Software Paradigms (Lesson 7)

Logic Programming & Software Engineering

Table of Contents

1 Goals (Queries).......................................................................................................................... 2
  1.1 Verifier Goal...................................................................................................................... 3
  1.2 Finder Goals ...................................................................................................................... 4
  1.3 Doer Goals...................................................................................................................... 5

2 Some Implementation Issues................................................................................................. 7
  2.1 The Backtracking Resolution Algorithm............................................................................ 7
  2.2 The Forward Resolution Algorithm .................................................................................. 9
  2.3 Discussion ...................................................................................................................... 10
1 Goals (Queries)

The term "executing programs" in logic programming typically refers to verification of one or a collection of predicates with respect to a given set of rules of inference and basic facts. This verification is done according to a specific algorithm which will be presented in this lesson. The term resolution mechanism is often used to refer to this algorithm.

Thus, a calculation in logic programming is identified by the following components:

- database of basic facts
- set of rules of inference
- a given goal

A computation in logic programming is initiated by a goal.

Example:

- **Database of basic facts**

  \[\text{locVO}(\text{Subject}, \text{Day}, \text{Room}):\]

<table>
<thead>
<tr>
<th>Subject</th>
<th>Day</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypermedia</td>
<td>Monday</td>
<td>Room 22</td>
</tr>
<tr>
<td>Databases</td>
<td>Tuesday</td>
<td>Room 23</td>
</tr>
<tr>
<td>Programming</td>
<td>Thursday</td>
<td>Room 22</td>
</tr>
</tbody>
</table>

- **EnrollVO(Student, Subject)**

<table>
<thead>
<tr>
<th>Student</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Palina</td>
<td>Hypermedia</td>
</tr>
<tr>
<td>Palina</td>
<td>Programming</td>
</tr>
<tr>
<td>Alex</td>
<td>Programming</td>
</tr>
<tr>
<td>Alex</td>
<td>Databases</td>
</tr>
</tbody>
</table>

- **Rules of inference:**

  \[\text{mustGo}(	ext{Student}, \text{Room}, \text{Day}) :-
  \exists \text{Subject} (\text{enrollVO}(	ext{Student}, \text{Subject}) \& \text{locVO}(	ext{Subject}, \text{Day}, \text{Room}))\]

- **Goal (Query):**

  \(? \text{mustGo}('\text{Alex}', 'Room 22', '\text{Thursday}')\)

The output will be success or failure, depending whether the validity of this predicate (fact) could be verified or not.

In this particular case, the calculation will produce a true value as a result.
In a general case we should distinguish between three different types of goals:

- **Verifier Goal**
- **Finder Goal**
- **Doer Goal**

### 1.1 Verifier Goal

A verifier goal is a predicate without free variables, i.e. it is a fact which can be true or false depending on a given set of basic facts and rules of inference.

For instance, given that we have facts and rules describing a genealogy tree,

<table>
<thead>
<tr>
<th>parent</th>
<th>G</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nick</td>
<td>Pauline</td>
<td></td>
</tr>
<tr>
<td>Pauline</td>
<td>Alex</td>
<td></td>
</tr>
<tr>
<td>Pauline</td>
<td>Paul</td>
<td></td>
</tr>
<tr>
<td>Alex</td>
<td>Tom</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{descendent}(G,D) :- \text{parent}(G,D)
\]

\[
\text{descendent}(G,D) :- \exists X (\text{parent}(G,X) \land \text{descendent}(X,D))
\]

we could issue the following verifier goal:

? \text{descendent}('Nick','Tom')

This verifier goal can be paraphrased as "Is Tom a descendent of Nick?" In this case, the engine would simply return yes or no depending on the predicates of the program.

Another verifier goal may look as follows:

? \exists X (\text{parent}(X,'Alex') \land \text{parent}(X,'Paul'))

The second verifier goal corresponds to the question "Are Alex and Paul siblings?". The last query is somewhat more complicated. It is composed of a conjunction of two goals which basically attempt to verify the following: "Is there a single person (X) who is a parent of Alex and parent of Paul?". Note, here we use variables to define a verifier goal, but this variable is bound with the existential quantifier.

Of course, goals are interpreted in a certain environment. Consider, for example, facts and rules describing a bill of materials,

<table>
<thead>
<tr>
<th>need</th>
<th>P</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D_A</td>
<td>D_B</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>D_B</td>
<td>D_D</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>D_B</td>
<td>D_C</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>D_D</td>
<td>D_E</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\text{need}+(P,C,Q) :- \text{need}(P,C,Q)
\]

\[
\text{need}+(P,C,(Q1*Q2)) :- \exists X (\text{need}(P,X,Q1) \land \text{need}+(X,C,Q2))
\]
In this environment, we could issue the following verifier goals:

? \exists X (\text{need+}('D_A','D_E',X))

This verifier goal can be paraphrased as "Do we need a component ‘D_E’ to assembly ‘D_A’?"

? \exists X \exists Y (\text{need+}(Y,'D_E',X) \& \text{greaterThan}(X,100))

“Are there any components where the component ‘D_E’ is reused in amount greater than 100?”

1.2 Finder Goals

In cases in which a goal contains unbound variables, such goal is called a finder goal. A finder goal requests the logic programming system to show particular values of free variables that make this goal. To clarify this last point, let's consider some sample queries.

Example “Students enrollment”

Basic Facts:

\text{enrollVO}(Student, Subject)
\text{locVO}(Subject, Day, Room))

Rules of Inference:

\text{mustGo}(Student, Room, Day) :-

\exists Subject \text{enrollVO}(Student, Subject) \& \text{locVO}(Subject, Day, Room))

Finder Goals:

? \text{mustGo} ('Alex',X,'Thursday')

The engine is requested to show all the values satisfying the \text{mustGo} ('Alex',X,'Thursday') predicate: X = \{Room 22\}

? \text{mustGo} ('Alex',X,Y)

The engine is requested to display all pairs of values (X,Y) such that the predicate \text{mustGo} ('Alex',X,Y) is valid

\{<X,Y>\} = \{<Room 23, Tuesday>,<Room 22, Thursday>\}

Example “genealogy tree”

<table>
<thead>
<tr>
<th>parent</th>
<th>G</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nick</td>
<td>Pauline</td>
</tr>
<tr>
<td></td>
<td>Pauline</td>
<td>Alex</td>
</tr>
<tr>
<td></td>
<td>Pauline</td>
<td>Paul</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>Tom</td>
</tr>
</tbody>
</table>

descendant(G,D):-parent(G,D)

descendant(G,D):- \exists X (parent(G,X) \& descendent(X,D))
Finder Goals:

? descendent('Nick', X)

In this case, the engine will show all values for the variable X that satisfy the descendent('Nick', X) predicate.

X = {Pauline, Alex, Paul, Tom}

? \exists X (parent(X, 'Alex') & parent(X, Y))

This goal can be paraphrased as "Get all siblings of a person “Alex”". This is interesting to note, despite the fact that we use two variables to define the goal – X and Y, the system is requested to show just values for the variable Y that satisfy the \( \exists X \ (parent(X, 'Alex') \land parent(X, Y)) \) predicate. Variable X is bound with the existential quantifier and we do not care which particular value of the variable X satisfy the conditions parent(X, 'Alex') and parent(X, Y)

Example “bill of materials”

<table>
<thead>
<tr>
<th>need</th>
<th>P</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D_A</td>
<td>D_B</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>D_B</td>
<td>D_D</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>D_B</td>
<td>D_C</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>D_D</td>
<td>D_E</td>
<td>4</td>
</tr>
</tbody>
</table>

need+(P,C,Q):-need(P,C,Q)

need+( P,C,(Q1*Q2)):- \exists X (need(P,X,Q1) \land need+(X,C,Q2))

Finder Goals:

? need+('D_B', X, Y)

Actually, we are asking systems to display list of materials needed to produce a component “D_B” and to provide additional information on quantity of the materials.

\(<X,Y> = \{<D_D, 3> , <D_C, 8> , <D_D, 12>\>\)

1.3 Doer Goals

Another kind of goals in logical programming is called a Doer Goal. As the name suggests, these statements are used to communicate specific information to modify environment for other goals.

In a simplest case, a doer goal simply adds/deletes basic facts. More complex cases of application of doer goals will be discussed below in the chapter “User interface and active rules of inference”.

Example “Students enrollment”

Basic Facts:

\[ \text{enrollVO(Student, Subject)} \]
\[ \text{locVO(Subject, Day, Room)} \]

<table>
<thead>
<tr>
<th>locVO</th>
<th>Subject</th>
<th>Day</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hypermedia</td>
<td>Monday</td>
<td>Room 22</td>
</tr>
<tr>
<td></td>
<td>Databases</td>
<td>Tuesday</td>
<td>Room 23</td>
</tr>
<tr>
<td></td>
<td>Programming</td>
<td>Thursday</td>
<td>Room 22</td>
</tr>
</tbody>
</table>

? delete(locVO('Programming', 'Thursday', 'Room 22'))

? add(locVO('Software Paradigms', 'Thursday', 'Room 21'))

<table>
<thead>
<tr>
<th>locVO</th>
<th>Subject</th>
<th>Day</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hypermedia</td>
<td>Monday</td>
<td>Room 22</td>
</tr>
<tr>
<td></td>
<td>Databases</td>
<td>Tuesday</td>
<td>Room 23</td>
</tr>
<tr>
<td></td>
<td>Software Paradigms</td>
<td>Thursday</td>
<td>Room 21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>enrollVO</th>
<th>Student</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Palina</td>
<td>Hypermedia</td>
</tr>
<tr>
<td></td>
<td>Palina</td>
<td>Programming</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>Programming</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>Databases</td>
</tr>
</tbody>
</table>

? add(enrollVO('Alex', 'Software Paradigms'))

? delete(enrollVO('Alex', 'Programming'))

? add(enrollVO('Palina', 'Software Paradigms'))

<table>
<thead>
<tr>
<th>enrollVO</th>
<th>Student</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Palina</td>
<td>Hypermedia</td>
</tr>
<tr>
<td></td>
<td>Palina</td>
<td>Programming</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>Databases</td>
</tr>
<tr>
<td></td>
<td>Alex</td>
<td>Software Paradigms</td>
</tr>
<tr>
<td></td>
<td>Palina</td>
<td>Software Paradigms</td>
</tr>
</tbody>
</table>
2 Some Implementation Issues

2.1 The Backtracking Resolution Algorithm

How do deductive DBMS infer a response to a given goal?

Given a program and an initial query, the resolution algorithm to verify this query can be described as follows.

Phase 1: A given goal is reduced to simple sub-goals corresponding to basic facts or procedures. The process is repeated until the goal is represented as a tree having only simple sub-goals corresponding to basic facts or procedures as leaves. Such tree is called a tree on inference or resolution tree (see example below).

Phase 2: Searching for values for (Verifying) the simple sub-goals and putting them together to form the response. This process is actually called a backtracking.

Example:

Basic Facts:
- B1(x) Database:{2, 3, 4}
- B21(x) Database:{1, 2, 3}
- B22(x) Database:{1, 3, 9}
- B3(x) Database:{1, 2, 5, 36}

Rules of Inference:
- P1(x) := B1(x) | P2 (x) & B3(x)
- P2(x) := B21(x) & B22(x)

Verifier Goal:
? P1(1)

Phase 1: The system built a resolution tree that, in this particular case, looks as two step process:

- Step 1.1:

- Step 1.2:
Phase 2: The system evaluates simple sub-goals and propagate values in a bottom-up (backtracking) fashion, in this particular case, this looks also as a two step process:

- **Step 2.1:**

  ![Diagram 1](image)

- **Step 2.2:**

  ![Diagram 2](image)

Finder goals are interpreted in a similar way.

Consider the Finder Goal: ? P1(X)

Phase 1: The system built a resolution tree as above:

![Diagram 3](image)

Phase 2: A particularly selected term is used to select a number of basic facts which define all potentially possible values of the free variables. Such selected values are called *hypothesis*.
Phase 3: All hypothesis are checked out as Verifier Goals to select valid values of free variables

2.2 The Forward Resolution Algorithm

Counterpart for the Backward Chaining Procedure is called Forward Chaining Procedure. The Forward Chaining Procedure deals with putting basic facts together to infer all possible new facts.

Example:

Basic Facts:
- B11(x) Database:{1, 2, 3, 4}
- B12(x) Database:{3, 4, 5, 6}
- B21(x) Database:{1, 2, 3, 4}
- B22(x) Database:{1, 2, 3, 5}

Rules of Inference:
- P(x) := P1(x) | P2(x)
- P1(x) := B11(x) & P12(x)
- P2(x) := B21(x) & B22(x)
Step 1: System infers all facts $P_1$;

**Rules of Inference:**
- $P_1(x) = P_1(x) \lor P_2(x)$
- $P_1(x) = B_{11}(x) \land B_{12}(x)$
- $P_2(x) = B_{21}(x) \land B_{22}(x)$

**Database**
- $B_{11}: (1, 2, 3, 4)$
- $B_{12}: (3, 4, 5, 6)$
- $B_{21}: (1, 2, 3, 4)$
- $B_{22}: (1, 2, 3, 5)$

Step 2: System infers all facts $P_2$;

**Rules of Inference:**
- $P_2(x) = P_1(x) \lor P_2(x)$
- $P_1(x) = B_{11}(x) \land B_{12}(x)$
- $P_2(x) = B_{21}(x) \land B_{22}(x)$

**Database**
- $B_{11}: (1, 2, 3, 4)$
- $B_{12}: (3, 4, 5, 6)$
- $B_{21}: (1, 2, 3, 4)$
- $B_{22}: (1, 2, 3, 5)$

Step 3: System infers all facts $P$;

**Rules of Inference:**
- $P(x) = P_1(x) \lor P_2(x)$
- $P_1(x) = B_{11}(x) \land B_{12}(x)$
- $P_2(x) = B_{21}(x) \land B_{22}(x)$

**Database**
- $B_{11}: (1, 2, 3, 4)$
- $B_{12}: (3, 4, 5, 6)$
- $B_{21}: (1, 2, 3, 4)$
- $B_{22}: (1, 2, 3, 5)$

Note that the Backward Chaining Procedure needs to be invoked every time when the user issues a *Verifier or Finder Goal*. The response is inferred dynamically and no inferred data are stored into the database.

On the contrary, the Forward Chaining Procedure needs to be invoked only when the database is modified (i.e. when the user issues a *Doer Goal*). All facts defined by means of rules of inference are inferred and stored into the database. Thus, forthcoming Verifier or Finder Goals can be interpreted by means of searching such previously stored inferred fact.

Since, interpreting a Finder goal may take considerable time, the Backward Chaining Procedure is normally applied for database systems which are not critical to response time.

Deductive systems which are supposed to control processes in real time (say, an oil refinery process), normally supports Forward Chaining Procedure to minimize a response time.

### 2.3 Discussion

We can view the task of executing a logic program as a search problem. The representation of goals in terms of a tree, and the algorithm itself support this analogy. Hence, the objective of the
algorithm is to find a solution if one exists in the search space. The depth-first nature and the
backtracking of the algorithm introduced are simply in this context computational features to deal
with the non-determinism of the task at hand. In theory, any other (correct) means of dealing with
such non-determinism could be used. For instance, the search of the tree could be done breadth-
first manner instead. Since predicates can be recursive, breadth-first search would have the
advantage of always finding a solution whenever one exists. On the other hand, a depth-first
search of the same tree might not terminate, when for instance recursive clauses are attempted
first.

Since breadth-first search seems to avoid this problem of infinite recursion, one might wonder
why it is not used for searching at the first place. The answer is due to efficiency considerations.
Much more information would have to be kept at runtime (in fact all the information of the tree
up to that point) with breadth-first search. Depth-first search can be implemented much more
efficiently in this respect. But, as is often the case, the gain in efficiency comes at a cost. In this
case, the cost is considerable in that it forces programmers to be careful and aware of how
programs are executed and arrange their programs accordingly. The order of in which clauses of a
predicate appear will matter (depth-first). In particular, the possibility of infinite recursion must
be avoided either by placing at least one non-recursive clause before the recursive one(s), or by
ensuring that the head of the recursive clause(s) will at some point not unify with a goal.