AUTOMATED REASONING

NOTE: THERE ARE NO SLIDES 2!

SLIDES 3:

RESOLUTION
- **The resolution rule for clauses**
- **Unification**
- **Refutation by resolution**
- **Factoring**

Resolution rule for propositional clauses

Recall the structure of a Clause

A clause has the form A1 ∨ A2 ∨... ∨An, where each Ai is a literal

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

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

A, ¬A ∨ B ==> B

¬B, ¬A ∨ B ==> ¬A

¬A ∨ B, ¬B ∨ C ==> ¬A ∨ C

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

A, A → B ==> B

¬B, A → B ==> ¬A

A → B, B →C ==> A →C

A ∧ D → B, B→ C ∨ E
==> A ∧ D → C∨ E

A, ¬A ==>[] ([] is the empty clause and is equivalent to False)

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 |- False by resolution, then S |= False (called Soundness)

But S |= False means S has no models and is unsatisfiable

So resolution is a way to show unsatisfiability of S

First order Clauses and Resolution

First order Clauses

A clause has the form A1 ∨ A2 ∨... ∨An, where each Ai is a literal

An atom has the form predicate(argument1, arg2,.., argn)

Clauses may be ground (no variables)

Clauses may have variables which are all implicitly universally quantified.

e.g. less(x,s(x)) ∨ ¬pos(x) ≡ ∀x(less(x,s(x)) ∨ ¬pos(x))
(also ≡ ∀x(pos(x)→less(x,s(x))))

Generally, the statement ∀x(P(x)) is read as "for every x P(x) is true",
or "for each instance of the clause the sentence is true".

e.g. P(x,s(x)) ∨ ¬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)) ∨ ¬R(a), P(s(a),s(s(a))) ∨ ¬R(s(a)), P(b,s(b)) ∨ ¬R(b), etc.
(Assume the language contains constants a, b and function symbol s)

**NOTATION CONVENTION: Variables will start u, v, w, x, y, z;
other terms are constants or functional terms. e.g. a,b, f(...), etc.**

Resolution:

Slides 3 introduce 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.

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 defined on slide 3av is called *Binary Resolution*. In order to form a resolvent, (the clause obtained by resolving two clauses) 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.

Robinson actually proposed a more flexible version of the rule, 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 by resolving ¬G and E. 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 the Optional material on slides 3dii/3diii you can see how to achieve clausal form from arbitrary first order sentences using a process called Skolemisation. We'll look at Skolemisation later in relation to Tableaux. Until then, for the exercises in this course you will be given data that is already in clausal form.

Binary Resolution:

The structure of the Resolution Rule was ¬A ∨ B, ¬B ∨ C ==> ¬A ∨ C

But consider P(x,s(x)) ∨ ¬R(x) and R(f(a)) ∨ ¬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.
Can 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?

The resolvent is P(f(a),s(f(a))) ∨ ¬Q(y)
That is, substitute f(a) for x everywhere it occurs as we have x==f(a)

(NOTE: the resolvent won't be P(x,s(x)) ∨ ¬Q(y), as we required x==f(a))

This gives us first order resolution!

Binary Resolution:

Given clauses C1 = ¬G ∨ H and C2 = E ∨ F,
where E and G are atoms and H and F are clauses of none or more literals.

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

Example:
(1) P(x, f(x)) ∨ ¬ R(x) (Use u-z for variables)
(2) ¬P(a, y) ∨ S(g(y)) (Use a...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 ¬R(a) ∨ P(a, f(a)) (use x==a)
(iii) *Instantiate* (2) giving ¬P(a, f(a)) ∨ S(g(f(a))) (use y==f(a))
(iv) *Derive* ¬R(a) ∨ S(g(f(a))) (by resolution)

(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

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

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

2) Select an equation and apply the action for the appropriate Case from below.

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

for cases d) and e) there is no action as these are failure cases
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(g(x)) – called *occurs check*)

Repeat 2) until there are no equations left (*success*) or d) or e) applies (*failure*).

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) ∨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}.

Some Miscellaneous Properties of Unifiers

A ***substitution*** λ in a language L is a set of equations {xi == ti} such that each xi is unique, xi ≠ ti and xi does not occur in ti. (xi == ti is sometimes written as xi/ti (xi is replaced by ti), or ti/xi (ti replaces xi).

A substitution λ can be applied to P, where P may be a clause, literal or term; the application is written as Pλ and means that the substitutions indicated by λ are made to variables in P.

Usually λ will be *idempotent* (λ is fully evaluated); i.e. no xi occurs in any tj. Then (Xλ)λ = Xλ for any X.

If Pλ = Qλ and P and Q are both literals or both terms, then λ is a *unifier* of P and Q. Pλ is called a *ground instance* of P if it has no variables.

The unification algorithm for X,Y produces a *most general unifier* (mgu) of X,Y. A mgu θ of X and Y is a unifier of X and Y, such that, for any other unifier λ of X and Y, ∃σ (Xθ) σ = X λ = Y λ. i.e. you can find σ to apply to Xθ that yields Xλ.

Substitutions σ and θ can be *composed:* X(σ λ) is defined as (Xσ) λ.
If σ = {xi == ti} and λ = {yi == si}, then σλ = {xi == tiλ, yi == si}, where xi ≠ tiλ, xi does not occur in tiλ, and yi ≠ any xj. i.e. only those yi ≠ any xj are retained.)

e.g. θ = {x == f(y), z ==f (y)} unifies P(z,z) and P(x,f(y))
λ1 = {z == f(y), x == z} does not unify P(z,z) and P(x,f(y)) and is not idempotent; another unifer is λ = {x==f(a), z==f(a), y==a} and λ = θ {y == a}

General Resolvent = many ground resolvents

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

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

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

E.g. P(x,y) ∨ Q(y) and ¬P(v,f(v)) resolve to give Q(f(x))
(unify v==x and y==f(v)==f(x))

captures P(b,f(b)) ∨ Q(f(b)) and ¬P(b,f(b)) resolve to give Q(f(b))
(substitute x==v==b, y==f(b))

and P(a,f(a)) ∨ Q(f(a)) and ¬P(a,f(a)) resolve to give Q(f(a))
(substitute x==v==a, y==f(a))

and P(f(a),f(f(a))) ∨ Q(f(f(a))) and ¬P(f(a),f(f(a))) resolve to give Q(f(f(a)))
(substitute x==v==f(a), y==f(f(a)))

etc.

This is what makes general resolution such a powerful deduction rule

Logical Basis of Resolution

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(f(x),y) ≡ ∀z∀v[¬P(f(z),v)]

Resolving P(x,f(x)) and ¬P(f(z), v).....
unifier is: x==f(z), v==f(x)==f(f(z))

instances are: P(f(z), f(f(z))) ∨Q(f(z)) and ¬P(f(z),f(f(z)))
resolvent is: Q(f(z)) ≡ ∀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:

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 always exists. (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) ∨ ¬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 P(a) and ¬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.

Refutation by Resolution

3bii

1. The aim of a resolution proof is to use resolution to derive from given clauses C the empty clause **[ ]**, which represents False

2. Such a 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 ¬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 and show that if C derives **[ ]** then C is unsatisfiable.

Next we look at a simple and fair strategy for finding refutations.

Saturation Search:

3biii

The method outlined on Slide 3biv is called *saturation search*. See Slide 3bv 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}.
Stop if 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

3biv

How is a resolution proof made?

The simplest strategy is called a SATURATION refinement.

Saturation refinement:

1) state S0 = given clauses S
2) state S1 = {resolvents using two clauses from S0}
3) state S2 = {resolvents using clauses from S1∪S0 : at least one is from S1}
4) Continue to form states S3 from S2∪S1∪S0, etc. such that each resolvent in Sj is formed using at least one clause from Sj-1}
5) Stop if a state contains [], or if a state is empty

Other possibilities for controlling generation of resolvents include:

Always use 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 (considered later),
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)

3bv

State S0 (given clauses)

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

State S1 (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

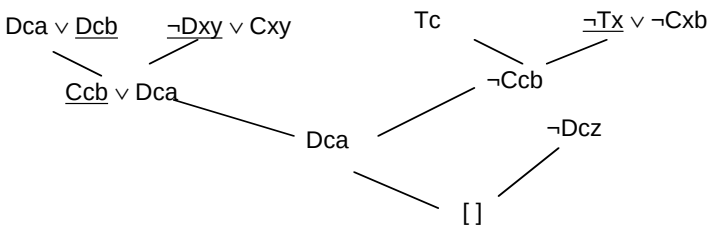
State S2 (resolvents formed from clauses in S1 with clauses in S0 or S1)

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 particular resolution refutation as a tree: 3bvi



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

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

• The analogous and more general operation is called *Factoring*

Given a clause C = E1 ∨ E2∨ ... ∨ En ∨H, where Ei are literals of the same sign,
F is a **factor** of C if F=(E ∨H)θ , where θ=mgu{Ei} and Eθ= Eiθ (for every i)

FACTORING (see ppt) 3cii

Given a clause C = E1 ∨ E2∨ ... ∨ En ∨H, where Ei are literals of the same sign,
F is a **factor** of C if F=(E ∨H)θ , where θ=mgu{Ei} and Eθ= Eiθ (for every i)

Examples - what are the bindings applied to give the factor?:
P(x,a) ∨ P(b,y) factors to P(b,a)
P(x) ∨ P(a) factors to P(a)
Q(a,b) ∨ Q(a,b) factors to Q(a,b) (factoring identical literals is called merging)
P(x,x) ∨ P(a,y) factors to P(a,a)
P(x,y) ∨ P(x,x) ∨ P(y,z) factors to P(x,x) ∨ P(x,z) and also to P(x,x)

FACTORING PRACTICE

Find factors of 1. P(u,u,c) ∨ P(d,u,v)
2. P(x,y) ∨ P(z,x)
3. P(x,y) ∨ ¬P(x,x) ∨ P(y,z)

Why are there no factors of 4. P(x) ∨ ¬P(f(x))? Or indeed of P(x) ∨ 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)....

Factoring Continued 3ciii

• Unlike merge, factoring does not always preserve equivalence

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

Factoring is sometimes necessary:

eg given ¬P(a) ∨ ¬P(v) and P(x) ∨ P(y)

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

• Logically we can derive the empty clause from the given clauses:
¬P(a) ∨ ¬P(v) means ∀v[¬P(a) ∨ ¬P(v)] from which we can derive ¬P(a), and
P(x) ∨ P(y) means ∀x∀y[P(x) ∨ P(y)] from which we can derive ∀z.P(z)
We *factor* by applying a binding to enable literals to be merged.

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

4. 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).

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

6. Resolution derivations can be depicted as a tree.

START of OPTIONAL MATERIAL (SLIDES 3)

Conversion to Clausal Form Skolemisation

Conversion to Clausal Form

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 Data |= Conclusion, we can instead derive a contradiction from Data + ¬ Conclusion.

• So we need somehow to convert Data + ¬ Conclusion to clauses.

Here's how .....

Conversion to clauses uses 6 basic steps:

1. *Eliminate →:* $A \rightarrow B \Rightarrow \neg A \vee B$, $A \leftrightarrow B \Rightarrow (A \rightarrow B) \wedge (B \rightarrow A)$.
   $\neg (A \wedge B) \Rightarrow \neg A \vee \neg B$ (and similar rewrites to push ¬ inwards).
2. *Rename quantified variables* to be distinct.
3. *Skolemise* - (See 3dii)
4. *Move universal quantifiers into a prefix*:
   A op $\forall x$ P[x] $\Rightarrow$ $\forall x$[A 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 $\wedge$.

Conversion to Clausal Form

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 $\exists x \exists y$[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).

But what about a sentence such as $\forall x \exists y$.lives(x,y)?
Why would $\forall x$.lives(x,h), where "h" is a new constant, be WRONG?
Answer: it says everyone lives at "h"!

More on Skolemisation

3diii

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

For instance: given ∀x∃y.lives(x,y) (meaning everyone lives in some place), we may have ∃y.lives(kb, y), ∃y.lives(ar, y), ∃y.lives(pp, y), etc.

Skolemising each of ∃y.lives(kb, y), ∃y.lives(ar, y), ∃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 ∀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 ≡ 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 Appendix 1.)

PRACTICE IN CONVERSION TO CLAUSAL FORM

(See ppt) 3div

Convert to clausal form:

1. ∀x [∃yS(x,y) ↔ ¬ P(x)] done below
2. ∀z [P(z) → R(z)] → Q(a)
3. ∀x [P(x) ∨ R(x) → ∃y∀w [Q(y,w,x)]]

∀x [∃yS(x,y) ↔ ¬ P(x)]

(convert ↔) ∀x [(∃yS(x,y) → ¬ P(x)) ∧ (¬ P(x) → ∃yS(x,y))]

(convert →) ∀x [(¬∃yS(x,y) ∨¬ P(x)) ∧ (¬ ¬P(x) ∨ ∃yS(x,y))]

(move ¬) ∀x [(∀y¬S(x,y) ∨¬ P(x))∧ (P(x) ∨ ∃yS(x,y))]

(rename quantifiers) ∀x [(∀z¬S(x,z) ∨¬ P(x)) ∧ (P(x) ∨ ∃yS(x,y))]

(Skolemise ∃yS(x,y)) ∀x [(∀z¬S(x,z) ∨¬ P(x)) ∧ (P(x) ∨ S(x,f(x)))]

(Pull out quantifiers) ∀x∀z[(¬S(x,z) ∨¬ P(x)) ∧ (P(x) ∨ S(x,f(x)))]

(Redistribute ∀x ∀z) ∀x∀z[¬S(x,z) ∨¬ P(x)] ∧ ∀x[P(x) ∨ S(x,f(x))]

NOTE: there are many ways to Skolemise ∃x S; in step 3 on 3dii the Skolem function is dependent only on universal variables in whose scope ∃x S lies and which occur in S. eg ∀x [P(x) ∨ ∃y Q(y)] Skolemises to ∀x [P(x) ∨ Q(a)] with the rules here, as x doesn't occur in ∃y Q(y), not to ∀x [P(x) ∨Q(f(x))]. This is the result if "and which occur in S" is omitted, which it often is.

More SKOLEMISATION Examples

3dv

∀z [P(z) → R(z)] → Q(a) ⇒ ¬(∀z[P(z) → R(z)])∨Q(a) ⇒

∃z[¬(P(z) → R(z))]∨Q(a) ⇒ ∃z[P(z) ∧ ¬R(z)]∨Q(a)
(all by step 1) (no need for step 2, 1 bound variable)

⇒(P(c) ∧ ¬R(c))∨Q(a) (by step 3, c is a new constant) (no need for step 4)

⇒(P(c)∨Q(a)) ∧ (¬R(c) ∨Q(a)) (by step 5) (no need for step 6)

∀x[P(x) ∨ R(x) → ∃y∀w [Q(y,w,x)]] ⇒ ∀x[¬(P(x) ∨ R(x)) ∨ ∃y∀w [Q(y,w,x)]]

⇒ ∀x[(¬P(x) ∧ ¬R(x)) ∨ ∃y∀w [Q(y,w,x)]]
(by step 1) (no need for step 2, all bound variables different)

⇒ ∀x[(¬P(x) ∧ ¬R(x)) ∨ ∀w[Q(f(x),w,x)]]
(by step 3, f is new functor, y replaced by f(x) as y in scope of x)

⇒∀x∀w[(¬P(x) ∧ ¬R(x)) ∨ Q(f(x),w,x)] (step 4)

⇒∀x∀w[(¬P(x)∨ Q(f(x),w,x)) ∧ (¬R(x)) ∨ Q(f(x),w,x))] (step 5)

⇒∀x∀w[¬P(x)∨ Q(f(x),w,x)] ∧ ∀x∀w[¬R(x)) ∨ Q(f(x),w,x)] (step 6)