University of Malta
BSc(Hons.) IT Year I

Introduction to Logic and Logic Programming

Lecture 12

Resolution

- Resolution Based Systems
- Clausal Form
- Propositional Resolution
- Horn Clauses
- Logic Programming

Department of Computer Science and AI
1988/9

Resolution Based Systems

- Resolution logic (a logic based on resolution) underlies most automatic theorem proving and logic programming systems.
- Resolution-logic systems are based on **refutation**: to prove that P follows from a set S of sentences, we assume -P and try to derive a contradiction. In other words, we try to show that the set

$$S \cup \{-P\}$$

is not satisfiable.
- Resolution logic operates with sentences in **clausal form**, a restricted kind of CNF in which each of the conjuncts is a **clause**.

Clausal Form: Some Definitions

Atom: A propositional symbol

p q r etc.

Literal: An atom or its negation

- positive e.g. p
- negative e.g. -p

Clause: a disjunction of literals.

p v -q v r v -s

but not p v -(q & r) v s

The latter can be turned into a clause by applying De Morgan's Law: to yield

p v -q v -r v s

Clausal Form

- Logically, clausal form is a restricted form of CNF

  $$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

  in which each conjunct C_i is a clause.

- We often represent clausal form as a **set** of clauses

  $$\{C_1, C_2, \ldots, C_n\}$$

  in which explicit mention of conjunction is supressed.

- In order to show that a set of clauses is unsatisfiable, it is sufficient to show that just one clause is unsatisfiable.

Propositional Resolution

Resolution is an operation that maps a pair of clauses having certain properties to a third clause called the *resolvent*.

$$\begin{array}{c}\text{clause1 clause2}\\ \Downarrow \\ \text{clause3}\end{array}$$

Suppose we have $C_1 \vee p$ and $C_2 \vee -p$ where C_1 and C_2 are clauses and p is an atom.

The rule of resolution allows us to eliminate the p and derive the formula $C_1 \vee C_2$.

$$C_1 \vee p,\ \ C_2 \vee -p \quad \vdash_{res} \quad C_1 \vee C_2$$

Example: $C_1 = p, \quad C_2 = r$

$$p \vee q,\ \ r \vee -q \quad \vdash_{res} \quad p \vee r$$

N.B. For resolution to work we must have a pair of complementary literals, in this case q and -q

Propositional Resolution

More generally, given

$$p_1 \vee, \ldots, \vee p_n$$
$$r_1 \vee, \ldots, \vee r_m$$

and i, j such that p_i and r_j are complimentary literals, the result of resolution is obtained by cancelling out literals p_i and r_j and forming the disjunction of the the union of the remaining literals. For example

$$\begin{array}{l} p \vee q \vee r \vee s \\ p \vee -q \vee t \end{array} \vdash_{res} p \vee r \vee s \vee t$$

Propositional Resolution

Note also the following special case:

$$p, -p \vdash_{res} \diamond$$

$\diamond$ is the symbol we use to denote the empty clause, which is unsatisfiable. Clearly, if *that* is ever derived, the entire set of clauses from which it is derived is also unsatisfiable.

The Resolution Principle

Let S be a set of clauses and let R(S) be the result of adding to S all possible resolvents of pairs of members of S. Then S is satisfiable if and only if R(S) is satisfiable

Hence S is unsatisfiable if and only if R(S) is unsatisfiable.

This is the basic principle underlying logic programming and automatic theorem provers.

Horn Clauses

Horn Clause: a clause with *at most* one positive literal. We distinguish three types.

1. zero negative literals, one positive literal:

$$p$$

2. n negative literals, one positive literal:

$$-p_1 \vee -p_2 \vee, ..., \vee -p_n \vee q$$

3. n negative literals, zero positive literals:

$$-p1 \vee -p2 \vee, ..., \vee -p_n$$

Special Cases of Resolution

These special cases are particularly relevant for logic programming.

- **Type 3 resolved with Type 1**
  - **terminal case:** length(type 3) = 1.

    $-p \quad , \quad p$

    $\vdash_{res} \diamond$
  - **general case:** length(type 3) = 1 + n

    $-p \vee -r_1 \vee, .., \vee - r_n \quad , \quad p$

    $\vdash_{res} -r_1 \vee, .., \vee - r_n$
- **Type 3 resolved with Type 2**

  $-p \vee -r_1 \vee, .., \vee - r_n \quad , \quad -q_1 \vee, .., \vee - q_n \vee p$

  $\vdash_{res} -q_1 \vee, .., \vee - q_n \vee -r_1 \vee, .., \vee - r_n$

Rewriting Propositional Horn Clauses

Note that the following clause of type 2

$$-p_1 \vee -p_2 \vee, ..., \vee -p_n \vee q$$

can be rewritten (using De Morgan's Laws)

$$-(p_1 \& p2, ..., \& p_n) \vee q$$

and thus in the form of an implication corresponding to a logic program rule:

$$p_1 \& p_2, ..., \& p_n \rightarrow q$$

Similarly, clauses of type 3

$$-p1 \vee -p2 \vee, ..., \vee -p_n$$

can be rewritten in the form of a negated conjunction, corresponding to (the negation of) a logic program query

$$-(p_1 \& p_2, ..., \& p_n)$$

Special Cases of Resolution Rewritten

Here are the rewritten versions.

- **Type 3 resolved with Type 1**. This corresponds to resolving a query (type 3) against a program fact (type 1).

  - **terminal case:** length(type 3) = 1.

    $-(p) \quad , \quad p$

    $\qquad \vdash_{res} \diamond$

  - **general case:** length(type 3) = 1 + n

    $-(p \& r_1 \&, .., \& r_n) \quad , \quad p$

    $\qquad \vdash_{res} -(r_1 \&, .., \& r_n)$

- **Type 3 resolved with Type 2.** This corresponds to resolving a query (type 3) against a rule (type 2).

  $-(p \& r_1 \&, .., \& r_n) \quad , \quad (q_1 \&, .., \& - q_n) \rightarrow p$

  $\qquad \vdash_{res} -(q_1 \&, .., \& - q_n \;\&\; r_1 \&, .., \& - r_n)$

Logic Programming and Deduction

A *logic program* comprises a set **P** of program clauses. The program together with a query **Q** describes a *deduction*.

$$\left.\begin{array}{l}\text{Program Clauses } \mathbf{P} \\ \text{type 1 (facts)e.g. } p, q \\ \text{type 2 (rules) e.g. } p\&q \rightarrow r \\ \\ \text{Query } \mathbf{Q} \\ \text{type 3 (goals), e.g. r}\end{array}\right\} \boxed{P \vdash Q}$$

The subsequent behaviour of the logic program arises out of the attempt to establish the deduction **using resolution**, i.e. by showing that the set of clauses

$$\{p, q, p\&q \rightarrow r, -r\}$$

is unsatisfiable. Next we show just that.

```
(-p v -q v r), p, q v -r
```

```
Program: (-p v -q v r), p, q
Query: r
1. -p v -q v r (program)
2. p                (program)
3. q                (program)
4. -r               (negated query)
5. -q v r           (res 1,2)
6. r                (res 5,3)
7. @                (res 4,6)
```

```
Program: (p & q -> r), p, q
Query: r
1. p & q -> r       (program)
2. p                (program)
3. q                (program)
4. -r               (negated query)
5. q -> r           (res 1,2)
6. r                (res 5,3)
7. @                (res 4,6)
```

Proofs and Subgoals

The left hand side gives the formal resolution proof. The right hand side shows the corresponding subgoals (subqueries, subproofs).

Pgm: `(-p v -q v s), (-t v -w v q), p, t, w`
Query: `s`

```
Proof
1.  -p v -q v s                  p & q -> s
2.  -t v -w v q                  t & w -> q
3.  p                                     p
4.  t                                     t
5.  w                                     w
                                 Subgoals
6.  -s                           s
7.  -p v -q     (res 1,6)        p,q     (from 1)
8.  -q          (res 7,3)          q     (from 3)
9.  -t v -w     (res 2,8)          t,w   (from 2)
10. -w          (res 9,4)            w   (from 4)
11. @           (res 5,10)           @   (from 5)
```

Resolution Trees

We can regard the last proof as a *linear* tree whose nodes contain *subgoals* that are either derived from the original query, else by resolving the clause containing the subgoal against program clauses of type 2. Subgoals are eliminated when such clauses are resolved against clauses of type 1.

$\vert$	query (clause 6)
$\boxed{s}$	
$\vert$	clause 1
$\boxed{p,q}$	
$\vert$	clause 3
$\boxed{q}$	
$\vert$	clause 2
$\boxed{t,w}$	
$\vert$	clause 4
$\boxed{w}$	
$\vert$	clause 5
$\boxed{@}$	

Branching Resolution Trees

In this case the tree is linear because because we never had to make a choice about which program clause to resolve with. In the following example we consider whether the logic program

$$\boxed{\begin{array}{l} p\&q \rightarrow r \\ w \rightarrow r \\ s \rightarrow w \\ p \\ s \end{array}}$$

implies the query r.

In this case we do have a choice. When we add the negated conclusion `-r` it could combine with either the first clause or the second. This choice causes the proof tree to branch.

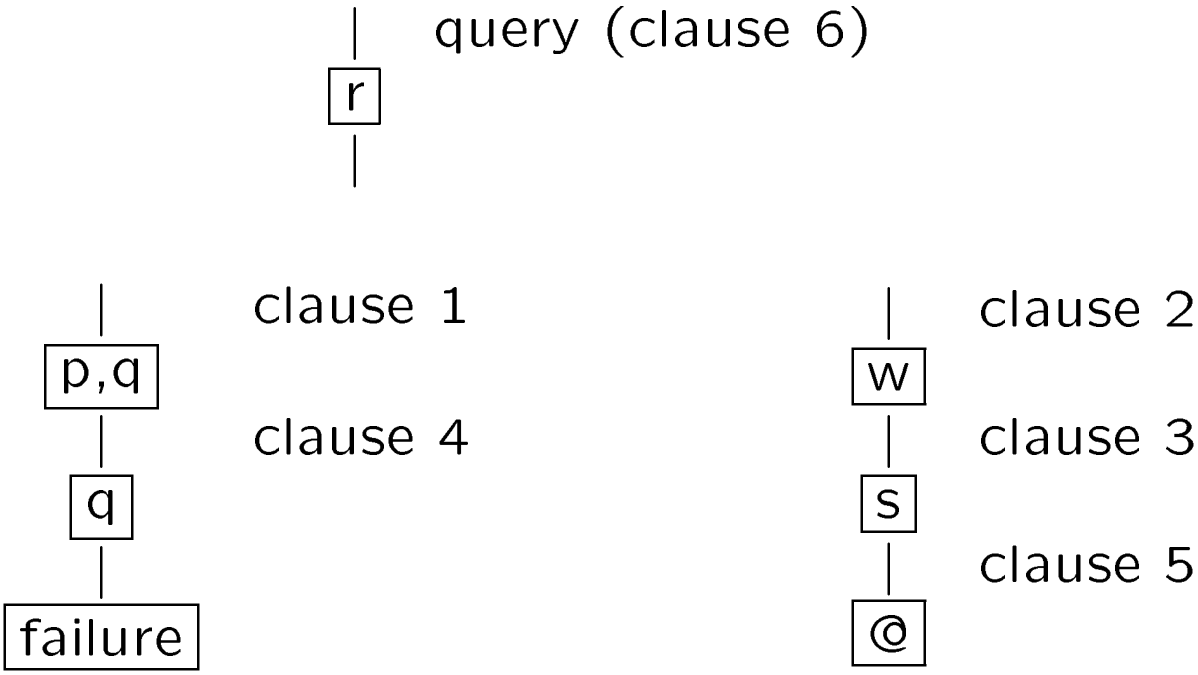
Branching Resolution Trees

$$p\&q \rightarrow r, w \rightarrow r, s \rightarrow w, p, s \vdash r$$

```
                  1. p & q -> r
                  2. w -> r
                  3. s -> w
                  4. p
                  5. s
                  6. -r
                     !
                     !
        ---------------------------------
        !                               !
        !                               !

7a. -(p & q)                        7b. -w
8a. -q                              8b. -s
                                    9b. @
```

Resolution Tree



In general, the hardest part of proving theorems concerns ordering decisions. Prolog adopts a particular ordering strategy known as SLD resolution.