actually proposed a more flexible version, which allowed several literals to be unified within each of the two clauses to give the literals ¬G and E, before forming the binary resolvent. This initial step of unifying literals is called *factoring*, and is more usually performed as a separate step in theorem provers. See Slide 3cii for the factoring rule.

Resolution requires the data to be clauses, and in slides 3dii you'll see how to achieve clausal form from arbitrary first order sentences using a process called Skolemisation.

Binary Resolution:

3aiii

Given clauses C1 = ¬G ∨ H and C2 = E ∨ F,
where E and G are literals and H and F are clauses or literals.

Example:
(1) P(x, f(x)) ∨ ¬ R(x) (or R(x) → P(x, f(x)) Use u-z for variables)
(2) ¬P(a, y) ∨ S(g(y)) (or P(a,y) → S(g(y)) Use a...m for constants)

- *Unify* (a,y) with (x,f(x)) to give {x == a , y == f(a)} (or {x/a, y/f(a)}
- *Instantiate* (1) giving ¬R(a) ∨ <u>P(a, f(a))</u> (or R(a)→ P(a,f(a)))
- Instantiate (2) giving <u>¬P(a, f(a))</u> ∨ S(g(f(a))) (or P(a,f(a))→ S(g(f(a))))
- Derive ¬R(a) ∨ S(g(f(a))) (or R(a) → S(g(f(a))) by transitivity of →)

The **binary resolvent** of C1 and C2 (R(C1,C2)) is (H ∨ F)θ, where θ=mgu(E,G); ie θ makes E and G identical and is computed by unification.

(1) and (2) *resolve* to give ¬R(a) ∨ S(g(f(a)))

FIRST "match" a positive and negative literal by unifying them,
NEXT apply the substitution to the other literals,
THEN remove the complementary literals and take disjunction of rest.

The Unification Algorithm

3aiv

To unify P(a1,...,an) and ¬ P(b1,...,bn): (i.e. find the mgu (most general unifier))

first equate corresponding arguments to give equations E (a1=b1, ..., an=bn)

Either reduce equations (eventually to the form var = term) by:

a) remove var = var;
b) mark var = term (or term = var) as the unifier var == term and
replace all occurrences of var in equations and RHS of unifiers by term;
c) replace f(args1) = f(args2) by equations equating corresponding argument terms;

or fail if:
d) term1 = term2 and functors are different; (eg f(...)=g(...) or a=b)
e) var = term and var occurs in term; (eg x=f(x) or x=h(gx)) – called *occurs check*)

Repeat until there are no equations left (*success*)
or d) or e) applies (*failure*).

UNIFICATION PRACTICE

(On this Slide variables are x,y,z,etc, constants are a,b,c, etc.}

Unify: 1. M(x,f(x)) , M(a,y) 2. M(y,y,b) , M(f(x), z, z)
3. M(y,y) , M(g(z) , z) 4. M(f(x), h(z), z) , M(f(g(u)), h(b), u)

RESOLUTION PRACTICE

Resolve:
1. P(a,b) ∨Q(c), ¬ P(a,b) ∨ R(d) ∨ E(a,b)
2. P(x,y) ∨ Q(y,x) , ¬P(a,b)
3. P(x,x)∨Q(f(x)) , ¬P(u,v)∨R(u)
4. P(f(u), g(u)) ∨ Q(u) , ¬ P(x,y) ∨ R(y,x)
5. P(u,u,c) ∨ P(d,u,v) , ¬ P(a,x,y) ∨ ¬ P(x,x,b)

To **Resolve** two clauses C and D:
FIRST "match" a literal in C with a literal in D of opposite sign,
NEXT apply the substitution to all other literals in C and D,
THEN form the resolvent R = C+D-{matched literals}.

3av

Logical Basis of Resolution

3avi

What should we do with (1) P(x,f(x)) ∨Q(x) and (2) ¬P(f(x),y)?

(1) ≡ ∀x[P(x,f(x)) ∨Q(x)]
(2) ≡ ∀x∀y[¬P(x,f(y)) ≡ ∀z∀v[¬P(z,f(v)]
Resolving ....
z==x, v==x and resolvent is Q(x) ≡ ∀x[Q(x)]

In general, variables in two clauses should be standardized apart – i.e. the variables are renamed so they are distinct between the two clauses

Refutation by Resolution

1. The aim of a resolution proof is to use resolution to derive from given clauses C the empty clause [ ], which represents False (ie show the clauses C are contradictory). The derivation is called a *refutation*.

2. The empty clause is derived by resolving two unit clauses of opposite sign - eg P(x,a) and ¬P(b,y).

Informally, P(x,a) is true for every x and P(b,y) is false for every y, so P(b,a) is true and P(b,a) is false - a contradiction.

Constructing Resolution Proofs:

3bi

Now that you know what resolution is, you may ask "how is a resolution proof constructed?" In fact, the ***Completeness Property*** of resolution says that for a set of unsatisfiable clauses a refuation does exist. (See Slides 4 for more on unsatisfiability for first order clauses.) So perhaps it is enough just to form resolvents as you fancy, and hope you eventually get the empty clause. This isn't very systematic and so it isn't guaranteed that you'll eventually find a refutation, even if one exists.

e.g. if S={P(f(x) ∨ ¬P(x), P(a), ¬P(a)}, then the sequence of resolvents P(f(a)), P(f(f(a))),... formed by continually resolving with the first clause won't lead to [] , even though resolving clauses 2 and 3 gives it immediately.

A systematic approach is obtained if the given clauses are first resolved with each other in all possible ways and then the resolvents are resolved in all possible ways with themselves and with the original clauses. Resolvents from this second stage are then resolved with each other and with all clauses, either given or derived as previous resolvents. This continues until the empty clause is generated, or no more clauses can be generated, or until one wishes to give up!

For example, a limit may be imposed on the number of clauses to be generated, on the size of clauses to be generated, on the number of stages completed, etc.

Saturation Search:

3bii

The method outlined on Slide 3biii is called *saturation search*. See Slide 3biv for an example. In this approach, we can say that the resolvents are generated in *groups*. The first group, S0 say, is the given clauses (for which a refutation is sought). The second group, S1, is the set of all resolvents that can be derived using clauses from S0. In general,

S0 = {C: C is a given clause}
Si (i>0) = {R: R is a resolvent formed from clauses in Sj, j<i,
and which uses at least one clause from Si -1}.
Continue until some Sj is reached containing the empty clause.

There is a wonderful theorem prover called OTTER (and its successor called Prover9) that you will use soon. This prover has a very basic strategy that employs the above saturation search.

It is easy to make resolution steps, but for a large problem (either many clauses or extra large clauses) the number of resolvents will increase rapidly. Therefore, some method is needed to decide which ones to generate, which ones not to generate, which ones to keep and which ones to throw away. There are many variations on the basic idea of Saturation search to address this issue, in which not all possible resolvents are found at each stage, but some are left out. It is then necessary to prove that this does not compromise being able to find a refutation. We'll look at these things a bit later.

A Simple Strategy – Saturation Search

3biii

How is a resolution proof made?

The simplest strategy is called a SATURATION refinement. All resolvents that can be formed from initial set of clauses S0 are formed giving S1, then all clauses that can be formed from S0 and S1 together are formed giving S2, etc.

Saturation refinement:

1) state S0 = given clauses S ;
2) to generate state Si (i ≥1) :
generate all resolvents involving at least one clause from Si-1;
3) increment i and repeat step 2 until a state contains [], or is empty.

Other possibilities (considered later) include:

Generate resolvents using the <u>previous</u> resolvent as one of the two clauses involved. This is called a LINEAR strategy.

Impose syntactic restrictions to control which resolvents are allowed and which are prohibited, or to indicate a preference for certain resolvents.
e.g. a preference for generating facts (clauses with a single literal).

Example of Saturation Search

3biv

<u>State S0</u> (given clauses)

1 Dca ∨ Dcb 2 ¬Dxy ∨ Cxy 3 ¬Tx ∨ ¬Cxb 4 Tc 5. ¬Dcz

<u>State S1</u> (resolvents formed from given clauses)

6 (1,2) Cca ∨ Dcb 7 (1,2) Ccb ∨ Dca
8 (1,5) Dcb 9 (2,3) ¬Dxb ∨ ¬Tx
10 (3,4) ¬Ccb 11 (1,5) Dca

<u>State S2</u> (resolvents formed from clauses in S1 with clauses in S0 or S1)

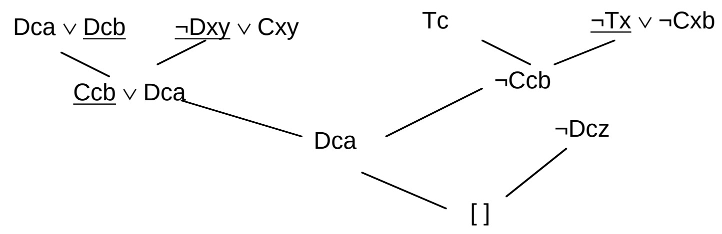
12 (8,2) Ccb 13 (8,9) ¬Tc 14 (8,5), (11,5) []
15 (9,4) ¬Dcb 16 (10,2) ¬Dcb 17. (11,2) Cca

There are some more possible resolvents in State S2. Which are they?

Notice that some resolvents subsume earlier clauses.
eg clause 8 subsumes 6 and 1

We can also present a resolution refutation as a tree

3bv



Each step is indicated by two parent clauses joined to the resolvent.
If an initial clause is used twice it is usually included in the tree twice, once in each place it is used.

The order in which the steps in a refutation are made does not matter, though of course a clause must be derived before it can be used!

It's clear we need to restrict things a little.......

3ci

For any but the smallest sets of clauses the number of resolution steps can be huge
So what can we do to reduce redundancy?

• Recall: at the ground level (no variables) we have a merge operation that removes duplicate literals from a clause.

eg $p \vee \neg q \vee p \vee \neg q \equiv p \vee \neg q$

In other words it simplifies a clause by removing redundant literals.

• The analogous and more general operation is called *Factoring*
• Unlike merge, factoring does not always preserve equivalence.

eg given $P(x) \vee P(y)$ and $\neg P(a) \vee \neg P(v)$
What resolvents can you form? (Remember to rename variables apart)

• Logically we can derive the empty clause:
$P(x) \vee P(y)$ means $\forall x \forall y[P(x) \vee P(y)]$ from which we can derive $\forall z.P(z)$
and $\neg P(a) \vee \neg P(v)$ means $\forall v[\neg P(a) \vee \neg P(v)]$ from which we can derive $\neg P(a)$
We *factor* by applying a binding to enable literals to be merged.

• We introduce factoring here since resolution on its own is not always sufficient to derive [ ] even when the given clauses are contradictory.

FACTORING

3cii

Given a clause $C = E1 \vee E2 \vee \dots \vee En \vee H$, where Ei are literals of the same sign,
F is a **factor** of C if $F=(E \vee H)\theta$, where θ=mgu{Ei} and $E= Ei\theta$ (for every i)

Example:
$P(x,a) \vee P(b,y)$ factors to $P(b,a)$
$P(x) \vee P(a)$ factors to $P(a)$
$Q(a,b) \vee Q(a,b)$ factors to $Q(a,b)$ (factoring identical literals is called merging)
$P(x,x) \vee P(a,y)$ factors to $P(a,a)$
$P(x,y) \vee P(x,x) \vee P(y,z)$ factors to $P(x,x) \vee P(x,z)$ and also to $P(x,x)$

FACTORING PRACTICE

Find factors of
1. $P(u,u,c) \vee P(d,u,v)$
2. $P(x,y) \vee P(z,x)$
3. $P(x,y) \vee \neg P(x,x) \vee P(y,z)$

Why are there no factors of 4. $P(x) \vee \neg P(f(x))$?

To Factor a clause C:
FIRST "match" two (or more) same sign literals in C,
NEXT apply the substitution to all other literals,
THEN merge the matched literals.

So Far ...

3di

A Typical refutation has the form

C0 = Initial clauses $\Rightarrow$ C0+intermediate resolvent or factor (C1)
$\Rightarrow$ C0 +C1 +C2 $\Rightarrow$ ... $\Rightarrow$ C0+C1+ … + Cn (= [])
Each Ci can use clauses in C0 and {Cj : j<i} to form resolvents or a factor.

(See Slides 4 for a more formal account.)

BUT:
• What if the given data is not a set of clauses?
Suppose you are given some Data and a conclusion in normal predicate logic?

• We know to show Data |= Conclusion, we can instead derive a contradiction from Data + ¬ Conclusion.
• So we need somehow to convert Data + ¬ Conclusion to clauses.

Conversion to Clausal Form

3dii

Conversion to clauses uses 6 basic steps:

1. *Eliminate* $\rightarrow$: $A \rightarrow B \Rightarrow \neg A \vee B$, $A \leftrightarrow B \Rightarrow (A \rightarrow B) \wedge (B \rightarrow A)$.
$\neg (A \wedge B) \Rightarrow \neg A \vee \neg B$ (and similar rewrites to push ¬ inwards).
2. *Rename quantified variables* to be distinct.
3. *Skolemise* - remove existential-type quantifiers and replace bound variable occurrences of x in $\exists x S$ by *Skolem constants* or *Skolem functions*. The latter are dependent on universal variables in whose scope they lie and which also occur in S. (See 3diii)
4. *Move universal quantifiers into a prefix*:
$A \text{ op } \forall x P[x] \Rightarrow \forall x[A \text{ op } P[x]]$, etc.
5. *Convert to CNF*(conjunctive normal form) - conjunctions of disjunctions using distributivity: $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (B \vee A)$, etc.
6. *Re-distribute universal quantifiers* across $\wedge$.

Skolemisation is a process that gives a name to something "that exists".

eg1: We may be told that "there's someone who lives in NY and has 2 children and a dog and ....". We can refer to this individual as "a" for short.

eg2: Given $\exists x \exists y[person(x) \wedge place(y) \wedge lives(x,y)]$, we can introduce the new names "a" and "t" and write $person(a) \wedge place(t) \wedge lives(a,t)$.

More on Skolemisation

Skolemisation can seem mysterious, but it is not really so.

For instance: given $\forall x \exists y.lives(x,y)$ (meaning everyone lives in some place), we may have $\exists y.lives(kb, y)$, $\exists y.lives(ar, y)$, $\exists y.lives(pp, y)$, etc.

Skolemisation is a process that gives a name to something "that exists". It is important that the given name is NEW and not previously mentioned.

eg $\exists y.P(x)$ Skolemises to $P(a)$, where "a" is a new name called a *Skolem constant* which is not already in the signature.

Skolemising each of $\exists y.lives(kb, y)$, $\exists y.lives(ar, y)$, $\exists y.lives(pp, y)$, etc. we might get lives(kb,pkb), lives(ar,par), lives(pp,ppp), etc.

These can be captured more uniformly as $\forall x.lives(x, plc(x))$, where plc(x) is a new *Skolem function* that names the place where x lives.

So we get lives(kb,plc(kb)), lives(ar,plc(ar)), lives(pp,plc(pp)), etc.

All the conversion steps except Step 3 (Skolemisation) maintain equivalence, so we don't have $S \equiv$ converted(S). What we want is that converted (S) are contradictory if and only if (iff) S are contradictory. And this property **is** true. (See Slides Appendix 1 for details.)

PRACTICE IN CONVERSION TO CLAUSAL FORM

Convert to clausal form:
1. $\forall x\, [\, \exists y S(x,y) \leftrightarrow \neg P(x)\,]$ done below
2. $\forall z\, [\, P(z) \rightarrow R(z)\,] \rightarrow Q(a)$
3. $\forall x\, [\, P(x) \vee R(x) \rightarrow \exists y \forall w\, [\, Q(y,w,x)\,]\,]$

$\forall x\, [\, \exists y S(x,y) \leftrightarrow \neg P(x)\,]$

(convert $\leftrightarrow$) $\forall x\, [(\exists y S(x,y) \rightarrow \neg P(x)) \wedge (\neg P(x) \rightarrow \exists y S(x,y))]$

(convert $\rightarrow$) $\forall x\, [(\neg \exists y S(x,y) \vee \neg P(x)) \wedge (\neg \neg P(x) \vee \exists y S(x,y)\,)]$

(move $\neg$) $\forall x\, [(\forall y \neg S(x,y) \vee \neg P(x)\,) \wedge (P(x) \vee \exists y S(x,y))\,]$

(rename quantifiers) $\forall x\, [(\forall z \neg S(x,z) \vee \neg P(x)\,) \wedge (P(x) \vee \exists y S(x,y))\,]$

(Skolemise $\exists y S(x,y)$) $\forall x\, [(\forall z \neg S(x,z) \vee \neg P(x)) \wedge (P(x) \vee S(x,f(x)))\,]$

(Pull out quantifiers) $\forall x \forall z[\, (\neg S(x,z) \vee \neg P(x)\,) \wedge (P(x) \vee S(x,f(x))\,)]$

(Redistribute $\forall x\ \forall z$) $\forall x \forall z[\, \neg S(x,z) \vee \neg P(x)\,] \wedge \forall x[P(x) \vee S(x,f(x))\,]$

NOTE: there are many ways to Skolemise; in step 3 on 3dii the Skolem function is made to be dependent only on those universal variables in whose scope it lies. eg $\forall x\, [\, P(x) \vee \exists y\, Q(y)\,]$ Skolemises to $\forall x\, [\, P(x) \vee Q(a)]$ with the rules here, as x doesn't occur in $\exists y\, Q(y)$, not to $\forall x\, [\, P(x) \vee Q(f(x))\,]$.

More SKOLEMISATION Examples

$\forall z\, [\, P(z) \rightarrow R(z)\,] \rightarrow Q(a) \Rightarrow \neg(\forall z[P(z) \rightarrow R(z)]) \vee Q(a) \Rightarrow$

$\exists z[\neg(P(z) \rightarrow R(z))] \vee Q(a) \Rightarrow \exists z[P(z) \wedge \neg R(z)] \vee Q(a)$
(all by step 1) (no need for step 2, 1 bound variable)

$\Rightarrow (P(c) \wedge \neg R(c)) \vee Q(a)$ (by step 3, c is a new constant) (no need for step 4)

$\Rightarrow (P(c) \vee Q(a)) \wedge (\neg R(c) \vee Q(a))$ (by step 5) (no need for step 6)

$\forall x[P(x) \vee R(x) \rightarrow \exists y \forall w\, [Q(y,w,x)]] \Rightarrow \forall x[\neg(P(x) \vee R(x)) \vee \exists y \forall w\, [Q(y,w,x)]]$

$\Rightarrow \forall x[(\neg P(x) \wedge \neg R(x)) \vee \exists y \forall w\, [Q(y,w,x)]]$
(by step 1) (no need for step 2, all bound variables different)

$\Rightarrow \forall x[(\neg P(x) \wedge \neg R(x)) \vee \forall w[Q(f(x),w,x)]]$
(by step 3, f is new functor, y replaced by f(x) as y in scope of x)

$\Rightarrow \forall x \forall w[(\neg P(x) \wedge \neg R(x)) \vee Q(f(x),w,x)]$ (step 4)

$\Rightarrow \forall x \forall w[(\neg P(x) \vee Q(f(x),w,x)) \wedge (\neg R(x)) \vee Q(f(x),w,x))]$ (step 5)

$\Rightarrow \forall x \forall w[\neg P(x) \vee Q(f(x),w,x)] \wedge \forall x \forall w[\neg R(x)) \vee Q(f(x),w,x)]$ (step 6)

Summary of Slides 3:

1. Resolution is an inference rule between 2 clauses. It unifies two complementary literals and derives the resolvent clause consisting of the remaining literals in the two parent clauses.

2. Factoring is a related inference rule using a single clause. It unifies one or more literals in the clause that are of the same sign and results in the instance obtained by applying the unifier to the parent clause.

3. Conversion to clausal form is a 6 step process, that uses Skolemisation to eliminate existential quantifiers.

4. The unification algorithm applied to two literals produces the most general unifier (mgu) of the two literals.

5. Resolution derivations are usually constructed using a systematic search process called saturation search, in which resolvents and factors are produced in stages, all steps possible at each stage being made before moving to the next stage. This procedure prevents the same step from being taken more than once (but does not necessarily prevent the same clause from being derived in different ways).

6. More restrictions are needed on which resolvents and factors to generate.

7. Resolution derivations can be depicted as a tree.