Today:

- Clausal Form for First-Order Logic
- Compilation into Clauses
- Resolution Derivations
- Fault Diagnosis

Recall

A *Clause* is a set of literals (ie, possibly negated basic formulas), understood as disjoined ("or").

We use the single inference rule of *Resolution:*

$$\frac{\{\mathtt{F_1}, \ldots \mathtt{F_n}, \mathtt{L}\} \qquad \{\mathtt{G_1}, \ldots \mathtt{G_m}, \neg \mathtt{M}\}}{\{\mathtt{F_1S}, \ldots \mathtt{F_nS}, \mathtt{G_1S}, \ldots \mathtt{G_mS}\}}$$

where S is the most general unifier of L, M, i.e. there L, M can be made the same by applying some substitution S.

The empty clause $\{\ \}$ is a contradiction – it can never be true.

Proof by Refutation

We use resolution to answer a query by finding a *contradiction* (the empty clause).

Given a KB of clauses, and a query Q, look for a resolution proof of the empty clause from $KB \cup \{\ \neg Q\ \}$.

Notice that this is true exactly when Q follows from the KB:

$$KB \models Q$$

if and only if

$$KB \cup \{\ \neg Q\ \} \models \text{contradiction}$$

Example

Given clauses

$$\{\ greater(s(x), x)\ \}$$
$$\{\ greater(x, z), \neg greater(x, y), \neg greater(y, z)\ \}$$

together with the query $greater(s(s(0)), 0)$, we get a resolution proof as follows.

First name the variables apart (so there are no shared variables between clauses).

Then add the *negated* query.

1	$\{\ greater(s(x), x)\ \}$	Ax
2	$\{\ greater(w, z),$ $\neg greater(w, y), \neg greater(y, z)\ \}$	Ax
3	$\{\ \neg greater(s(s(0)), 0)\ \}$	Ax
4	$\{\ \neg greater(s(s(0)), y), \neg greater(y, 0)\ \}$	Res 3,2 $w/s(s(0)), z/0$
5	$\{\ \neg greater(s(0), 0)\ \}$	Res 4,1 $x/s(0), y/s(0)$
6	$\{\ \}$	Res 5,1

Compilation

We compile formulas in the full predicate calculus into clauses. The translation has the property that:

KB in the predicate calculus is contradictory
if and only if
the compiled KB is contradictory.

From this property, we can show that resolution is the basis for a *complete* inference strategy.

Clausal Form Algorithm

The algorithm contains 8 steps.

1. Get rid of $\rightarrow$, by replacing $A \rightarrow B$ with $\neg A \vee B$ wherever this occurs.
2 Move all negations to immediately before the predicates, using these replacements as often as possible:

$$\begin{aligned} \neg\neg A &\Rightarrow A \\ \neg(A \vee B) &\Rightarrow \neg A \wedge \neg B \\ \neg(A \wedge B) &\Rightarrow \neg A \vee \neg B \\ \neg\forall x\ F &\Rightarrow \exists x\ \neg F \\ \neg\exists x\ F &\Rightarrow \forall x\ \neg F \end{aligned}$$

3 Rename any *duplicate* bound variables, eg

$$(\forall x\ P(x)) \wedge (\exists x\ Q(x)) \quad \Rightarrow \quad (\forall x\ P(x)) \wedge (\exists y\ Q(y))$$

4 (Skolemisation)
Get rid of $\exists$ by introducing new constants and function symbols. The idea is to replace with typical instances, eg

$$\exists x\ rich(x) \quad \Rightarrow \quad rich(millionaire)$$

Skolemisation

First take the case where $\exists$ is not in the scope of any $\forall x$. Here just replace the quantified variable with a *new* constant

If the $\exists$ is beneath some universals, say $\forall y_1\, \forall y_2\, \ldots \forall y_n$, we need to introduce instead a new *function* symbol applied to the variables $y_1, y_2, \ldots, y_n$.

For example,

$$\forall y\, \exists x\, dislikes(x, y) \quad \Rightarrow \quad \forall y\, dislikes(enemy(y), y)$$

Note that the name of the new function is arbitrary, as long as it does not already appear in the KB.

5 Now drop the universal quantifiers ($\forall x$) — any variables are implicitly taken to be universal.

6 Put the propositions into *conjunctive normal form*, ie put disjunctions inside the conjunctions.

Use the replacements:

$$\begin{array}{rcl} A \vee (B \wedge C) & \Rightarrow & (A \vee B) \wedge (A \vee C) \\ (B \wedge C) \vee A & \Rightarrow & (B \vee A) \wedge (C \vee A) \end{array}$$

7 Use clause notation (and drop any duplicate literals).

$$\begin{array}{rcl} L_1 \vee \ldots \vee L_n & \Rightarrow & \{\ L_1, \ldots, L_n\ \} \\ F \wedge G & \Rightarrow & F, G \end{array}$$

8 Finally, make sure no variable appears in more than one clause, by naming apart if necessary.

Example

initial	$\forall \mathtt{x}\, (\forall \mathtt{y}\, \mathtt{P(x,y)} \rightarrow \neg(\forall \mathtt{y}\, \mathtt{Q(x,y)} \rightarrow \mathtt{R(x,y)}))$
step 1	$\forall \mathtt{x}\, \neg(\forall \mathtt{y}\, \mathtt{P(x,y)}) \vee \neg(\forall \mathtt{y}\, \neg\mathtt{Q(x,y)} \vee \mathtt{R(x,y)})$
step 2	$\forall \mathtt{x}\, (\exists \mathtt{y}\, \neg\mathtt{P(x,y)}) \vee (\exists \mathtt{y}\, \mathtt{Q(x,y)} \wedge \neg\mathtt{R(x,y)})$
step 3	$\forall \mathtt{x}\, (\exists \mathtt{y}\, \neg\mathtt{P(x,y)}) \vee (\exists \mathtt{z}\, \mathtt{Q(x,z)} \wedge \neg\mathtt{R(x,z)})$
step 4	$\forall \mathtt{x}\, \neg\mathtt{P(x,f(x))} \vee (\mathtt{Q(x,g(x))} \wedge \neg\mathtt{R(x,g(x))})$
step 5	$\neg\mathtt{P(x,f(x))} \vee (\mathtt{Q(x,g(x))} \wedge \neg\mathtt{R(x,g(x))})$
step 6	$(\neg\mathtt{P(x,f(x))} \vee \mathtt{Q(x,g(x))}) \wedge (\neg\mathtt{P(x,f(x))} \vee \neg\mathtt{R(x,g(x))})$
step 7	$\{\neg\mathtt{P(x,f(x))}, \mathtt{Q(x,g(x))}\}, \{\neg\mathtt{P(x,f(x))}, \neg\mathtt{R(x,g(x))}\}$
step 8	$\{\neg\mathtt{P(x,f(x))}, \mathtt{Q(x,g(x))}\}, \{\neg\mathtt{P(y,f(y))}, \neg\mathtt{R(y,g(y))}\}$

Note that we can get an exponential increase in the size of the KB when we do the compilation!!
But in general it does not change the size very much.

Advantages of compilation

- We can get efficient deduction from the single inference rule.
- *If* we can write directly in clauses, we can still have an intelligible KB, together with efficient inference.
- We can compile the KB just once for a whole set of queries, *if* all the queries are simple.

Problems

- The compilation process itself can be expensive, if we need to do it often (eg when KB is updated).
- The compiled KB is often unintelligible.
  Apart from the response to the query, we also want an *explanation*, which is easier to get from the predicate calculus version.
- Even when we restrict to clauses, the question of whether a given query follows is still *not* computable in general.

Getting more info

Given the KB below, want to know "Who is Jane's parent?".

$$\begin{array}{rcl} & & father(fred, jim) \\ & & father(bob, jane) \\ \forall x\, \forall y\, father(x, y) & \rightarrow & parent(x, y) \end{array}$$

Instead of adding the negated query $\neg parent(z, jane)$, we can add an *answer literal:*

$$\{\ \neg parent(z, jane), ans(z)\ \}$$

(that is, $parent(z, jane) \rightarrow ans(z)$). Now use resolution.


```
1: { father(fred,jim) }  Ax
2: { father(bob,jane) }  Ax
3: { ~father(v2,v3),  parent(v2,v3) }  Ax
4: { ~parent(v1,jane),  ans(v1) }  Ax
5: { ans(v2),  ~father(v2,jane) }  Res 4 3
                        { v3/jane , v1/v2 }
6: { ans(bob) }  Res 5 2 { v2/bob }
```

Stop when find an answer clause (rather than { }). The proof says that *bob* answers the original query.

Multiple answers

There can be several answers to a query. For example, suppose we have the extra clauses:

$$\begin{array}{rcl} & & mother(frieda, jane) \\ \forall x\, \forall y\, mother(x,y) & \rightarrow & parent(x,y) \end{array}$$

Now a resolution proof can derive two answer clauses:

$$\{\ ans(frieda)\ \} \qquad \{\ ans(bob)\ \}$$

These are both acceptable answers. There may even be a *stream* of acceptable answers.

Disjunctive answers

If the KB has disjunctive information $(A \vee B)$, we can get answers in the form

$$\{\ ans(t1), ans(t2)\ \}.$$

This says that either $t1$ or $t2$ satisfy the query (but we don't know which one).

This is *different* from the usual situation in Logic Programming, where if there is an answer, there is always a term in the language for which the result holds.

For example,

```
1: { father(bob,jane),  father(fred,jane) }  Ax
2: { ~father(v2,v3),  parent(v2,v3) }  Ax
3: { ~parent(v1,jane),  ans(v1) }  Ax
4: { ans(v2),  ~father(v2,jane) }  Res 3 2 { v3/jane, v1/v2 }
5: { ans(bob),  father(fred,jane) }  Res 4 1 { v2/bob }
6: { ans(bob),  parent(fred,jane) }  Res 5 2
                                     { v3/jane, v2/fred }
7: { ans(bob),  ans(fred) }  Res 6 3 { v1/fred }
```

Get

$$\{\ ans(a), ans(b)\ \}$$

Disjunctive solutions ctd

- This is *not* very useful; better to get a definite answer, where possible.
- If the clauses are all Horn clauses, then we can always find some number of single values for a true query.
- Prolog implements this behind the scenes, to return unique substitutions for variables that appear in the query.

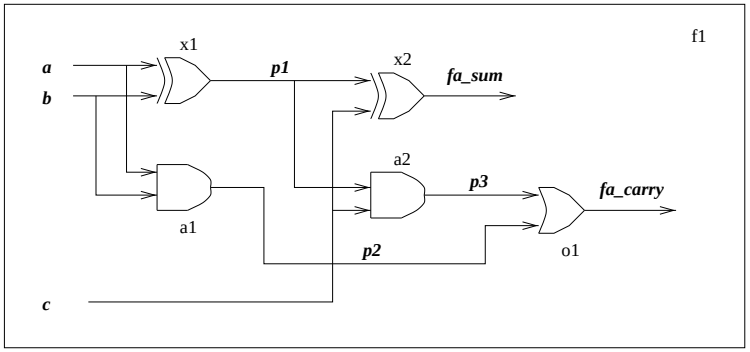
Reasoning about Circuits

As a case study, we look at electrical circuits. A KB can be used for several tasks here.

First we need a description language for our circuit; there are various choices. Here we will work with *logic gates* and with their *ports* – so we won't name connecting wires (though we do need to wknow what is connected to what).

Other possibilities exist.

A Full Adder



Description language

Predicates	*Functions*
$adder(x)$	$input(i, x)$
$xorg(x)$	$output(i, x)$
$org(x)$	
$andg(x)$	
$conn(x, y)$	

Describe configuration

Give the components and the connections:

$xorg(x1).$ $conn(input(1, f1), input(1, x1)).$
$xorg(x2).$ $conn(input(2, f1), input(2, x1)).$
$andg(a1).$ ⋮
⋮

Describe the behaviour; val describes the voltage associated with given points in the circuit (as 0 or 1).

$$\forall x\ andg(x) \wedge val(input(1, x), 1) \wedge (input(2, x), 1) \rightarrow val(output(1, x), 1)$$

Similarly,

$$\forall x\, \forall n\, andg(x) \wedge val(input(n,x),0) \rightarrow val(output(1,x),0)$$

Do this for all the components; now translate the circuit description into clausal form, eg

$$\{\neg \texttt{andg(v1)}, \neg \texttt{val(input(v2,v1),0)}, \texttt{val(output(1,v1),0)}\}$$

This gives us a KB for the circuit that we can use for several different purposes.

Simulation

Add to the KB some input values for the circuit:

$val(input(1,f1),1)$
$val(input(2,f1),0)$
$val(input(3,f1),1)$

Now use resolution to deduce that

$val(output(2,f1),1)$
$val(output(1,f1),0)$

- We get exactly these as clauses.
- Could have use the "answer" predicate.

Fault Diagnosis

Suppose there is something wrong with the circuit. Suppose we know the input and output values directly, and that they do not agree with the KB. Then some statement in the KB must be false.

Take the KB and the observed values; remove the statements classifying the components. Now resolution lets us conclude that

$$\{\ \neg xorg(x1), \neg xorg(x2)\ \}.$$

So either $x1$ or $x2$ must be faulty.

Diagnostic tests

We can use the KB to devise a test for faulty components.

The *single faulty component hypothesis* says that there is only one component malfunctioning.

We can express this as follows:

$\neg xorg(x1) \rightarrow (xorg(x2) \wedge andg(a1) \wedge \ldots$
$\neg xorg(x2) \rightarrow (xorg(x1) \wedge andg(a1) \wedge \ldots$
$\neg andg(a1) \rightarrow (xorg(x1) \wedge xorg(x2) \wedge \ldots$
$\vdots$

Adding this to the KB, we can use resolution to derive a "test clause" that says that *if* some component is OK, and given input values are assigned, *then* the output is determined; eg

$$\begin{aligned} xorg(x1) \quad & \wedge \quad val(input(1, f1), 1) \\ & \wedge \quad val(input(2, f1), 0) \\ & \wedge \quad val(input(3, f1), 0) \\ & \rightarrow \quad val(output(2, f1), 1) \end{aligned}$$

Now go and check the value – if it is not as predicted (and there is only one faulty component) then $x1$ is the faulty component.

Summary

Using KB and resolution for various tasks:

- find answer values
- to predict
- to diagnose
- to devise tests