

AUTOMATED REASONING

SLIDES 3:

RESOLUTION

The resolution rule

Unification

Refutation by resolution

Factoring

Clausal Form and Skolemisation

KB - AR - 12

Resolution for propositional clauses

3ai

Recall the structure of a Clause

A clause has the form $A1 \vee A2 \vee \dots \vee A_n$, where each A_i is a literal

A literal is either an atom or a negated atom (from a language L)

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

Let S be a set of clauses

We'll see that if $S \vdash \text{False}$ by resolution, then $S \models \text{False}$ (called Soundness)

But $S \models \text{False}$ means S has no models and is unsatisfiable

So resolution is a way to show unsatisfiability of S

The Resolution Rule for prop. logic is essentially "Modus Ponens" or $(\rightarrow E)$

$A, \neg A \vee B \implies B$

$A, A \rightarrow B \implies B$

$\neg B, \neg A \vee B \implies \neg A$

$\neg B, A \rightarrow B \implies \neg A$

$\neg A \vee B, \neg B \vee C \implies \neg A \vee C$

$A \rightarrow B, B \rightarrow C \implies A \rightarrow C$

$\neg A \vee \neg D \vee B, \neg B \vee C \vee E$

$A \wedge D \rightarrow B, B \rightarrow C \vee E$

$\implies \neg A \vee \neg D \vee C \vee E$

$\implies A \wedge D \rightarrow C \vee E$

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 $\neg A$.)

(Clauses may occasionally be written as sets e.g. $A \vee C \vee B \equiv \{A, C, B\}$ or ACB)

First order Clauses and Resolution

3aii

First order Clauses

A clause has the form $A1 \vee A2 \vee \dots \vee A_n$, where each A_i is a literal

A literal is either an atom or a negated atom

An atom is $\text{predicate}(\text{argument1}, \text{arg2}, \dots, \text{argn})$

Clauses may be ground (no variables)

Clauses may have variables which are all implicitly universally quantified.

e.g. $\text{less}(x, s(x)) \vee \neg \text{pos}(x) \equiv \forall x (\text{less}(x, s(x)) \vee \neg \text{pos}(x))$
 $[\equiv \forall x (\text{pos}(x) \rightarrow \text{less}(x, s(x)))]$

Generally, the statement $\forall x (P(x))$ is read as "for every x $P(x)$ is true",
or "for each substitution for x the sentence is true".

e.g. $P(x, s(x)) \vee \neg R(x)$ stands for many ground clauses, where x has been

instantiated to a ground term in the language (more formally on slides 4)

e.g. $P(a, s(a)) \vee \neg R(a)$, $P(s(a), s(s(a))) \vee \neg R(s(a))$, $P(b, s(b)) \vee \neg R(b)$, etc.

NOTE: Variables will start u, v, w, x, y, z ;

other terms are constants or functional terms. e.g. $a, b, f(\dots)$, etc.

Resolution:

3aiii

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 Slide 3avi; as it is used in Prolog you should already be familiar with it.

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

The rule on slide 3av 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 $\neg G$ and E , before forming the binary resolvent. This initial step of unifying literals is called *factoring*, and is now 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/3diii you'll see how to achieve clausal form from arbitrary first order sentences using a process called Skolemisation.

Binary Resolution:

3aiv

The structure of the Resolution Rule was $\neg A \vee B, \neg B \vee C \implies \neg A \vee C$

But consider $P(x, s(x)) \vee \neg R(x)$ and $R(f(a)) \vee \neg Q(y)$?

Although there are two literals of opposite sign and the same predicate their arguments are not equal.

Remember that variables such as x and y are universally quantified. Maybe we can find values for x and y so $R(x)$ and $R(f(a))$ are identical?

We can! It's done by **unification** and here it gives $x = f(a)$

What will the resolvent be?

It won't be $P(x, s(x)) \vee \neg Q(y)$, as we just required $x = f(a)$

The resolvent is $P(f(a), s(f(a))) \vee \neg Q(y)$

That is, substitute $f(a)$ for x everywhere it occurs

This gives us first order resolution!

The Unification Algorithm

3avi

To unify $P(a_1, \dots, a_n)$ and $\neg P(b_1, \dots, b_n)$: (i.e. find the mgu (most general unifier))

first equate corresponding arguments to give equations E ($a_1 = b_1, \dots, a_n = b_n$)

Either reduce equations (eventually to the form $\text{var} = \text{term}$) by:

- remove $\text{var} = \text{var}$;
- mark $\text{var} = \text{term}$ (or $\text{term} = \text{var}$) as the **unifier** $\text{var} = \text{term}$ and replace all occurrences of var in equations and RHS of unifiers by term ;
- replace $f(\text{args}_1) = f(\text{args}_2)$ by equations equating corresponding argument terms;

or fail if:

- $\text{term}_1 = \text{term}_2$ and functors are different; (eg $f(\dots) = g(\dots)$ or $a = b$)
- $\text{var} = \text{term}$ and var occurs in term ; (eg $x = f(x)$ or $x = h(g(x))$) – called *occurs check*

Repeat until there are no equations left (*success*)

or d) or e) applies (*failure*).

Binary Resolution:

3av

Given clauses $C1 = \neg G \vee H$ and $C2 = E \vee F$,
where E and G are atoms and H and F are clauses of none or more literals.

Example:

- (1) $P(x, f(x)) \vee \neg R(x)$ (Use $u-z$ for variables)
- (2) $\neg P(a, y) \vee S(g(y))$ (Use $a \dots m$ for constants)
- (i) Unify (a, y) with $(x, f(x))$ to give $\{x = a, y = f(a)\}$ (or $\{x/a, y/f(a)\}$)
- (ii) Instantiate (1) giving $\neg R(a) \vee P(a, f(a))$ (use $x = a$)
- (iii) Instantiate (2) giving $\neg P(a, f(a)) \vee S(g(f(a)))$ (use $y = f(a)$)
- (iv) Derive $\neg R(a) \vee S(g(f(a)))$ (by resolution)

(1) and (2) resolve to give $\neg R(a) \vee S(g(f(a)))$

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 **binary resolvent** of $C1$ and $C2$ ($R(C1, C2)$) is $(H \vee F)\theta$, where $\theta = \text{mgu}(E, G)$; ie θ makes E and G identical and is computed by unification.

UNIFICATION PRACTICE

(See ppt)

(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. $P(a, b) \vee Q(c)$, $\neg P(a, b) \vee R(d) \vee E(a, b)$
2. $P(x, y) \vee Q(y, x)$, $\neg P(a, b)$
3. $P(x, x) \vee Q(f(x))$, $\neg P(u, v) \vee R(u)$
4. $P(f(u), g(u)) \vee Q(u)$, $\neg P(x, y) \vee R(y, x)$
5. $P(u, u, c) \vee P(d, u, v)$, $\neg P(a, x, y) \vee \neg P(x, x, b)$

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 $R = C + D - \{\text{matched literals}\}$.

3avii

Refutation by Resolution

3aviii

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 unsatisfiable, or contradictory)
2. The derivation is called a **refutation**.
3. The empty clause is derived by resolving two unit clauses of opposite sign
For example, $P(x, a)$ and $\neg P(b, y)$
i.e. $P(x, a)$ is true for every x and $P(b, y)$ is false for every y,
for instance $P(b, a)$ is true and $P(b, a)$ is false - a contradiction
4. In slides 4 we'll define unsatisfiability for sentences in first order languages

General Resolvent = many ground resolvents

3aix

A **ground term** is a term with no variables from the language of the data.

A **ground instance of a clause** is obtained by substituting ground terms for its variables.

Unification allows a resolution step to capture several ground resolution steps at once.

E.g. $P(x, y) \vee Q(y)$ and $\neg P(v, f(v))$ resolve to give $Q(f(x))$
(match $v=x$ and $y=f(v)=f(x)$)

captures $P(b, f(b)) \vee Q(f(b))$ and $\neg P(b, f(b))$ resolve to give $Q(f(b))$
(substitute $x=b$, $y=f(b)$)

and $P(a, f(a)) \vee Q(f(a))$ and $\neg P(a, f(a))$ resolve to give $Q(f(a))$
(substitute $x=a$, $y=f(a)$)

and $P(f(a), f(f(a))) \vee Q(f(f(a)))$ and $\neg P(f(a), f(f(a)))$ resolve to give $Q(f(f(a)))$
(substitute $x=f(a)$, $y=f(f(a))$)

etc.

This is what makes general resolution a very powerful deduction rule

Logical Basis of Resolution

3ax

What should we do with (1) $P(x, f(x)) \vee Q(x)$ and (2) $\neg P(f(x), y)$?

$$(1) \equiv \forall x [P(x, f(x)) \vee Q(x)]$$

$$(2) \equiv \forall x \forall y [\neg P(f(x), y) \equiv \forall z \forall v [\neg P(f(z), v)]]$$

Resolving $P(x, f(x))$ and $\neg P(f(z), v)$

$$x = f(z), v = f(x) = f(f(z))$$

giving $P(f(z), f(f(z))) \vee Q(f(z))$ and $\neg P(f(z), f(f(z)))$

$$\text{and resolvent } Q(f(z)) \equiv \forall z [Q(f(z))]$$

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

Questions:

1) If clauses are not standardized apart, what happens?

Hint: Consider the above two clauses

2) Where does standardizing apart occur in Prolog?

Constructing Resolution Proofs:

3bi

Now that you know what resolution is, you might ask “how is a resolution proof constructed?” In fact, the **Completeness Property** of resolution says that for a set of unsatisfiable clauses a refutation 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 does exist.

e.g. if $S = \{P(f(x) \vee \neg P(x), P(a), \neg P(a)\}$, then the sequence of resolvents $P(f(a))$, $P(f(f(a)))$,... formed by continually resolving with the first clause won’t lead to \square , even though resolving clauses $P(a)$ and $\neg P(a)$ 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 other clauses, either the given clauses, or those derived as earlier 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, S_0 say, is the given clauses (for which a refutation is sought). The second group, S_1 , is the set of all resolvents that can be derived using clauses from S_0 . In general,

$$S_0 = \{C: C \text{ is a given clause}\}$$

$$S_i (i > 0) = \{R: R \text{ is a resolvent formed from clauses in } S_j, j < i, \text{ and which uses at least one clause from } S_{i-1}\}.$$

Continue until some S_j 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 S_0 are formed giving S_1 , then all clauses that can be formed from S_0 and S_1 together are formed giving S_2 , etc.

Saturation refinement:

- 1) state S_0 = given clauses S ;
- 2) to generate state S_i ($i \geq 1$) :
generate all resolvents involving at least one clause from S_{i-1} ;
- 3) increment i and repeat step 2 until a state contains [], or is empty.

Other possibilities (considered later) include:

Generate resolvents using the previous 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 (see ppt)

3biv

State S_0 (given clauses)

- 1 $Dca \vee Dcb$ 2 $\neg Dxy \vee Cxy$ 3 $\neg Tx \vee \neg Cxb$ 4 Tc 5. $\neg Dcz$

State S_1 (resolvents formed from given clauses)

- 6 (1,2) $Cca \vee Dcb$ 7 (1,2) $Ccb \vee Dca$
8 (1,5) Dcb 9 (2,3) $\neg Dxb \vee \neg Tx$
10 (3,4) $\neg Ccb$ 11 (1,5) Dca

State S_2 (resolvents formed from clauses in S_1 with clauses in S_0 or S_1)

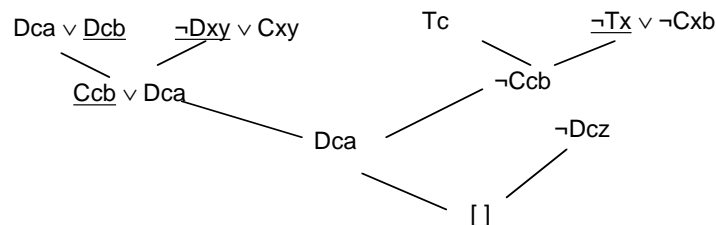
- 12 (8,2) Ccb 13 (8,9) $\neg Tc$ 14 (8,5), (11,5) []
15 (9,4) $\neg Dcb$ 16 (10,2) $\neg Dcb$ 17. (11,2) Cca

There are some more possible resolvents in State S_2 . Which are they?

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

We can also present a particular 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.

$$\text{eg } p \vee \neg q \vee \neg q \vee p \equiv p \vee \neg q$$

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

e.g. $P(a,x) \vee P(y,b)$ factors to give $P(a,b)$, but the two clauses are NOT equivalent

On the other hand factoring is sometimes necessary

eg given $\neg P(a) \vee \neg P(v)$ and $P(x) \vee P(y)$

What resolvents can you form? (Remember to rename variables apart)

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

FACTORING (see ppt)

3cii

Given a clause $C = E_1 \vee E_2 \vee \dots \vee E_n \vee H$, where E_i are literals of the same sign, F is a **factor** of C if $F = (E \vee H)\theta$, where $\theta = \text{mgu}\{E_i\}$ and $E\theta = E_i\theta$ (for every i)

Examples - what are the bindings applied to give the factor?:

$P(x,a) \vee P(b,y)$ 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. $P(x) \vee \neg P(f(x))$? Or indeed of $P(x) \vee P(f(x))$?

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.

(More in slides 6)....

Conversion to Clausal Form

3di

Resolution is a good method **for clauses**.

• What if the given data is not a set of clauses?

Suppose you are given some Data and a conclusion in standard predicate logic?

• We know to show $\text{Data} \models \text{Conclusion}$, we can instead derive a contradiction from $\text{Data} + \neg \text{Conclusion}$.

• So we need somehow to convert $\text{Data} + \neg \text{Conclusion}$ to clauses.

Here's how

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 \neg inwards).

2. **Rename quantified variables** to be distinct.

3. **Skolemise** - (See 3dii)

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: $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (A \vee C)$, etc.

6. **Re-distribute universal quantifiers** across \wedge .

Conversion to Clausal Form

3dii

Conversion to clauses uses 6 basic steps.....

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** .

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

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

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

But what about a sentence such as $\forall x \exists y.\text{lives}(x,y)$?

Why would $\forall x.\text{lives}(x,h)$, where "**h**" is a new constant, be **WRONG**?

More on Skolemisation

3diii

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

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

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

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

So we get $\text{lives}(kb, \text{plc}(kb))$, $\text{lives}(ar, \text{plc}(ar))$, $\text{lives}(pp, \text{plc}(pp))$, etc.

All the conversion steps except Step 3 (Skolemisation) maintain equivalence, so we don't have $S \equiv \text{converted}(S)$. In fact, it is sufficient to know that

converted (S) are contradictory if and only if (iff) S are contradictory.

And this property is true. (See later.)

PRACTICE IN CONVERSION TO CLAUSAL FORM

(See ppt)

3div

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 $\exists x S$; in step 3 on 3dii the Skolem function is dependent only on universal variables in whose scope $\exists x S$ lies and which occur in S . 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 $\forall x [P(x) \vee Q(f(x))]$. This is the result if "and which occur in S " is omitted, which it often is.

More SKOLEMISATION Examples

3dv

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

$\exists z [\neg(P(z) \rightarrow R(z))] \vee Q(a) \Rightarrow \exists z [P(z) \wedge \neg R(z)] \vee Q(a)$
(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:

3ei

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.