Chapter 16

Logic Programming

Topics

- Summary (resolution, unification, Prolog search strategy)
- Disjoint goals
- The “cut” operator
- Negative goals
- Predicate “fail”
- Debugger / tracer
- Arithmetic
- Lists

Resolution

- Resolution is an inference rule for Horn clauses.
- Given two clauses:
  - If the head of the first clause can be matched with one of the statements in the body of the second clause then the first clause can be used to replace its head in the second clause by its body.

$$a \leftarrow a_1, \dots, a_n.$$
$$b \leftarrow b_1, \dots, b_i, \dots, b_m.$$

and b_i matches a, then

$$b \leftarrow b_1, \dots, b_{i-1}, a_1, \dots, a_n, b_{i+1}, \dots, b_m$$

Unification

- Unification is the process by which variables are instantiated so that patterns match during resolution.
- Unification is the process of making two terms “the same” in some sense.
  - `‘foo’ = foo`
    - Prolog’s answer: `yes`.
    - Both terms are atoms.
  - `‘5’ = 5`
    - Prolog’s answer: no.
    - LHS is an atom and RHS is a number.

Prolog’s Search Strategy

- Prolog applies resolution in a strictly linear fashion
  - Replaces goals left to right.
  - Considers clauses in top-to-bottom order.
  - Subgoals are considered immediately once they are set up.
  - Search can be viewed as a depth-first search on a tree of possible choices.

Prolog’s Search Strategy

- Given the following clauses:

```
(1) ancestor(X,Y) :- parent(X,Z),
                     ancestor(Z,Y).
(2) ancestor(X,X).
(3) parent(amy,bob).
```

- Given the goal `ancestor(X,bob)`, Prolog’s search strategy is left to right and depth first on the following tree of subgoals.
  - Edges are labelled by the number of the clause used by Prolog for resolution
  - Instantiation of variables are written in curly brackets.

Prolog's Search Strategy

- Leaf nodes in this tree occur
  - No match is found for the leftmost clause.
  - All clauses have been eliminated (success).
- Whenever failure occurs Prolog backtracks up the tree to find further paths to a leaf, releasing instantiations of variables.

Prolog's Search Strategy

- Original Prolog program:

```
(1) ancestor(X,Y) :- parent(X,Z),
                     ancestor(Z,Y).
(2) ancestor(X,X).
(3) parent(amy,bob).
```

- Clauses in slightly different order:

```
(1) ancestor(X,Y) :- ancestor(Z,Y),
                     parent(X,Z).
(2) ancestor(X,X).
(3) parent(amy,bob).
```

- Problem?

Existential Queries

- Variables in queries are existentially quantified.

```
father(abraham,X)?
```

  - Reads: "Does there exist an `X` such that `abraham` is the father of `X`?"
  - For convenience, existential quantification is omitted.

Universal Facts

- Variables in facts are implicitly universally quantified.
- In general, a fact $p(T_1,\ldots T_n)$ reads that for all $X_1,\ldots,X_k$ where the X_i are variables occurring in the fact $p(T_1,\ldots T_n)$ is true.

```
likes(X,apple).
```

- From a universally quantified fact one can deduce any instance of it.

```
likes(adan,apple).
```

"Universal" Rules

- Rule specifies things that are true if some condition is satisfied.

For all X and Y,
Y is an offspring of X if
X is a parent of Y.

```
?-offspring(Y,X):-parent(X,Y).
```

Disjoint Goals ("or")

- Prolog provides a semicolon, meaning "or".
  - The definition of `parent` could be written as a single rule:

```
parent(X,Y) :- father(X,Y); mother(X,Y).
```

- The normal way to express an "or" relation in Prolog is to state two rules.
  - The semicolon adds little or no expressive power to the language.
  - It looks so much like the comma that it often leads to typographical errors.
  - The use of semicolon is not recommended.

The "Cut" Operator (!)

- Automatic backtracking is one of the most characteristic features of Prolog.
  - Lead to inefficiency: explore possibilities that lead nowhere.
- The cut predicate tells the Prolog system to forget about some of the backtrack points.
  - Discards all backtrack points that have been recorded since execution entered the current clause.

The "Cut" Operator (!)

- After executing a cut:
  - It is no longer possible to try other clauses as alternatives to the current clause.
  - It is no longer possible to try alternative solutions to subgoals preceding the cut in the current clause.
- Reduces the search space.
  - "do not go to" (alternatives that we know are bound to fail).

```
writename(1) :- !, write('One').
```

- Confirms the choice of a rule.

```
max(X,Y,Y) :- X>=Y, !.
max(X,Y,X).
```

The "Cut" Operator (!)

- Simulates an "else" statement when we are testing cases.
  - Mutually exclusive conclusions .

```
f(X,0) :- X=0, !.
f(X,1) :- X>0, !.
f(X,undefined).
```

- Example: Given the knowledge base

```
f(X) :- g(X), !, h(X).
f(X) :- j(X).
g(a).
j(a).
```

What is the result of executing the query `?- f(a).` ?

Negative Goals ("not)

- The special predicate `\+` means "not" or "cannot prove".
- If g is any goal, then `\+ g` succeeds if g fails, and fails if g succeeds.

```
?- \+ likes(adan,apple).
```

- The predicate `\+` can appear only in a query or on the right-hand side of a rule.
  - It cannot appear in a fact or in the head of a rule.

```
\+ likes(adan,apple).
```

Negation as Failure

- The behaviour of `\+` is called *negation as failure*.
  - In Prolog, you cannot state a negative fact . All you can do is conclude a negative statement if you cannot conclude the corresponding positive statement.
    - What is the definition a a person who is not a parent?

```
non_parent(X,Y) :- \+ father(X,Y),
                   \+ mother(X,Y).
```

Closed-World Assumption

- What happens if you ask about people who are not in the knowledge base?
- The Prolog systems assumes that its knowledge base is complete (this is called the closed-world assumption).
  - Something that cannot be proved to be true is assumed to be false.

Predicate “fail”

- Predicate *fail* is a special symbol that will immediately fail when Prolog encounters it as a goal.
- *Cut-fail* combination is used to say that something is not true.

```
    likes(mary,X) :- snake(X), !, fail
    likes(mary,X) :- animal(X).
```

Debugger / Tracer

- The debugger allows you to trace exactly what is happening as Prolog executes a query.

```
?- trace.
yes
```

  - *Return*: computation is shown step by step.
  - *s* (for “skip”): the debugger will skip to the end of the current query (useful if the current query has a lot of subgoals which you do not want to see).
  - *a* (for “abort”): the computation will stop.

Arithmetic

- Prolog supports both integers and floating-point numbers and interconvert them as needed.
- Operator “**is**”: takes an arithmetic expression on its right, evaluates it, and unifies the result with its argument on the left.

```
?- Y is 2+2.          ?- 5 is 3+3.
Y = 4
yes                    no
?- Z is 4.5+(3.9/2.1).
Z = 6.3571428
yes
```

Arithmetic

- The precedence of operators is about the same as in other programming languages.
- Prolog is not an equation solver.
  - Prolog does not solve for unknowns on the right hand side of is:

```
?- 5 is 2 + What.
instantiation error.
```

Constructing Expressions

- Prolog vs. other programming languages
  - Other programming languages evaluate arithmetic expressions wherever they occur.
  - Prolog evaluates arithmetic expressions only in specific places.
    - 2+2 evaluates to 4 only when it is an argument of the predicates of the following table; the rest of the time it is just a data structure consisting of 2, +, and 2.

Built-in Predicates that Evaluate Expressions

`R is Expr`	Evaluates `Expr` and unifies result with `R`
`Expr1 =:= Expr2`	Succeeds if results of both expressions are equal.
`Expr1 =\= Expr2`	Succeeds if results of the expressions are not equal.
`Expr1 > Expr2`	Succeeds if `Expr1 > Expr2`
`Expr1 < Expr2`	Succeeds if `Expr1 < Expr2`
`Expr1 >= Expr2`	Succeeds if `Expr1 >= Expr2`
`Expr1 =< Expr2`	Succeeds if `Expr1 =< Expr2`

Constructing Expressions

- There is a clear difference between:
  - `is`, which takes an expression (on the right), evaluates it, and unifies the result with its argument on the left.
  - `=:=`, which evaluates two expressions and compares the results.
  - `=`, which unifies two terms (which need not be expressions and, if expressions, will not be evaluated).

Constructing Expressions: examples

```
?- What is 2+3.
What = 5        % Evaluates 2+3, unify result with What

?- 4+1 =:= 2+3.
yes             % Evaluates 4+1 and 2+3, compare results

?- What = 2+3
What = 2+3      % Unify What with the expression 2+3
```