Introduction to AI with Prolog

Matching and Proof Search

Alexey Sery

a.seryj@g.nsu.ru

Department of Information Technologies
Novosibirsk State University

September 1, 2021

Overview

Goals of the lesson

- Discuss the idea of matching in Prolog.
- Explain the differences between Prolog matching and standard unification.
- Introduce the built-in predicate for matching.
- Explain the search strategy Prolog uses when trying to prove something.

Matching

- The basic idea of matching is follows: **Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal**.
- The built-in binary predicate =/2 tests whether its two arguments match.

The following terms match:

```
orange = orange.
fruit(orange) = fruit(orange).
cat(C) = cat(fluffy).
cat(C) = cat(cornie).
f(g(X,Y)) = f(g(x, Z)).
f(g(X,Y)) = f(g(Z,y)).
f(g(X,Y)) = f(g(a,b)).
```

Matching

1. If `term1` and `term2` are constants, then `term1` and `term2` match if and only if they are the same atom, the same string or the same number.
2. If `term1` is a variable and `term2` is any type of term, then `term1` and `term2` match, and `term1` is instantiated to `term2`. Vice versa is also true. If they are both variables, they're both instantiated to each other, and we say that they **share values**.
3. If `term1` and `term2` are complex terms, then they match if and only if:
   3.1 They have the same functor and arity.
   3.2 All their corresponding arguments match.
   3.3 The variable instantiations are compatible.
4. Two terms match if and only if it follows from the previous three clauses that they match.

Matching

Let us have some examples:

`someString = someString.`	`true.`
`100 = 100.`	`true.`
`someString = "someString".`	`false.`
`someString = 'someString'.`	`true.`
`100 = '100'`	`false.`
`Var1 = Var2`	`true.`
`triang(p(1,1),p(Px,Py),p(Px1,8)) = triang(P,p(7,4),p(n(7),8)).`	`true.`
`triang(p(X,X),p(2,5),p(7,11)) = triang(p(1,7), P2, P3).`	`false.`
`f(X) = X.`	`true?`

Unification

- Terms are symbolic expressions used to model logical propositions.
- Terms can easily be represented as trees, where variables and constants are leaves and functors are branches.
- A substitution of terms is a set of variables paired with terms: $\{(x_1 \to t_1), \ldots, (x_n \to t_n)\}$, where each pair represents a variable x_i, that should be substituted with a term t_i.
- **Unification** is the process of unifying equations, called terms, so that they become equivalent. This is done by finding a substitution which when applied on the variables of the terms will result in them becoming identical.
- A unification algorithm commonly takes 2 terms and returns this substitution if it exists. The substitution that unifies the terms is called **a unifier**.

Unification

- A unification algorithm performs **the occur check**.
- The occur check checks whether a variable appears among the arguments of the functor, which it is being unified with. This is required to prohibit infinite terms like $X = f(X)$, which results in something like $f(f(f(f(\ldots))))$. This can create cycles that could cause termination problems.

Suppose we have the following terms. What will the Standard Unification Algorithm do?

$$f(X_1, X_2, \ldots, X_n)$$

$$f(g(X_0, X_0), g(X_1, X_1), \ldots, g(X_{n-1}, X_{n-1}))$$

Unification

$$f(X_1, X_2, \ldots, X_n)$$

$$f(g(X_0, X_0), g(X_1, X_1), \ldots, g(X_{n-1}, X_{n-1}))$$

- Obviously enough, the algorithm will reduce the problem by recognizing that the two functors are identical.
- Then problem becomes to transform the variable input of the two functors such that they become identical. Hence the algorithm will start unifying the variables of f.
- First X_1 will be substituted with $g(X_0, X_0)$. The rest of the list will then replace each instance of X_1 with $g(X_0, X_0)$.
- The latter means that X_2 will be substituted with $g(g(X_0, X_0), g(X_0, X_0)$ and so on.

Unification

$$f(X_1, X_2, \ldots, X_n)$$

$$f(g(X_0, X_0), g(X_1, X_1), \ldots, g(X_{n-1}, X_{n-1}))$$

$X_1 \rightarrow g(X_0, X_0)$

$X_2 \rightarrow g(g(X_0, X_0), g(X_0, X_0))$

$X_3 \rightarrow g(g(g(X_0, X_0), g(X_0, X_0)), g(g(X_0, X_0), g(X_0, X_0)))$

$\vdots$

$X_n \rightarrow g(g(g(g(g \ldots$

Unification

- Prolog **omits** the occur check when matching terms since the running time of the occur check can escalate.
- Standard unification algorithms are **pessimistic**. They first look for strange variables (using the occurs check) and only when they are sure that the two terms are "safe" do they go ahead and try and match them. So a standard unification algorithm will never get locked into a situation where it is endlessly trying to match two unmatchable terms.
- Prolog, on the other hand, is **optimistic**. It assumes that you are not going to give it anything dangerous. So it does not make an occurs check. As soon as you give it two terms, it charges full steam ahead and tries to match them.

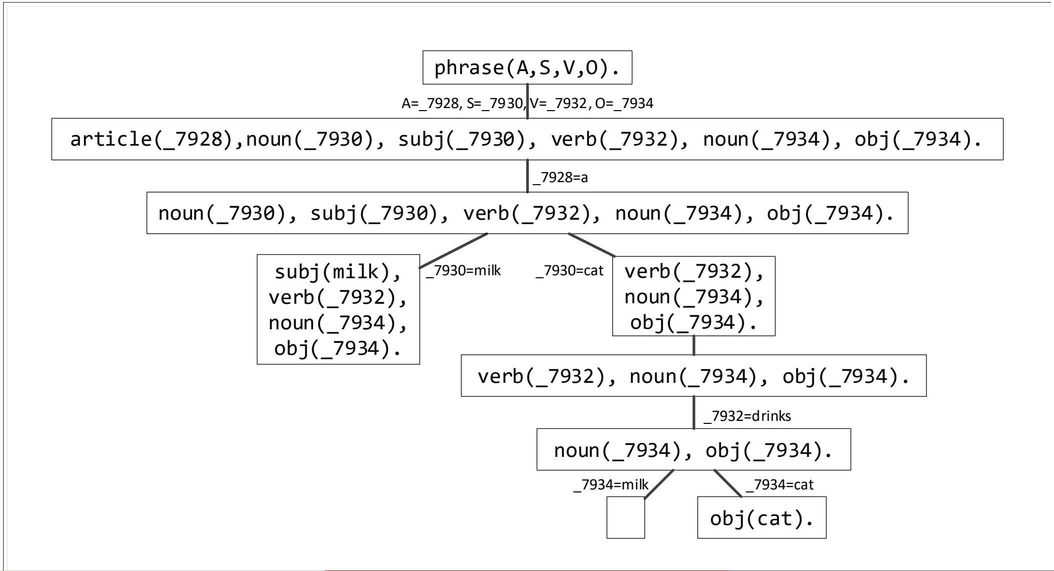
Proof search

Suppose we are working with the following knowledge base:

```
verb(drinks).
noun(milk).
noun(cat).
article(a).
subj(cat).
obj(milk).

phrase(A,S,V,O) :- article(A),
noun(S), subj(S), verb(V), noun(O),
obj(O).
```

```
Suppose we then pose a
query:  phrase(A,S,V,O).

The answer will be

A = a,
S = cat,
V = drinks,
O = milk
```

Proof search


phrase(A,S,V,O).
A=_7928, S=_7930, V=_7932, O=_7934
article(_7928),noun(_7930), subj(_7930), verb(_7932), noun(_7934), obj(_7934).
_7928=a
noun(_7930), subj(_7930), verb(_7932), noun(_7934), obj(_7934).
_7930=milk
_7930=cat
subj(milk), verb(_7932), noun(_7934), obj(_7934).
verb(_7932), noun(_7934), obj(_7934).
verb(_7932), noun(_7934), obj(_7934).
_7932=drinks
noun(_7934), obj(_7934).
_7934=milk
_7934=cat
obj(cat).