

Lecture 15: Logic Programming

10:30 AM, Mar 17, 2009

Contents

1	Overview	1
2	Logic Programming	2
2.1	Facts	2
2.2	Rules	3
2.3	Queries	4
3	Prolog	4
3.1	Syntax	4
3.2	SLD Resolution	5
3.3	Lists	5
3.4	Recursion	5
3.5	is	5
3.6	Cut	6
3.7	Negation	7
3.8	Flow of Control	7
	3.8.1 Clause Order	7
	3.8.2 Goal Order	8
4	Documentation	8

1 Overview

Logic programs are an axiomatization of a problem space. This axiomatization entails a set of logical consequences, which gives meaning to the logic program and solves the problem.

Rather than being derived from the von Neumann machine model and instruction set by a series of abstractions and reorganizations, [logic programming] is derived from an abstract model, which has no direct relation to or dependence on one machine model or another.¹

¹These lecture notes, which were compiled by Keith Hall circa 2002, are primarily based on Sterling and Shapiro [1].

2 Logic Programming

Before delving into the details of the Prolog programming language (one specific logic programming implementation), we define the general terminology and conceptual tools of logic programming.

A logic program is a knowledge base of **facts** and **rules**.

2.1 Facts

A **predicate** describes a relation among objects in a set. In the simplest case, the objects in the set are referred to by **constants**, and a predicate expresses a relationship among objects by relating constants. Some simple examples of the use of predicates and constants in logic programming follow:

```
plus(0,0,0).
plus(2,3,5).
plus(10,20,30).
```

```
older(michael,amy).
older(michael,merrie).
older(amy,merrie).
```

```
mother(anne,keith).
mother(piggy,gonzo).
mother(barbara,georgejr).
```

```
father(henry,keith).
father(patrick,henry).
father(georgesr,georgejr).
```

In this sample logic program, **plus**, **older**, **mother**, and **father** are predicates; and 0, 2, 3, 5, 10, 20, 30, etc. are constants. Together predicates and constants (or, more generally, terms) comprise atomic formulas, which are called **goals** in logic programming. A goal followed by a period is a **fact**. We read the facts in the above logic program as “2 plus 3 is 5;” “michael is older than merrie;” “piggy is the mother of gonzo;” and so on.

A logic program embodies a knowledge base as well as a specific interpretation. The terms of a logic program define its domain, and the predicates give it an interpretation. The domain D of the sample program is given by:

$$D = \{0, 2, 3, 5, 10, 20, 30, \text{michael}, \text{amy}, \text{merrie}, \\ \text{anne}, \text{keith}, \text{piggy}, \text{gonzo}, \text{patrick}, \text{henry}, \\ \text{barbara}, \text{georgesr}, \text{georgejr}, \text{TRUE}, \text{FALSE}\}$$

The mapping M describes the interpretation of the predicates:

$$\begin{aligned} \text{plus}^M &= \{(0, 0, 0), (0, 1, 1), (1, 0, 1), (0, 2, 2), (2, 0, 2), (0, 3, 3), (3, 0, 3), \dots\} \\ \text{older}^M &= \{(\text{michael}, \text{amy}), (\text{michael}, \text{merrie}), (\text{amy}, \text{merrie}), \dots\} \\ \text{mother}^M &= \{(\text{anne}, \text{keith}), (\text{piggy}, \text{gonzo}), (\text{barbara}, \text{georgejr})\} \\ \text{father}^M &= \{(\text{henry}, \text{keith}), (\text{patrick}, \text{henry}), (\text{georgesr}, \text{georgejr})\} \end{aligned}$$

The domain of a logic program always includes the atoms `TRUE` and `FALSE`. Facts are truths: the meaning of a fact in any domain of interpretation is `TRUE`. The sample logic program defines the truths “Henry is the father of Keith;” “Patrick is the father of Henry;” “George senior is the father of George junior;” and so on.

2.2 Rules

A **rule** in a logic program is a predicate defined in terms of other predicates. For example,

$$\text{parent}(X, Y) \leftarrow \text{father}(X, Y).$$

is a rule that defines the `parent` predicate in terms of the `father` predicate. This rule reads “ X is a parent of Y just in case X is the father of Y .” In this context, X and Y are variables, or placeholders, as in first-order logic. By convention, variables begin with capital letters and constants begin with lowercase letters.

Rules in a logic program are expressed as Horn clauses in normal form:

$$\text{newrule}(X_1, \dots, X_n) \leftarrow \text{oldrule1}(Y_1, \dots, Y_m), \dots, \text{oldruleK}(Z_1, \dots, Z_l).$$

The goal on the left hand side of \leftarrow is called the **head** of a rule and the goals on the right hand side of \leftarrow constitute the **body** of a rule. Commas between goals indicate conjunction.

In order to define a disjunctive predicate, additional rules are necessary: *e.g.*,

$$\begin{aligned} \text{parent}(X, Y) &\leftarrow \text{mother}(X, Y). \\ \text{parent}(X, Y) &\leftarrow \text{father}(X, Y). \end{aligned}$$

Together these rules state that “ X is a parent of Y just in case X is the mother of Y *or* X is the father of Y .” A set of rules with the same predicate on the left hand side of \leftarrow is called a **procedure**.

By convention, all variables in a logic program are universally quantified, except for those variables that appears only in the body of a rule, which can be understood to be existentially quantified. For example, the rule

$$\text{grandson}(X, Y) \leftarrow \text{parent}(Y, Z), \text{parent}(Z, X), \text{male}(X).$$

expresses all of the following the (equivalent) logical formulas:

$$\begin{aligned} &\forall X, Y, Z (\text{parent}(Y, Z) \wedge \text{parent}(Z, X) \wedge \text{male}(X)) \rightarrow \text{grandson}(X, Y) \\ \text{iff } &\forall X, Y, Z \neg(\text{parent}(Y, Z) \wedge \text{parent}(Z, X) \wedge \text{male}(X)) \vee \text{grandson}(X, Y) \\ \text{iff } &\forall X, Y, Z (\neg\text{parent}(Y, Z) \vee \neg\text{parent}(Z, X) \vee \neg\text{male}(X)) \vee \text{grandson}(X, Y) \\ \text{iff } &\forall X, Y \neg(\exists Z \neg(\neg\text{parent}(Y, Z) \vee \neg\text{parent}(Z, X) \vee \neg\text{male}(X))) \vee \text{grandson}(X, Y) \\ \text{iff } &\forall X, Y \neg(\exists Z \text{parent}(Y, Z) \wedge \text{parent}(Z, X) \wedge \text{male}(X)) \vee \text{grandson}(X, Y) \\ \text{iff } &\forall X, Y \exists Z (\text{parent}(Y, Z) \wedge \text{parent}(Z, X) \wedge \text{male}(X)) \rightarrow \text{grandson}(X, Y) \end{aligned}$$

2.3 Queries

A **query** is a goal followed by a question mark. It retrieves information from a logic program, by asking the question: is the goal true: *i.e.*, is the goal a logical consequence of the program? Given the logic program of Section 2.1, the queries

```
father(henry,keith) ?
father(patrick,keith) ?
```

are interpreted as TRUE and FALSE, respectively.

Conjunctive queries evaluate a logical conjunction of goals. For example,

```
father(henry,keith), mother(anne,keith) ?
```

is a conjunctive query. This query evaluates to TRUE iff both goals are true in the context of the current logic program.

Variables in queries are existentially quantified. Thus, the query

```
mother(X,keith) ?
```

asks the question “does there exist an object that is the mother of keith?” Logic programs derive all such X that satisfy the relation described by the query.

Why is it that variables in logic programming queries are existentially quantified? The answer is that these variables appear only in the body of a procedure. In particular, the query

```
mother(X,keith) ?
```

is an abbreviation for

```
FALSE ← mother(X,keith)
```

3 Prolog

Prolog was developed by Alain Comerauer and his group at the University of Marseilles-Aix in 1973. It was originally written in Fortran. The name Prolog is an abbreviation for *Programmation en Logique*. But no Prolog implementation is yet to achieve the ideals of logic programming. To minimize space complexity, Prolog generates search trees in a depth-first manner; thus, it is not complete. To minimize time complexity, Prolog skips occurs checks; thus, it is not sound.

3.1 Syntax

The syntax of Prolog looks much like the logic programming syntax presented in the first half of this lecture, except that Prolog uses the digraph $:-$ to represent logical implication, rather than \leftarrow . Here is some additional Prolog terminology: a **clause** is a rule with n goals in the **body**; a **unit clause** is a clause *s.t.* $n = 0$: *i.e.*, a unit clause is a fact; an **iterative clause** is a clause *s.t.* $n = 1$. A **program** is a set of clauses.

3.2 SLD Resolution

Clause selection in Prolog is performed in a depth-first backtracking SLD fashion. SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite (i.e., Horn) clauses. In performing resolution, a Prolog interpreter evaluates the first goal in the body of a rule, resolves that rule and evaluates the first goal of the new rule. This depth first process proceeds until a bound literal is found or a goal is unable to be proven. When a goal fails, the Prolog interpreter backs up to the last point where a decision was made. This will be that last point a variable was assigned. If additional assignments are possible, the variable is reassigned and resolution is repeated with the new variable assignment.

3.3 Lists

Lists are a built-in Prolog data structure, with multiple syntactic variations. The canonical representation of a list is simply $.(H, T)$, where H is the head of the list and T is the tail. A more convenient syntax is the element syntax denoted $[H|T]$. In this syntax, the empty list is denoted by the symbol $[]$. The three columns in the following table present three alternative Prolog representations of lists. The third one is the most common because it is the easiest to read.

$.(a, [])$	$[a []]$	$[a]$
$.(a, (b, []))$	$[a [b []]]$	$[a, b]$
$.(a, (b, X))$	$[a [b X]]$	$[a, b X]$

3.4 Recursion

Recursion is the primary means of flow control in logic programs. The following example of recursive programming in Prolog implements arithmetic operations. By definition, a natural number is either 0, or the successor function $s()$ applied to a natural number. For example, the number 1 is represented by $s(0)$. The following Prolog code defines natural numbers, addition, and multiplication. The three predicates, `natural_number`, `plus`, and `times` are all defined recursively.

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).

plus(0,X,X) :- natural_number(X).
plus(s(X),Y,s(Z)):- plus(X,Y,Z).

times(0,X,0) :- natural_number(0).
times(s(X),Y,Z) :- times(X,Y,XY), plus(XY,Y,Z).
```

3.5 is

Prolog provides standard arithmetic operators as well as an assignment operator `is(Value, Expression)`. The `is` operator is true if the result of `Value` unifies with `Expression`. `Expression` must have an arithmetic value.

```
?- X is 4 + 3 * 2.
?- A is -1, X is A + 1.
?- X is A + 1, A is -1.
```

The first query is true: set $X = 10$. In the second query, the variable A is bound to -1 prior to being its evaluation in the expression $A + 1$; thus, this query is also true: set $X = 0$. In the third query, however, the variable A is not bound at the time the `is` predicate is evaluated. Prolog produces an error if an **Expression** without an arithmetic value is passed to the `is` predicate.

3.6 Cut

The `cut` predicate, denoted `!`, provides logic programmers with an explicit means of flow control. In particular, the cut predicate prevents backtracking.

Given a Prolog clause, say `p(X) :- q(X), !, r(X).`, if `q(X)` succeeds, and the cut predicate is encountered, then:

- the cut predicate always succeeds
- if `r(X)` succeeds, then `p(X)` succeeds
- if `r(X)` fails, then `p(X)` fails

Consider the following program, which succeeds if its third argument contains the minimum of its first two arguments:

```
minimum(X,Y,X) :- X < Y, !.
minimum(X,Y,X) :- X is Y, !.
minimum(X,Y,Y) :- X > Y, !.
```

This program has the same semantics as the `minimum` program without cuts. But the first cut prevents evaluation of the second clause once it is determined that $X < Y$, and the second cut prevents evaluation of the third clause once it is determined that $X \leq Y$. The following example presents an incorrect use of the cut predicate, since the query `error(0,1,1)` succeeds in this program:

```
error(X,Y,X) :- X < Y, !.
error(X,Y,X) :- X is Y, !.
error(X,Y,Y).
```

If the use of the `cut` predicate does not alter the meaning of the program then it is called a **green** cut. Green cuts improve efficiency. Cuts that are not green are called **red** cuts. Red cuts can be used to implement an if-then-else statement:

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

Red cuts are also used to avoid backtracking, if only a single solution is desired. The following version of `member` succeeds when the first occurrence of X appears in the list, but it does not backtrack to search for additional occurrences of X like the cut-free version (see Section 3.8):

```
member(X, [X|Xs]) :- !.
member(X, [_|Ys]) :- member(X, Ys).
```

3.7 Negation

If a query fails, then the goal is not a logical consequence of a program. Failure does *not* imply that the negated form of the goal *is* a logical consequence of the program. Prolog makes use of a closed-world-assumption, however, under which this implication holds. Interpreting the lack of a proof a formula as proof of its negation is called **negation as finite failure**.

Negation is implemented in Prolog by the `fail` predicate, which always fails. Using `!` and `fail`, negation is achieved as follows:

```
not_f(X) :- f(X), !, fail.
not_f(X).
```

This program succeeds only if `f(X)` fails. The second clause, which always succeeds, is evaluated only if `f(X)` fails for all X . Some Prolog implementations include a built-in predicate `not`, which provides the same functionality.

3.8 Flow of Control

Prolog evaluates clauses in a procedure from top to bottom, and goals in a clause from left to right. Efficient Prolog programs exploit these orderings.

3.8.1 Clause Order

The following Prolog program defines the `member` relation.

```
member(X, [X|Xs]).
member(X, [_|Ys]) :- member(X, Ys).
```

In this implementation, the query `member(a, [a,b,c,d,a]) ?` unifies with the first rule and succeeds immediately. But consider the following implementation of membership in which the clause ordering is reversed.

```
member(X, [_|Ys]) :- member(X, Ys).
member(X, [X|Xs]).
```

The same query, `member(a, [a,b,c,d,a]) ?`, does not succeed until the entire list has been processed, and the final `a` is discovered in the unrolling of the recursion stack.

```

member(a, [a,b,c,d,a]) ?
member(a, [a|b,c,d,a]) :- member(a, [b,c,d,a]).
member(a, [b|c,d,a]) :- member(a, [c,d,a]).
member(a, [c|d,a]) :- member(a, [d,a]).
member(a, [d|a]) :- member(a, [a]).
member(a, [a]) :- member(a, []).
member(a, [a]).

```

The second to last statement in the above computation fails. After backtracking, the final statement resolves with the second rule.

3.8.2 Goal Order

One definition of the son relation is presented below, in two ways, using two competing goal orderings. The first rule defines X to be a son of Y if X is a child of Y and X is male. The second rule defines X to be a son of Y if X is male and X is a child of Y .

```

son(X,Y) :- child(X,Y), male(X).
son(X,Y) :- male(X), child(X,Y).

```

Typically, the set of males is larger than the set of children of any parent. Thus, the first goal ordering, which asks only whether the children of Y are male, is more efficient than the second, which asks whether any male is a child of Y .

4 Documentation

Documentation for the SICStus Prolog system is available at:

<http://www.sics.se/isl/sicstuswww/site/documentation.html>

The current (as of February 2002) version of SICStus Prolog on the Brown CS server is 3.8.5. Documentation for this version can be found at:

<http://www.sics.se/isl/sicstus/docs/3.8.5/html/sicstus.html>

References

- [1] Sterling, Leon & Shapiro, Ehud (1994) The Art of Prolog, MIT Press.