Logic Programming

The Theoretical Basis of Logic Programming

December 16, 2015

Resolution

- Ground resolution was not practical.

Resolution

- Ground resolution was not practical.
- It turns out that a practical version of resolution is possible, using unification.

Resolution

- Ground resolution was not practical.
- It turns out that a practical version of resolution is possible, using unification.
- Recall the notions of literal, clause, clause sets introduced in the previous lecture.

Resolution

- Ground resolution was not practical.
- It turns out that a practical version of resolution is possible, using unification.
- Recall the notions of literal, clause, clause sets introduced in the previous lecture.
- **Notation.** Let L be a literal. We denote with L^c the complementary literal (i.e. L and L^c are opposite, one is the negation of the other).

Definition (General resolution step)

Let C_1, C_2 be clauses *with no variables in common*. Let $L_1 \in C_1$ and $L_2 \in C_2$ be literals in the clauses such that L_1 and ${L_2}^c$ can be unified by a mgu σ. Then C_1 and C_2 are said to be **clashing clauses**, that **clash on the literals** L_1 and L_2, and **resolvent of** C_1 **and** C_2 is the clause:

$$Res(C_1, C_2) = (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma).$$

Example (Resolvent of two clauses)

Consider the clauses:

$$p(f(x), g(y)) \vee q(x, y) \qquad \neg p(f(f(a)), g(z)) \vee q(f(a), g(z))$$

$L_1 = p(f(x), g(y))$ and ${L_2}^c = p(f(f(a)), g(z))$ can be unified with the mgu $\{x \leftarrow f(a), y \leftarrow z\}$ and the resolvent of the clauses is:

$$q(f(a), z) \vee q(f(a), g(z)).$$

- Note that the requirement for clauses to have no variables in common does not impose any real restrictions on the clause set. Remember that clauses are implicitly universally quantified, so changing the name of a variable does not change the meaning of the clause set.

General Resolution Procedure.

INPUT: A set of clauses S.

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat
  - **Choose** $C_1, C_2 \in S_i$ clashing clauses and let $C = Res(C_1, C_2)$.

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat
  - **Choose** $C_1, C_2 \in S_i$ clashing clauses and let $C = Res(C_1, C_2)$.
  - If $C = \emptyset$ terminate with the answer "not satisfiable".

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat
  - **Choose** $C_1, C_2 \in S_i$ clashing clauses and let $C = Res(C_1, C_2)$.
  - If $C = \emptyset$ terminate with the answer "not satisfiable".
  - Otherwise, $S_{i+1} = S_i \cup \{C\}$.

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat
  - **Choose** $C_1, C_2 \in S_i$ clashing clauses and let $C = Res(C_1, C_2)$.
  - If $C = \emptyset$ terminate with the answer "not satisfiable".
  - Otherwise, $S_{i+1} = S_i \cup \{C\}$.
- until $S_{i+1} = S_i$

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

General Resolution Procedure.

INPUT: A set of clauses S.

OUTPUT: S is satisfiable or S is not satisfiable. Also the algorithm may not terminate.

- Start with $S_0 = S$.
- Repeat
  - **Choose** $C_1, C_2 \in S_i$ clashing clauses and let $C = Res(C_1, C_2)$.
  - If $C = \emptyset$ terminate with the answer "not satisfiable".
  - Otherwise, $S_{i+1} = S_i \cup \{C\}$.
- until $S_{i+1} = S_i$
- Return "satisfiable".

Note that the algorithm above may not terminate, it is not a decision procedure (indeed that would not be expected since first order predicate logic is undecidable). The reason for nontermination is the existence of infinite models.

Example (Resolution refutation, from [Ben-Ari, 2001])

1.	$\neg p(x) \vee q(x) \vee r(x, f(x))$		
2.	$\neg p(x) \vee q(x) \vee s(f(x))$		
3.	$t(a)$		
4.	$p(a)$		
5.	$\neg r(a, y) \vee t(y)$		
6.	$\neg t(x) \vee \neg q(x)$		
7.	$\neg t(x) \vee \neg s(x)$		
8.	$\neg q(a)$	$x \leftarrow a$	$3, 6$
9.	$q(a) \vee s(f(a))$	$x \leftarrow a$	$2, 4$
10.	$s(f(a))$		$8, 9$
11.	$q(a) \vee r(a, f(a))$	$x \leftarrow a$	$1, 4$
12.	$r(a, f(a))$		$8, 11$
13.	$t(f(a))$	$y \leftarrow f(a)$	$5, 12$
14.	$\neg s(f(a))$	$x \leftarrow f(a)$	$7, 13$
15.	$\emptyset$		$10, 14$

Example (Resolution refutation with variable renaming, from [Ben-Ari, 2001])

First four clauses represent the initial clause set.

1.	$\neg p(x,y) \vee p(y,x)$	
2.	$\neg p(x,y) \vee \neg p(y,z) \vee p(x,z)$	
3.	$p(x, f(x))$	
4.	$p(x,x)$	
$3'$.	$p(x', f(x'))$ Rename 3.	
5.	$p(f(x),x)$	$\sigma_1 = \{y \leftarrow f(x), x' \leftarrow x\}1, 3'$
$3''$.	$p(x'', f(x''))$ Rename 3	
6.	$\neg p(f(x),z) \vee p(x,z)$	$\sigma_2 = \{y \leftarrow f(x), x'' \leftarrow x\}2, 3'$
$5'''$.	$p(f(x'''),x''')$ Rename 5	
7.	$p(x,x)$	$\sigma_3 = \{z \leftarrow x, x''' \leftarrow x\}6, 5'''$
$4''''$.	$\neg p(x'''', x'''')$ Rename 4	
8.	$\emptyset$	$\sigma_4 = \{x'''' \leftarrow x\}7, 4''''$

- The substitution resulting from composing all intermediary substitutions:

$$\sigma = \sigma_1\sigma_2\sigma_3\sigma_4 = \{y \leftarrow f(x), z \leftarrow x, x' \leftarrow x,$$

$$x'' \leftarrow x, x''' \leftarrow x, x'''' \leftarrow x\}$$

- Restricted to the variables from the initial set, the resulting substitution is:

$$\sigma = \{y \leftarrow f(x), z \leftarrow x\}$$

Theorem (Soundness of substitution)

If the unsatisfiable clause $\emptyset$ is derived during the general resolution procedure, then the set of clauses is unsatisfiable.

Theorem (Completeness of substitution)

If a set of clauses is unsatisfiable, then the empty clause $\emptyset$ can be derived by the resolution procedure.

- For details on how the proofs of these theorems, see [Ben-Ari, 2001].

Some remarks on the resolution procedure

- Note that the resolution procedure is nondeterministic: which clashing clause to choose and which clashing literals to resolve on is not specified.
- Good choices will lead to the result quickly, while bad choices may lead to the algorithm not terminating.
- The completeness theorem says that if the clause set is unsatisfiable a resolution refutation (generation of the empty clause) exists, i.e. that which uses good choices. Variants with bad choices may miss the solution.

- Consider a fragment of the theory of strings, with the binary function symbol $\cdot$ (concatenation) and binary predicates *substr*, *prefix*, *suffix*, described by the following axioms:

  1. $\forall x\ \mathit{substr}(x, x)$
  2. $\forall x \forall y \forall z\ ((\mathit{substr}(x, y) \wedge \mathit{suffix}(y, z)) \Rightarrow \mathit{substr}(x, z))$
  3. $\forall x \forall y\ \mathit{suffix}(x, y \cdot x)$
  4. $\forall x \forall y \forall z\ ((\mathit{substr}(x, y) \wedge \mathit{prefix}(y, z)) \Rightarrow \mathit{substr}(x, z))$
  5. $\forall x \forall y\ \mathit{prefix}(x, x \cdot y)$

- The procedural interpretation of these formulae is:
  1. x is a substring of x.
  2. To check is x is a substring of z, find a suffix y of z and check if x is a substring of y.
  3. x is a suffix of $y \cdot x$,
  4. To check if x is a substring of z, find a prefix y of z and check if x is a substring of y.
  5. x is a prefix of $x \cdot y$.

- The clausal form of these axioms is:

1. $substr(x, x)$
2. $\neg substr(x, y) \vee \neg suffix(y, z) \vee substr(x, z)$
3. $suffix(x, y \cdot x)$
4. $\neg substr(x, y) \vee \neg prefix(y, z) \vee substr(x, z)$
5. $prefix(x, x \cdot y)$

- Now consider a refutation of $\neg substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$:

6.	$\neg substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$	
7.	$\neg substr(a \cdot b \cdot c, y1) \vee \neg suffix(y1, a \cdot a \cdot b \cdot c \cdot c)$	$6, 2$
8.	$\neg substr(a \cdot b \cdot c, a \cdot b \cdot c \cdot c)$	$7, 3$
9.	$\neg substr(a \cdot b \cdot c, y2) \vee \neg prefix(y2, a \cdot b \cdot c \cdot c)$	$8, 4$
10.	$\neg substr(a \cdot b \cdot c, a \cdot b \cdot c)$	$9, 5$
11.	$\emptyset$	$10, 1$

i.e. we have shown by resolution that
$substr(a \cdot b \cdot c, a \cdot a \cdot b \cdot c \cdot c)$.

- Another way to use resolution is to check whether

$$\exists w substring(w, a \cdot a \cdot b \cdot c \cdot c)$$

Denoting the axioms by the formula *Axioms*, we must show that the following formula is unsatisfiable:

$$Axioms \wedge \neg(\exists w substring(w, a \cdot a \cdot b \cdot c \cdot c)).$$

But this is

$$Axioms \wedge \forall w(\neg substring(w, a \cdot a \cdot b \cdot c \cdot c))$$

which can be written in a straightforward manner into clausal form (and resolution is applicable).

- A resolution refutation works with the substitution $\{w \leftarrow a \cdot b \cdot c\}$: not only was the logical consequence proved, but a value for w was **computed** such that $substring(w, a \cdot a \cdot b \cdot c \cdot c)$ is true. In this context, the string axiom constitute a **program** that computes **answers** to questions. However, the program is highly nondeterministic. Choices that can be made during the execution of the program (by resolution) influence the result and even the answer.
- The nondeterministic formalism can be turned into a practical logic programming language by specifying rules for making choices.

Definition (Horn Clauses)

A **Horn clause** is a clause $A \Leftarrow B_1, \ldots, B_n$ with at most one positive literal. The positive literal A is called the **head** and the negative literals B_i form the **body**. A unit positive Horn clause $A \Leftarrow$ is called a **fact**, and a Horn clause with no positive literals $\Leftarrow B_1, \ldots, B_n$ is called a **goal clause**. A Horn clause with one positive literal and one or more negative literals is called a **program clause**.

Note that the notation $A \Leftarrow B_1, \ldots, B_n$ is equivalent to $(B_1 \wedge \ldots \wedge B_n) \Rightarrow A$ which in turn is equivalent to $\neg B_1 \vee \ldots \vee \neg B_n \vee A$.

- The notation used in the first part of this lecture (in particular by the Prolog syntax) for program clauses is A:-$B_1, \ldots, B_n$. From now on we will use this notation.

Definition (Logic programs)

A set of non-goal Horn clauses whose heads have the same predicate is called a **procedure**. A set of procedures if a **(logic) program**. A procedure composed only of ground facts is called a **database**.

Definition

A **computation rule** is a rule for choosing a literal in a goal clause to solve with. A **search rule** is a rule for choosing a clause to resolve with the chosen literal in the goal clause.

- The difference between logic programming and imperative programming is the control of the program:
  - in imperative programming the control of the program is given explicitly as part of the code by the programmer,
  - in logic programming the programmer writes declarative formulae that describe the relationship between the input and the output, and resolution together with the search and computation rules supply an **uniform** control structure.
- As a consequence of the uniform control structure, there are cases when logic programs will not be as efficient as special handcrafted control structures for specific computations.

Example (Logic program, from [Ben-Ari, 2001])

The following program has two procedures:

1. $q(x, y)$:- $p(x, y)$
2. $q(x, y)$:- $p(x, z), q(z, y)$

3. $p(b, a)$
4. $p(c, a)$
5. $p(d, b)$
6. $p(e, b)$
7. $p(f, b)$
8. $p(h, g)$
9. $p(i, h)$
10. $p(j, h)$

Definition (Correct answer substitution)

Let P be a program and G a goal clause. A substitution θ for variables in G is called a **correct answer substitution** iff $P \models \forall(\neg G\theta)$, where $\forall$ denotes the universal closure of the variables that are free in G.

In other words, the correct answer substitution makes the negation of the goal clause a logical consequence of the program.

Example

Consider a refutation for the goal clause $:\text{-}q(y, b), q(b, z)$ from the program introduced in Example 10. At each step, choose a literal within the goal clause and a clause whose head clashes with that literal:

1. Choose $q(y, b)$ and resolve with the clause 1, obtain $:\text{-}p(y, b), q(b, z)$.
2. Choose $p(y, b)$ and resolve with clause 5, obtain $:\text{-}q(b, z)$. The needed substitution is $\{y \leftarrow d\}$.
3. Choose the remaining literal $:\text{-}q(b, z)$ and resolve with clause 1, obtain $:\text{-}p(b, z)$.
4. Choose the remaining literal $:\text{-}p(b, z)$ and resolve with clause 3, obtain $\emptyset$. The needed substitution is $\{z \leftarrow a\}$.

We obtained the empty clause $\emptyset$. With the correct answer substitution $\{y \leftarrow d, z \leftarrow a\}$ applied to the goal, we get that

$$P \vDash q(d, b) \wedge q(b, a).$$

Definition (SLD Resolution)

Let P be set of program clauses, R a computation rule and G a goal clause. A **derivation by SLD-resolution** is defined as a sequence of resolution steps between the goal clause and the program clauses. The first goal clause G_0 is G. Assume that G_i has been derived. G_{i+1} is defined by **selecting** a literal $A_i \in G_i$ according to the computation rule R, **choosing** a clause $C_i \in P$ such that the head of C_i unifies with A_i by mgu θ_i and resolving:

$$\begin{aligned} G_i &= \text{:-}A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n \\ C_i &= A\text{:-}B_1, \ldots, B_k \\ A_i\theta_i &= A\theta_i \\ G_{i+1} &= \text{:-}(A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta_i \end{aligned}$$

An **SLD refutation** is an SLD-derivation of $\emptyset$.

Soundness and completeness of SLD-resolution

Theorem (Soundness of SLD-resolution)

Let P be a set of program clauses, R a computation rule and G a goal clause. If there is an SLD-refutation of G, $\theta = \theta_1 \dots \theta_n$ is the sequence of unifiers used in the refutation and σ is the restriction of θ to the variables of G, then σ is a correct answer substitution for G.

Proof.

See [Ben-Ari, 2001], pp. 178. □

Theorem (Completeness of SLD-resolution)

Let P be a set of program clauses, R a computation rule and G a goal clause. Let σ be a correct answer substitution. Then there is an SLD-resolution of G from P such that σ is the restriction of the sequence of unifiers $\theta = \theta_1, \ldots, \theta_n$ to the variables in G.

Proof.

See [Ben-Ari, 2001], pp. 179. □

- Note that the above results only refer to Horn clauses (logic programs), not to arbitrary clauses. SLD resolution is not complete for arbitrary clauses:

$$p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q$$

is unsatisfiable, but there is no SLD-resolution of $\emptyset$ for it (exercise!).

Example (Examples 10, 12, revisited)

- If, in step 2 of Example 12 clause 6 would have been chosen to resolve with, the resolvent would have been $\text{:-}q(b, z)$. The resolution will succeed (exercise!) but the correct answer substitution would have been different: $\{y \leftarrow e, z \leftarrow a\}$. For a given goal clause there may be several correct answer substitutions.
- Suppose that the computation rule is to always choose the last literal in the goal clause. Resolving always with clause 2 gives:

$$\begin{aligned}
&\text{:-}q(y, b), q(b, z)\\
&\text{:-}q(y, b), p(b, z'), q(z', z)\\
&\text{:-}q(y, b), p(b, z'), p(z', z''), q(z'', z)\\
&\text{:-}q(y, b), p(b, z'), p(z', z''), p(z'', z'''), q(z''', z)\\
&\ldots
\end{aligned}$$

A correct answer substitution exists but this resolution will not terminate.

- Consider the computation rule that always chooses the first literal in the goal. The SLD resolution proceeds as follows:
  1. $q(y, b)$ is chosen and resolved with clause 2, obtain $\text{:-}p(y, z'), q(z', b), q(b, z)$.
  2. Choose the first literal $p(y, z')$ and resolve it with clause 6 $p(e, b)$, then with clause 1 and obtain $\text{:-}q(b, b), q(b, z)$, then $\text{:-}p(b, b), q(b, z)$.
  3. No program clause unifies with $p(b, b)$ and this resolution fails.

  Even though an answer substitution exists, resolution fails.

Definition (SLD-trees)

Let P be a set of program clauses, R a computation rule and G a goal clause. All possible SLD-derivations can be displayed on an **SLD-tree**, constructed in the following manner:

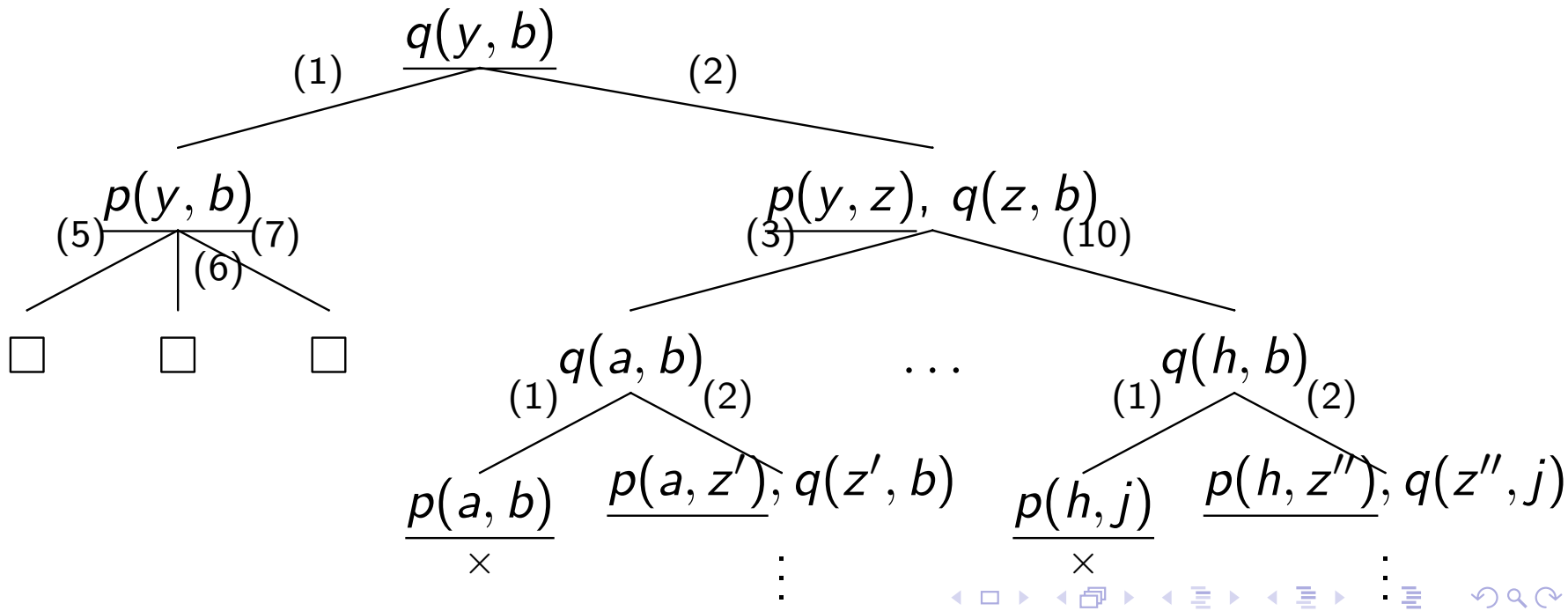
- Te root is labeled with the goal clause G.
- Given a node n labeled with a goal clause G_n, create a child n_i for each new goal clause G_{n_i} that can be obtained by resolving the literal chosen by R with the head of a clause from P.

Definition

In an SLD-tree, a branch leading to a refutation is called a **success branch**. A branch leading to a goal clause which cannot be resolved is called a **failed branch**. A branch corresponding to a non-terminating derivation is called an **infinite branch**.

Example (SLD-tree)

The following is the SLD tree for the derivation in Examples 10, 12 where the choice rule is to resolve with the leftmost literal in the goal (see also Example 16). The chosen literal is underlined, the clause resolved with is the label of the edge. Successful branches are indicated with $\square$, failed branches with $\times$, infinite branches with $\vdots$.



Theorem

Let P be a program and G be a goal clause. Then every SLD-tree for P and G has infinitely many success branches, or they all have the same finite number of success branches.

Proof.

Not given here. □

Definition

A **search rule** is a procedure for searching an SLD-tree for a refutation. A **SLD-refutation procedure** is the SLD-resolution algorithm together with the specification of a computation rule and a search rule.

Some comments on the completeness of SLD resolution

- Note that the SLD resolution is complete for any computation rule (i.e. a refutation exists). However, the choice of the search rule is essential in whether or not this refutation exists.

Some comments on the completeness of SLD resolution

- Note that the SLD resolution is complete for any computation rule (i.e. a refutation exists). However, the choice of the search rule is essential in whether or not this refutation exists.
- If a more restricted notion of completeness is considered (a refutation exists and is found), certain search rule make SLD resolution incomplete: for example the depth-first rule search rule (which Prolog uses: Prolog = SLD resolution with leftmost literal as computations rule and depth-first search rule).

Some comments on the completeness of SLD resolution

- Note that the SLD resolution is complete for any computation rule (i.e. a refutation exists). However, the choice of the search rule is essential in whether or not this refutation exists.
- If a more restricted notion of completeness is considered (a refutation exists and is found), certain search rule make SLD resolution incomplete: for example the depth-first rule search rule (which Prolog uses: Prolog = SLD resolution with leftmost literal as computations rule and depth-first search rule).
- There are search rules for which SLD resolution is complete (in the stronger sense):

However, these complete search rules are computationally expensive. A trade-off between completeness and efficiency is made.

Some comments on the completeness of SLD resolution

- Note that the SLD resolution is complete for any computation rule (i.e. a refutation exists). However, the choice of the search rule is essential in whether or not this refutation exists.
- If a more restricted notion of completeness is considered (a refutation exists and is found), certain search rule make SLD resolution incomplete: for example the depth-first rule search rule (which Prolog uses: Prolog = SLD resolution with leftmost literal as computations rule and depth-first search rule).
- There are search rules for which SLD resolution is complete (in the stronger sense):
  - breath-first (search every level of the SLD tree),

However, these complete search rules are computationally expensive. A trade-off between completeness and efficiency is made.

Some comments on the completeness of SLD resolution

- Note that the SLD resolution is complete for any computation rule (i.e. a refutation exists). However, the choice of the search rule is essential in whether or not this refutation exists.
- If a more restricted notion of completeness is considered (a refutation exists and is found), certain search rule make SLD resolution incomplete: for example the depth-first rule search rule (which Prolog uses: Prolog = SLD resolution with leftmost literal as computations rule and depth-first search rule).
- There are search rules for which SLD resolution is complete (in the stronger sense):
  - breath-first (search every level of the SLD tree),
  - bounded depth first (go down up to a certain depth, then try another branch down to that depth, and so on, if the solution is not found, increase the depth).

  However, these complete search rules are computationally expensive. A trade-off between completeness and efficiency is made.

- Read: Chapter 7, sections 7.5-7.8 of [Ben-Ari, 2001]; Chapter 8, sections 8.1-8.3 of [Ben-Ari, 2001].
- Also read: Chapter 2, Chapter 3 of [Nilsson and Maluszynski, 2000].


Ben-Ari, M. (2001).
Mathematical Logic for Computer Science.
Springer Verlag, London, 2nd edition.


Nilsson, U. and Maluszynski, J. (2000).
Logic, Programming and Prolog.
copyright Ulf Nilsson and Jan Maluszynski, 2nd edition.