Lecture 8
The Resolution Method

A minor miracle occurred in 1965 when Alan Robinson published his **resolution method**. This method uses a generalised version of the resolution rule of inference we saw in the previous lecture. It has been mathematically proven to be **refutation-complete** over first order logic. This means that if you write any set of sentences in first order logic which are **unsatisfiable** (i.e., taken together they are false, in that they have no models), then the resolution method will eventually derive the False symbol, indicating that the sentences somehow contradict each other.

In particular, if the set of first order sentences comprises a set of axioms and the negation of a theorem you want to prove, the resolution method can be used in a proof-by-contradiction approach. This means that, if your first order theorem is true then proof by contradiction using the resolution method is guaranteed to find the proof to a theorem eventually. The underlining here identifies some drawbacks to resolution theorem proving:

- It only works for true theorems which can be expressed in first order logic: it cannot check at the same time whether a conjecture is true or false, and it can't work in higher order logics. (There are related techniques which address these problems, to varying degrees of success.)
- While it is proven that the method will find the solution, in practice the search space is often too large to find one in a reasonable amount of time, even for fairly simple theorems.

Notwithstanding these drawbacks, resolution theorem proving is a complete method: if your theorem does follow from the axioms of a domain, then resolution can prove it. Moreover, it only uses one rule of deduction (resolution), rather than the multitude we saw in the last lecture. Hence, it is comparatively easy to understand how resolution theorem provers work. For these reasons, the development of the resolution method was a major accomplishment in logic, with serious implications to Artificial Intelligence research.

Resolution works by taking two sentences and resolving them into one, eventually resolving two sentences to produce the False statement. The resolution rule is more complicated than the rules of inference we've seen before, and we need to cover some preparatory notions before we can understand how it works. In particular, we need to look at conjunctive normal form and unification before we can state the full resolution rule at the heart of the resolution method.

8.1 Binary Resolution

We saw unit resolution (a propositional inference rule) in the previous lecture:

$$\frac{A \vee B, \quad \neg B}{A}$$

We can take this a little further to **propositional binary resolution**:

$$\frac{A \vee B, \quad \neg B \vee C}{A \vee C}$$

Binary resolution gets its name from the fact that each sentence is a disjunction of exactly two literals. We say the two opposing literals B and ¬B are **resolved** — they are removed when the disjunctions are merged.

The binary resolution rule can be seen to be sound because if both A and C were false then then at least one of the sentences on the top line would be false. As this is an inference rule, we are assuming that the top line is true. Hence we can't have both A and C being false, which means either A or C must be true. So we can infer the bottom line.

Kowalski Form

It might be easier to grasp binary resolution if we use the 'replace implication' equivalence rule from the last lecture to write top and bottom using implications. Writing ¬A as D we get:

$$\frac{D \rightarrow B, \quad B \rightarrow C}{D \rightarrow C}$$

This is the **Kowalski form** (or implicative form) of the binary resolution rule, after the department's own Professor Kowalski. It makes it easier to see that resolution is a sound rule of inference.

So far we've only looked at propositional versions of resolution. In first-order logic we need to also deal with variables and quantifiers. As we'll see below, we don't need to worry about quantifiers: we are going to be working with sentences that only contain free variables. Recall that we treat these variables as implicitly universally quantified, and that they can take any value. This allows us to state a more general **first-order binary resolution** inference rule:

$$\frac{A \vee B, \quad \neg C \vee D}{\text{Subst}(\theta, A \vee D)} \quad \text{Subst}(\theta, B) = \text{Subst}(\&\text{theta}, C)$$

This rule has the side condition Subst(θ, B) = Subst(&theta, C), which requires there to be a substitution θ which makes B and C the same before we can apply the rule. (Note θ can substitute fresh variables whie making B and C equal. It doesn't have to be a ground substitution!) If we can find such a θ, then we can make the resolution step and apply θ to the outcome. In fact, the first-order binary rule is simply equivalent to applying the substitution to the original sentences, and then applying the propositional binary rule.

8.2 Conjunctive Normal Form

For the resolution rule to resolve two sentences, they must both be in a normalised format known as **conjunctive normal form**, which is usually abbreviated to CNF. This is an unfortunate name because the

sentences themselves are made up of sets of disjunctions. It is implicitly assumed that the entire knowledge base is a big conjunction of the sentences, which is where conjunctive normal form gets its name. So, CNF is actually a conjunction of disjunctions. The disjunctions are made up of literals which can either be a predicate or the negation of a predicate (for propositional read a proposition or the negation of a proposition):

So, CNF sentences are of the form:

$$(p_1 \vee p_2 \vee ... \vee p_m) \quad \wedge \quad (q_1 \vee q_2 \vee ... \vee q_n) \quad \wedge \quad \text{etc.}$$

where each p_i and q_j is a literal. Note that we call the disjunction of such literals a **clause**. As a concrete example,

$$\text{likes(george, X)} \vee \text{likes(tony, george)} \vee \neg \text{is\_mad(maggie)}$$

is in conjunctive normal form, but:

$$\text{likes(george, X)} \vee \text{likes(tony, george)} \longrightarrow \neg \text{is\_mad(maggie)} \wedge \text{is\_mad(tony)}$$

is not in CNF.

The following eight-stage process converts any sentence into CNF:

1. **Eliminate arrow connectives** by rewriting with
$$P \longleftrightarrow Q \quad => \quad (P \longrightarrow Q) \wedge (Q \longrightarrow P)$$
$$P \longrightarrow Q \quad => \quad \neg P \vee Q$$

2. **Move ¬ inwards** using De Morgan's laws (inc. quantifier versions) and double negation:
$$\neg (P \vee Q) \quad => \quad \neg P \wedge Q$$
$$\neg (P \wedge Q) \quad => \quad \neg P \vee Q$$
$$\neg \forall X.\ P \quad => \quad \exists X.\ \neg P$$
$$\neg \exists X.\ P \quad => \quad \forall X.\ \neg P$$
$$\neg\neg P \quad => \quad P$$

3. **Rename variables apart**: the same variable name may be reused several times for different variables, within one sentence or between several. To avoid confusion later rename each distinct variable with a unique name.

4. **Move quantifiers outwards**: the sentence is now in a form where all the quantifiers can be moved safely to the outside without affecting the semantics, provided they are kept in the same order.

5. **Skolemise existential variables** by replacing them with **Skolem constants and functions**. This is similar to the existential elimination rule from the last lecture: we just substitute a term for each existential variable that represents the 'something' for which it holds. If there are no preceeding universal quantifiers the 'something' is just a fresh constant. However, if there are then we use a function that takes all these preceeding universal variables as arguments. When we're done we just drop all the universal quantifiers. This leaves a quantifier-free sentence. For example:

   $\forall X.\ \exists Y.\ \text{person(X)} \longrightarrow \text{has(X, Y)} \wedge \text{heart(Y)}$

is Skolemised as

$person(X) \longrightarrow has(X, f(X)) \wedge heart(f(X))$

6. **Distribute $\wedge$ over $\vee$** to make a conjunction of disjunctions. This involves rewriting with:

$$P \wedge (Q \vee R) \Rightarrow (P \wedge Q) \vee (P \wedge R)$$
$$(P \vee Q) \wedge R \Rightarrow (P \wedge R) \vee (Q \wedge R)$$

7. **Flatten binary connectives**: replace nested $\wedge$ and $\vee$ with flat lists of conjuncts and disjuncts:

$$P \wedge (Q \wedge R) \Rightarrow P \wedge Q \wedge R$$
$$(P \wedge Q) \wedge R \Rightarrow P \wedge Q \wedge R$$
$$P \vee (Q \vee R) \Rightarrow P \vee Q \vee R$$
$$(P \vee Q) \vee R \Rightarrow P \vee Q \vee R$$

The sentence is now in CNF. Further simplication can take place by removing duplicate literals and dropping any clause which contains both A and $\neg$A (one will be true, so the clause is always true. In the conjunction of clauses we want everything to be true, so we can drop it.) There is an optional final step that takes it to **Kowalski normal form**, also known as **implicative normal form (INF)**:

8. **Reintroduce implication** by gathering up all the negative literals (the negated ones) and forming their conjunction N, then taking the disjunction P of the positive literals, and forming the logically equivalent clause $N \longrightarrow P$.

Example: Converting to CNF

We will work through a simple propositional example:

$(B \vee (A \wedge C)) \longrightarrow (B \vee \neg A)$

This first thing to do is remove the implication sign:

$\neg (B \vee (A \wedge C)) \vee (B \vee \neg A)$

Next we use De Morgan's laws to move our negation sign from the outside to the inside of brackets:

$(\neg B \wedge \neg (A \wedge C)) \vee (B \vee \neg A)$

And we can use De Morgan's law again to move a negation sign inwards:

$(\neg B \wedge (\neg A \vee \neg C)) \vee (B \vee \neg A)$

Next we distribute $\vee$ over $\wedge$ as follows:

$(\neg B \vee (B \vee \neg A)) \wedge ((\neg A \vee \neg C) \vee (B \vee \neg A))$

If we flatten our disjunctions, we get our sentence into CNF form. Note the conjunction of disjunctions:

$(\neg B \vee B \vee \neg A) \wedge (\neg A \vee \neg C \vee B \vee \neg A)$

Finally, the first conjunction has $\neg$B and B, so the whole conjunction must be true. Also, we can remove the

duplicate ¬A in the second conjunction:

True $\wedge$ ($\neg$ A $\vee$ $\neg$ C $\vee$ B)

The truth of this sentence is only dependent on the second conjunct. If it is false, the whole thing is false, if it is true, the whole thing is true. Hence, we can remove the True, giving us a single clause in its final conjunctive normal form:

$\neg$ A $\vee$ $\neg$ C $\vee$ B

If we want Kowalski normal form we take one more step to get:

(A $\wedge$ C) $\longrightarrow$ B

8.3 Unification

We have said that the rules of inference for propositional logic detailed in the last lecture can also be used in first-order logic. However, we need to clarify that a little. One important difference between propositional and first-order logic is that the latter has predicates with terms as arguments. So, one clarification we need to make is that we can apply the inference rules as long as the predicates and arguments match up. So, not only do we have to check for the correct kinds of sentence before we can carry out a rule of inference, we also have to check that the arguments do not forbid the inference.

For instance, suppose in our knowledge base, we have the these two statements:

knows(john,X) $\longrightarrow$ hates(john, X)
knows(john, mary)

and we want to use the Modus Ponens rule to infer something new. In this case, there is no problem, and we can infer that, because john hates everyone he knows, and he knows Mary, then he must hate Mary, i.e., we can infer that hates(john, mary) is true.

However, suppose instead that we had these two sentences:

knows(john,X) $\longrightarrow$ hates(john, X)
knows(jack, mary)

Here, the predicate names have not changed, but the arguments are holding us back from making any deductive inference. In the first case above, we could allow the variable X to be instantiated to mary during the deduction, and the constant john before and after the deduction also matched without problem. However, in the second case, although we could still instantiate X to mary, we could no longer match john and jack, because they are two different constants. So we cannot deduce anything about john (or anybody else) from the latter two statements.

The problem here comes from our inability to make the arguments in knows(john, X) and the arguments in knows(jack, mary) match up. When we can make two predicates match up, we say that we have **unified** them, and we will look at an algorithm for unifying two predicates (if they can be unified) in this section. Remember that unification plays a part in the way Prolog searches for matches to queries.

- **A Unification Algorithm**

To unify two sentences, we must find a substitution which makes the two sentences the same. Remember that we write V/T to signify that we have substituted term T for variable V (read the "/" sign as "is substituted by"). The purpose of this algorithm will be to produce a substitution (a set of pairs V/T) for a given pair of sentences. So, for example, the output for the pair of sentences:

knows(john,X)
knows(john, mary)

will be: {X/mary}. However, for the two sentences above involving jack, the function should fail, as there was no way to unify the sentences.

To describe the algorithm, we need to specify some functions it calls internally.

- The function `isa_variable(x)` checks whether `x` is a variable.
- The function `isa_compound(x)` checks whether `x` is a **compound** expression: either a predicate, a function or a connective which contains subparts. The subparts of a predicate or function are the arguments. The subparts of a connective are the things it connects. We write `args(x)` for the subparts of compound expression `x`. Note that `args(x)` outputs a list: the list of subparts. Also, we write `op(x)` to signify the symbol of the compound operator (predicate name, function name or connective symbol).
- The function `isa_list(x)` checks whether `x` is a list. We write `head(L)` for the first term in a list `L` and `tail(L)` for the sublist comprising all the other terms except the head. Hence the head of [2,3,5,7,11] is 2 and the tail is [3,5,7,11]. This terminology is common in Prolog.

It's easiest to explain the unification algorithm as a recursive method which is able to call itself. As this is happening, a set, mu, is passed around the various parts of the algorithm, collecting substitutions as it goes. The method has two main parts:

`unify_internal(x,y,mu)`
which returns a substitution which makes sentence `x` look exactly like sentence `y`, *given an already existing set of substitutions `mu` (although `mu` may be empty).* This function checks various properties of `x` and `y` and calls either itself again or the `unify_variable` routine, as described below. Note that the order of the if-statements is important, and if a failure is reported at any stage, the whole function fails. If none of the cases is true for the input, then the algorithm fails to find a unifying set of substitutions.

`unify_variable(var,x,mu)`
which returns a substitution given a variable `var`, a sentence `x` and an already existing set of substitutions `mu`. This function also contains a set of cases which cause other routines to run if the case is true of the input. Again, the order of the cases is important. Here, if none of the cases is true of the input, a substitution is returned.

The algorithm is as follows:

```
 unify(x,y) = unify_internal(x,y,{})

unify_internal(x,y,mu) ----------------------

Cases

1. if (mu=failure) then return failure
```

```
2. if (x=y) then return mu.

3. if (isa_variable(x)) then return unify_variable(x,y,mu)

4. if (isa_variable(y)) then return unify_variable(y,x,mu)

5. if (isa_compound(x) and isa_compound(y)) then return
unify_internal(args(x),args(y),unify_internal(op(x),op(y),mu))

6. if (isa_list(x) and isa_list(y)) then return
unify_internal(tail(x),tail(y),unify_internal(head(x),head(y),mu))

7. return failure

unify_variable(var,x,mu) ------------------------

Cases

1. if (a substitution var/val is in mu) then return
unify_internal(val,x,mu)

2. if (a substitution x/val is in mu) then return
unify_internal(var,val,mu)

3. if (var occurs anywhere in x) then return failure

4. add var/x to mu and return
```

Some things to note about this method are:

(i) trying to match a constant to a different constant fails because they are not equal, neither is a variable and neither is a compound expression or list. Hence none of the cases in `unify_internal` is true, so it must return failure.

(ii) Case 1 and 2 in `unify_variable(var,x,my)` check that neither inputs have already been substituted. If `x` already has a substitution, then it tries to unify the substituted value and `var`, rather than `x` and `var`. It does similarly if `var` already has a substitution.

(iii) Case 3 in `unify_variable` is known as the **occurs-check** case (or occur-check). This is important: imagine we got to the stage where, to complete a unification, we needed to substitute `X` with, say, `f(X,Y)`. If we did this, we would write `f(X,Y)` instead of X. But this still has an `X` in it! So, we would need to substitute `X` by `f(X,Y)` again, giving us: `f(f(X,Y),Y)` and it is obvious why we should never have tried this substitution in the first place, because this process will never end. The occurs check makes sure this isn't going to happen before case 4 returns a substitution. The rule is: you cannot substitute a compound for a variable if that variable appears in the compound already, because you will never get rid of the variable.

(iv) The `unify_internal(op(x),op(y),mu))` part of case 5 in `unify_internal` checks that the operators of the two compound expressions are the same. This means that it will return false if, for example, it tries to unify two predicates with different names, or a $\wedge$ with a $\vee$ symbol.

(v) The unification algorithm returns the unique **most general unifier** (MGU) mu for two sentences. This means that if there is another unifier U then T.U is always an instance of T.mu. The MGU substitutes as little as it can get away with while still being a unifier.

Example: Unifying Two Sentences

Suppose we wanted to unify these two sentences:

1. p(X,tony) ∧ q(george, X, Z)
2. p(f(tony),tony) ∧ q(B,C,maggie)

We can see by inspection that a way to unify these sentences is to apply the substitution:

{X/f(tony), B/george, C/f(tony), Z/maggie}.

Therefore, our unification algorithm should find this substitution.

To run our algorithm, we set the inputs to be:

x = p(X,tony) ∧ q(george, X, Z)
and
y = p(f(tony),tony) ∧ q(B,C,maggie)

and then follow the algorithm steps.

Iteration one
`unify_internal` is called with inputs x, y and the empty list {}. This tries case 1, but as mu is not failure, this is not the case. Next it tries case 2, but this is also not the case, because x is not equal to y. Cases 3 and 4 similarly fail, because neither x nor y is a variable. Finally, case 5 kicks in because x and y are compound terms. In fact, they are both conjunctions connected by the ∧ connective. Using our definitions above, args(x)=[p(X,tony),q(george,X,Z)] and args(y)=[p(f(tony),tony),q(B,C,maggie)]. Also, op(x) = p and op(y) = p. So, case 5 means that we call unify_internal again with inputs [p(X,tony),q(george,X,Z)] and [p(f(tony),tony),q(B,C,maggie)]. Before we do that, the third input to the function will be `unify_internal(op(x),op(y),mu)`. Because our op(x) and op(y) are the same (both p), then this will return mu [check this yourselves]. mu is still the empty list, so this gets passed on.

Iteration two
So, we're back at the top of `unify_internal` again, but this time with a pair of lists as input. None of the cases match until case 6. This states that we have to split our lists into heads and tails, then unify the heads and use this to unify the tails. Unifying the heads means that we once again call `unify_internal`, this time with predicates `p(X,tony)` and `p(f(tony),tony)`.

Iteration three
Now case 5 fires again, because our two inputs are both predicates. This turns the arguments into a list, checks that the two predicate names match and calls `unify_internal` yet again, this time with lists [X,tony] and [f(tony),tony] as input.

Iteration four
In this iteration, all the algorithm does is split the lists into heads and tails, and first calls `unify_internal` with `X` and `f(tony)` as inputs, and later with `tony` and `tony` as input. In the latter case we can see that `unify_internal` will return mu, because the constant symbols are equal. Hence this will not affect anything.

Iteration five

When X and f(tony) are given as input, case 3 fires because `X` is a variable. This causes unify_variable(X,f(tony),{}) to be called. In this case, it checks that neither X nor f(tony) has been subject to a substitution already, which they haven't because the substitution list is still empty. It also makes an occurs-check, and finds that X does not appear anywhere in f(tony), so case 3 does not fire. Hence it goes all the way to case 4, and X/f(tony) is added to the substitution list. Finally, we have a substitution! This returns the substitution list {X/f(tony)} as output, and causes some other embedded calls to also return this substitution list.

It is left as an exercise to show that the same way in which the algorithm unified `p(X,tony)` and `p(f(tony),tony)` with the substitution {X/f(tony)}, it also unifies `q(george,X,Z)` and `q(B,C,maggie)`, adding B/george, C/f(tony) and Z/maggie to the substitution list. However, in this case, we had already assigned the substitution X/tony. Hence, when `unify_variable` was finally called, it fired case 2 (or is it 1?) to make sure that the already substituted variable was not given another substitution.

At this stage, all the return statements start to actually return things, and the substitution gets passed back all the way to the top. Finally, the substitution

$$\{X/f(tony), B/george, C/f(tony), Z/maggie\}.$$

is indeed produced by the unification algorithm. When applied to both sentences, the result is the same sentence:

$$p(f(tony),tony) \wedge q(george,f(tony),maggie)$$

The complexity of this relatively simple example shows why it is a good idea to get a software agent to do this, rather than doing it ourselves. Of course, if you wanted to try out the unification algorithm, you can simply run Prolog and type in your sentences separated by a single = sign. This asks Prolog to try to unify the two terms. This is what happens in Sicstus Prolog:

```
 ?- [p(X,tony),q(george,X,Z)]=[p(f(tony),tony),q(B,C,maggie)].

B = george, C = f(tony), X = f(tony), Z = maggie ?

yes.
```

We see that Prolog has come up with the same unifying substitution as before.

8.4 The Full Resolution Rule

Now that we know about unification, we can properly describe the full version of resolution:

$$\frac{p_1 \vee ... \vee p_j \vee ... \vee p_m, \quad q_1 \vee ... \vee q_k \vee ... \vee q_n}{\text{Subst}(\theta, p_1 \vee ... \vee p_{j-1} \vee p_{j+1} \vee ... \vee p_m \vee q_1 \vee ... q_{k-1} \vee q_{k+1} \vee ... \vee q_n)} \quad \text{Unify}(p_j, \neg q_k) = \theta$$

This resolves literals p_j and q_k. Note that we have to add $\neg$ to q_k to make it unify with p_j, so it is in fact p_j which is the negative literal here. The rule is more general than first-order binary resolution in that it allows an arbitrary number of literals in each clause. Moreover, θ is the most general unifier, rather than an arbitrary unifying substitution.

To use the rule in practice, we first take a pair of sentences and express them in CNF using the techniques described above. Then we find two literals, p_j and q_k for which can find a substitution mu to unify p_j and $\neg q_k$. Then we take a disjunction of all the literals (in both sentences) *except* p_j and q_k. Finally, we apply the substitution θ to the new disjunction to determine what we have just inferred using resolution.

In the next lecture, we will look at how resolution theorem proving is put into action, including some example proofs, some heuristics for improving its performance and some applications.
