Chapter 8: Logic Programming

November 24, 2008

Outline

8.1 Formulas as Programs

Most axioms in deductive systems are in one of the following two forms:

1. $\forall x_1 \forall x_2 \ldots \forall x_k (B_1 \wedge B_2 \ldots \wedge B_n \to B)$; or
2. $\forall x_1 \forall x_2 \ldots \forall x_k \; B$

where $B_i (i = 1, \ldots, n),\; B$ are atomic formulas.

These formulas can be interpreted as:

1. To check that the goal B is satisfied, verify that the conditions $B_1, \ldots, B_n$ are met.
2. B is always true without any conditions (i.e. B is a fact)

General Problem

Given a set of assumptions (axioms) which are in one of the following two forms:

1. $\neg B_1 \vee \ldots \neg B_n \vee B$
2. B

where $B_1, \ldots, B_n, B$ are atomic formulas, check if a formula G (goal) logically follows from these axioms.

- We add $\neg G$ and try to refute it using resolution.
- After each stop in resolution, we either get a resolvent of the form

  $$\neg C_1 \vee \ldots \vee \neg C_m$$

  in which all atomic formulas are negated, or the empty clause.

Example

Consider the following logic program:

1. $\forall x\ substr(x, x)$
2. $\forall x \forall y \forall z[(substr(x, y) \wedge suffix(y, z)) \rightarrow substr(x, z)]$
3. $\forall x\ suffix(x, y \cdot x)$
4. $\forall x \forall y \forall z[(substr(x, y) \wedge prefix(y, z)) \rightarrow substr(x, z)]$
5. $\forall x\ prefix(x, x \cdot y)$

in the language of string concatenation and binary predicates *substr*, *prefix*, and *suffix*.
Suppose we want to check if *abc* is a substring of *aabcc*, so our goal clause is

$$substr(abc, aabcc)$$

We will try to refute its negation

$$\neg substr(abc, aabcc)$$

Our program in clausal form takes the following form:

1. $\mathit{substr}(x, x)$
2. $\neg \mathit{substr}(x, y) \lor \neg \mathit{suffix}(y, z) \lor \mathit{substr}(x, z)$
3. $\mathit{suffix}(x, yx)$
4. $\neg \mathit{substr}(x, y) \lor \neg \mathit{prefix}(y, z) \lor \mathit{substr}(x, z)$
5. $\mathit{prefix}(x, xy)$

Then, the resolution refutation of $\neg substr(abc, aabcc)$ proceeds in the following way:

6.	$\neg substr(abc, aabcc)$	Goal
7.	$\neg substr(abc, y) \vee \neg suffix(y, aabcc)$	6,2
8.	$\neg substr(abc, abcc)$	7,3
9.	$\neg substr(abc, y) \vee \neg prefix(y, abcc)$	8,4
10.	$\neg substr(abc, abc)$	9,5
11.	$\square$	10,1

- From now on, we can write the axioms involving implication as

$$\forall x_1 \forall x_2 \ldots \forall x_k (B \leftarrow B_1 \wedge \ldots \wedge B_n)$$

- We interpret such axioms as:

  "In order to compute B, compute $B_1, \ldots, B_n$ first."

Example

In this interpretation, our previous logic program becomes:

1. x is a substring of x
2. To check if 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 yx
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 xy.

- The programs obtained in this way are highly nondeterministic.

Two main issues:

1. If we have a goal clause $B_1 \vee \ldots \vee B_n$, which of the atoms $B_1, B_2, \ldots, B_n$ can we use in resolution?
2. Once we decide on B_i to use in resolution, which other clause containing $\neg B_i$ should we use to resolve with it?

Definition

A computation ruleis a rule for choosing a literal in a goal clause to use in resolution. A search rule is a rule for choosing a clause in the program to resolve with the chosen literal in a goal clause.

Main difference between logic programming and ordinary (algorithmic) programming:

- In algorithmic programming, the programmer has **effective control** over program execution, generally through constructs such as `IF..THEN`, `FOR..DO`, `WHILE..DO`, etc.
- In logic programming, the control over the execution of the program is completely supplied by the resolution engine (compiler) and is said to be **uniform control**, realized by the declared relationships between the input and the output.

8.2 SLD-Resolution

Definition

A Horn clause is a clause containing at most one positive literal.

- A Horn clause can have one of the following three forms:
  1. $\neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n \vee A$
  2. $\neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$
  3. A

In the notation from Sec.8.1, we have following forms of Horn clauses:

1. $$A \leftarrow B_1, B_2, \ldots, B_n$$

   A is called the head and $B_1, \ldots, B_n$ the tail of the Horn clause.

2. $$\leftarrow B_1, B_2, \ldots, B_n$$

   In this case, the head is empty and such clause is called a goal clause

3. $$A \leftarrow$$

   In this type of clause, the tail is empty, and such clause is said to be a fact.

Definition

- Procedure: a set of non-goal Horn clauses with the same head.
- Logic program: a set of procedures.
- Database: a procedure composed of ground facts.

Example

$$1. q(x,y) \leftarrow p(x,y).$$
$$2. q(x,y) \leftarrow p(x,z), q(z,y).$$

$$\begin{array}{ll} 3. p(b,a). & 7. p(f,b). \\ 4. p(c,a). & 8. p(h,g). \\ 5. p(d,b). & 9. p(i,h). \\ 6. p(e,b). & 10. p(j,h). \end{array}$$

The first two clauses form a procedure. The remaining clauses (3)-(10) are ground facts that constitute the database of the program.

Definition

Suppose P is a logic program and G the goal clause.
If θ is a substitution of variables in G, we say that θ is a correct answer substitution, if

$$P \models \forall\,(\neg G\theta)$$

[Keep in mind that, if G is the goal clause for resolution, we are trying to show that P together with G form an unsatisfiable set of clauses; in other words, $\neg G$ is a consequence of the program clauses P.]

Example

Suppose P is the logical program which consists of the usual axioms (rules) for arithmetic in $\mathbb{Z}$, and let

$$G = \neg(x + 3 = y)$$

be the goal clause for P.

One correct answer substitution is

$$\theta = \{x \leftarrow 2, y \leftarrow 5\}$$

since

$$P \models 2 + 3 = 5$$

Another correct answer substitution is, for example,

$$\theta = \{x \leftarrow y - 3\}$$

since

$$P \models \forall y \; ((y - 3) + 3 = y)$$

$\square$

General Problem

Given a logic program P and the formula

$$B = \exists\, (B_1 \wedge B_2 \wedge \ldots \wedge B_n),$$

where $B_1, \ldots, B_n$ are atomic formulas, check if B is a logical consequence of P:

$$P \models B$$

Then,

$$P \models B \Longleftrightarrow P \models (B_1 \wedge \ldots \wedge B_n)\sigma$$

for some ground substitution σ.

Let θ be a substitution, so that, for **any** ground substitution λ, $\sigma = \theta\lambda$ [Such a substitution θ always, exists; take e.g $\theta = \sigma$.]

$$P \models (B_1 \wedge \ldots \wedge B_n)\theta\lambda, \quad \text{for all } \lambda$$

Then,

$$P \models \forall((B_1 \wedge \ldots \wedge B_n)\theta)$$

So, we are looking for a correct answer substitution θ for the goal clause

$$G = \neg(B_1 \wedge \ldots \wedge B_n) = \neg B$$

Therefore,

$$\begin{aligned} P \models B &\iff \models P \to B \\ &\iff \neg(P \to B) \text{ is unsatisfiable} \\ &\iff P \wedge \neg B \text{ is unsatisfiable} \\ &\iff P \wedge G \text{ is unsatisfiable} \end{aligned}$$

Example

For the logic program P from the beginning of this section, suppose we want to check if

$$\exists y \exists z\ (q(y, b) \wedge q(b, z))$$

follows logically from P and, if so, find a correct answer substitution for y, z

11.	$\leftarrow q(y, b), q(b, z).$	Goal clause	
12.	$\leftarrow p(y, b), q(b, z).$	Res 1,11	
13.	$\leftarrow q(b, z).$	Res 5,12	$\{y \leftarrow d\}$
14.	$\leftarrow p(b, z).$	Res 1,13	
15.	$\square$	Res 3,14	$\{z \leftarrow a\}$

In the process of refuting the goal clause, we used the substitution $\theta = \{y \leftarrow d, z \leftarrow a\}$, so

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

SLD-Resolution

Suppose P is a set of program clauses, R the computation rule, and G a goal clause.
A derivation using SLD-resolution consists of a sequence of steps between goal clauses and program clauses, in the following way:

- $G_0 := G$;
- Suppose G_i has been derived
- To find G_{i+1}, first select a literal A_i in G_i, using the computation rule R. Then, choose a clause C_i in P such that the head of C_i unifies with A_i using an m.g.u. θ_i and resolve:

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

An SLD-refutation is a derivation of the empty clause $\Box$ using SLD-resolution.

Theorem

SLD-resolution is sound and complete.

Examples

(a) In the preceding example, suppose that, at step 2, we had used the clause (6) $p(e, b)$ for resolution. In that case, we would have obtained

$$\leftarrow q(b, z)$$

as the resolvent and, eventually, we would have computed a different correct answer substitution

$$\theta = \{y \leftarrow e, z \leftarrow a\}$$

So, given a program P and a goal clause G, there may be more than one correct answer substitution.

(b) Suppose that, for the same example, we had always used the **last** literal in the goal clause to resolve with and that we had always used the second program clause.
The computation would have had the following form:

$$
\begin{array}{ll}
G_0 : & \leftarrow q(y,b), \underline{q(b,z)}. \\
G_1 : & \leftarrow q(y,b), \overline{p(b,u)}, \underline{q(u,z)}. \\
G_2 : & \leftarrow q(y,b), p(b,u), \overline{q(u,v)}, \underline{q(v,z)}. \\
G_3 : & \leftarrow q(y,b), p(b,u), q(u,v), \overline{q(v,w)}, \underline{q(w,z)}. \\
 & \vdots
\end{array}
$$

So, the refutation fails and, in fact, the computation never terminates.

(c) Finally, suppose that we had always used the first literal in the goal clause:

$$
\begin{aligned}
G_0 &: \quad \leftarrow \underline{q(y,b)}, q(b,z). \\
G_1 &: \quad \leftarrow \underline{p(y,u)}, q(u,b), q(b,z). \\
G_2 &: \quad \leftarrow \underline{q(a,b)}, q(b,z). \\
G_3 &: \quad \leftarrow p(a,b), q(b,z).
\end{aligned}
$$

However, there is no way to proceed past this point, since $p(a,b)$ is not in the database. So, the refutation fails, as in (b), but the computation terminates.

Definition

Let P be a set of program clauses, R a computation rule, and G a goal clause. All possible SLD-derivations can be displayed using an SLD-tree. The root of the tree is labeled by the goal clause G. Given a node n labeled with a goal clause G_i, its children will be all new goal clauses that are obtained by resolving the literal in G_i, chosen by R with the head of a clause in P.

Definition

In an SLD-tree, a branch leading to a refutation is called a success branch. A terminating branch leading to a non-refutation is called a failure branch, while a non-terminating branch is called an infinite branch.

Theorem

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

Definition

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

[E.g. two common search rules are **breadth-first** and **depth-first** searches of the SLD-tree.]

8.3 Prolog

- Prolog was the first programming language that was based on priciples of logic programming.
- It is an implementation of SLD-resolution on Horn clauses.

Basics

- **Computation rule:** choose the **leftmost** literal in the goal clause.
- **Search rule:** program clauses are examined for resolution **top-to-bottom**.
- **Variables and constants:** variables must start with upper-case letters, while constants must start with lower-case letters.
- the symbol $:-$ is used in place of $\leftarrow$.

Example

The first example from Section 8.2 can be interpreted as a Prolog program in the following way:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
parent(bob,allen).           parent(fred,bob).
parent(catherine,allen).     parent(henry,george).
parent(dave,bob).            parent(ida,henry).
parent(ellen,bob).           parent(joe,henry).
```

When posed the goal clause

```
:- ancestor(Y,bob),ancestor(bob,Z).
```

the program generates the correct answer substitution

```
:- Y=dave, Z=allen
```

- Since the search rule is always top-to-bottom, the SLD-tree is always traversed depth-first. This can result in non-termination or failure, even if there exists a success branch.
- For this reason, when writing program clauses, we must pay attention to the ordering of atoms in the tail of a clause.
- Since we may encounter a failure before reaching a success branch, we need to maintain a list of backtrack points; these are pointers to the previous nodes from which there are multiple branches.

Example

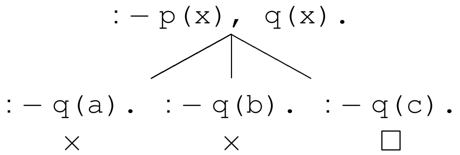
Consider the program

```
p(a).  p(b).  p(c).  q(c).
```

with the goal clause

```
:- p(x), q(x).
```

The SLD-tree is:



- Often, we need to use recursive statements such as

```
q(X,Y):-p(X,Z),q(Z,Y).
```

- Every program containing a recursive clause will possibly have an infinite SLD-tree since, at each node where we use this type of clause, one of the descendant nodes will contain `q`.
- Search points for recursion are in Prolog effectively stored using a stack.

Forcing Failure

In our previous example, the query

```
:- ancestor(Y,bob),ancestor(bob,Z).
```

would generate the first successful outcome in the SLD-tree

```
Y=dave, Z=allen.
```

and stop. What to do if we are looking for a different answer? The modified query

```
:- ancestor(Y,bob),ancestor(bob,Z),fail.
```

will, after generating the first answer, encounter the default failure clause `fail`, backtrack and try to resolve the query using the next available branch, so it would generate the next answer in the SLD-tree:

```
Y=ellen, Z=allen.
```

This is a way to simulate `FOR..DO` or `UNTIL..DO` constructions in Prolog.

- The syntax of Prolog contains a number of **non-logical predicates**
- Some non-logical predicates include I/O predicates `get` (meaning: get a character from the keyboard), `put` (meaning: put a character on the display).
- Prolog contains the usual arithmetic predicates

$$+, -, /, *, \ldots$$

  as well as the assignment predicate `is`.
- However, Prolog is not particularly suitable for more complicated numerical computations, since resolution is highly inefficient when it comes to arithmetical evaluations.

Cuts

- The cut predicate (!) is a controversial non-logical atom.
- The reason for this controversy is that cuts modify the SLD-tree used in search and, therefore, interfere with the basic principles of logic programming.
- In spite of that, cuts are very useful in Prolog since they can simplify the procedures and make computations more efficient.

Example

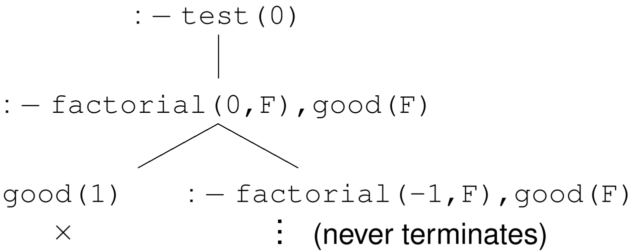
Consider the following procedure, `factorial(N,F)`, which computes the factorial `F` of a non-negative integer `F`.

```
factorial(0,1).
factorial(N,F) :-  N1 is N-1,
                   factorial(N1,F1),
                   F is N*F1.
```

Also, suppose that another procedure `good(N)` has been defined, which fails for `N`=1.
Then, consider the procedure `test(N)` :

```
test(N) :- factorial(N,F),good(F).
```

The query `:- test(0)` would result in the following SLD-tree



- ! cuts off all branches to the right of the one being currently examined and prevents unwanted backtracking and possible non-terminating branches.

```
factorial(0,1) :-  !.
factorial(N,F) :-  N1 is N-1,
                   factorial(N1,F1),
                   F is N*F1.
```

- It is possible to prove that any use of a cut in Prolog can be avoided by using logical predicates; for instance, our example can be modified in the following way:

```
factorial(0,1).
factorial(N,F):-  N>0,
                  N1 is N-1,
                  factorial(N1,F1),
                  F is N*F1.
```