Programming Language Theory
Lecture notes

4 An execution model for Prolog

Prolog is a programming language based on predicate logic.

4.1 Introduction

A Prolog program consists of *axioms* and *rules*. Examples of axioms are

```
male(gustav).
feamale(stina).
feamale(eva).
feamale(lena).
father(gustav, eva).
father(gustav, lena).
father(oskar, gustav).
mother(stina, eva).
wife(gustav, stina).
```

The intended meanings of theses axioms are obvious; `gustav` is a male while `stina, eva` and `lena` are females. `oskar` is the father of `gustav` who is the father of `eva` and `lena`. In Prolog variables starts with a capital letter while other strings are constants. Writing "`male(Gustav).`" will in fact mean that anything is a male.

A rule is an inference rule that can be used to deduce information from axioms and rules. With the above axioms the following rules are appropriate.

```
husband(X, Y) :- wife(Y, X).
male(X) :- father(X, Y).
son(X, Y) :- male(X), father(Y, X).
sonOfGustav(X) :- male(X), father(gustav, X).
grandfather(X, Z) :- father(X, Y), father(Y, Z).
```

These rules are to be read as

- If `X` is the wife of `Y` then `Y` is the husband of `X`.
- If `X` is the father of somebody then `X` is male.
- `X` is the son of `Y` if `X` is male and `Y` is the father of `X`.
- `X` is a son of Gustav if `X` is male and `gustav` is the father of `X`,
- If `X` is the father of `Y` and `Y` is the father of `Z` then `X` is the grandfather of `Z`.

Given a program, *queries* can be made. A query is preceded by `?-` and the runtime system will answer `yes` if the query can be deduced from the rules and `no` if it cannot.

```
?- feamale(eva).
yes
?- feamale(anna).
no
?- male(oskar).
yes
```

A query may contain variables and the runtime system will list all values that make the query deduceable.

```
?- grandfather(oskar, Y).
Y=eva
Y=lena
?- male(X).
X=gustav
X=oskar
```

Integers are of course implemented in Prolog using the representation provided by the platform. We may, however, implement natural numbers from scratch.

```
number(0).
number(suc(N)) :- number(N).
plus(N, 0, N) :- number(N).
plus(N, suc(M), suc(NpM) ) :- plus(N, M, NpM).
```

The intended meaning of `plus(N, M, S)` is that `S=N+M`. (`number` is a predefined predicate in gprolog so you may need to rename it if you try the example.)

This program can be used to add numbers but also to compute the difference of two numbers and to find all solutions to a simple Diofantine equation.

```
?- plus(suc(0), suc(suc(0)), N).
N=suc(suc(suc(0)))
?- plus(suc(0), N, suc(suc(0))).
N=suc(0)
?- plus(N, M, suc(suc(0))).
N=0, M=suc(suc(0))
N=suc(0), M=suc(0)
N=suc(suc(0)), M=0
?- plus(suc(N), suc(M), 0).
no
```

4.2 Concrete grammar

An axiom may be seen as a special case of a rule where the right hand part is empty.

$$\begin{aligned} \mathit{rule} &\rightarrow \mathit{term}. \mid \mathit{term}\ \texttt{:-}\ \mathit{terms}. \\ \mathit{terms} &\rightarrow \epsilon \mid \mathit{term}(,\mathit{term})^* \\ \mathit{term} &\rightarrow c \mid X \mid f(\mathit{terms}) \end{aligned}$$

where c is constant literal not starting with a capital letter, X is a variable starting with an upper case letter and f is a function name starting with a lower case letter.

4.3 Lists

Lists can be introduced by the following rules.

```
list(empty).
list(cons(X, Xs)) :- list(Xs).
```

So `list(X)` is a predicate that is true when it can be deduced that `X` is `empty` or has the form `cons(X, Xs)` where it can be deduced that `list(Xs)`. (`list` is predefined.)

```
?- list(empty).
yes
?- list(cons(3, empty)).
yes
?- list(cons(empty, 3)).
no
?- empty.
no
```

Lists are predefined in most Lisp system with a special syntax. `[]` will denote the empty list, `[X|Xs]` is the list with head `X` and tail `Xs`. A list may also be specified by enumerating the elements separated by commas, e.g. `[1,2,3]`. The intended meaning of `append(X, Y, Z)` is that `Z` is the result of appending `X` and `Y`.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

(`append` is predefined.)

4.4 Unification

When a Prolog system gets the query

```
?- append([1,2], [3,4], Ls)
```

it must use the rule

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

and match the actual arguments of `append` with the head of the rule. Actually, we have a system of equations.

```
[X|Xs] = [1,2]
    Ys = [3,4]
[X|Zs] = Ls
```

The solution is

```
X = 1
Xs = [2]
Ys = [3,4]
Ls = [1|Zs]
```

where `Zs` is any list.

The runtime system will need an algorithm to solve such systems of equations. Such an algorithm was devised by J.A. Robinson in 1965, the *unification* algorithm. It is presented below.

```
1  Subst σ; // initialized to the empty substitution
2  void unify(Term s, Term t) {
3    if (s is a variable) s := σ s;
4    if (t is a variable) t := σ t;
5    if (s is a variable && s.equals(t)) {
6      // do nothing
7    } else if (s== f(s_1,...,s_n) and t== g(t_1,...,t_m)) {
8      if (f.equals(g) and n == m) {
9        for (int i=1; i≤ n; i++) {
10         unify(s_i,t_i);
11       }
12     } else {
13       exit with failure
14     }
15   } else if (s is not a variable) {
16     unify(t, s);
17   } else if (s occurs in t) {
18     exit with failure
19   } else {
20     σ = σ[s ↦ t] and add [s ↦ t] to σ;
21   }
22 }
```

In this program σ is a global variable containing a substitution mapping variables to terms. We have previously used the symbol σ to denote a state mapping variables to integer values. Here σ denotes a substitution from variables to terms and $\sigma[s \mapsto t]$ represents an updated substitution where the variable s is replaced by the term t in all mappings. If `s` in the `else` is a constant f will be that constant and $n = 0$. Similarly for `t`.

If we want to apply the algorithm to the `append` query above we cannot use the special list syntax. We have to consider

```
s = append(cons(1, cons(2, empty)), cons(3, cons(4, empty)), Ls)
t = append(cons(X, Xs), Ys, cons(X, Zs))
```

The conditions in line 7 and 8 are true and there is a first recursive call within the for loop from line 10:

```
s = cons(1, cons(2, empty))
t = cons(X, Xs)
```

Again there is a recursive call in line 10 with

```
s = 1
t = X
```

Now the condition in line 15 and we have a new recursive call with

```
s = X
t = 1
```

This will make $\sigma = [X \mapsto 1]$ in line 20 without a new recursive call.

Next we continue with the for loop in line 9 and a call with

```
s = cons(2, empty)
t = Xs
```

which will add the substitution $[Xs \mapsto cons(2, empty)]$ to σ.

Lines 3 and 4 will have no effect until line 20 has been executed. Line 20 is the basis case for the recursion (unless the unification fails). The reason for applying σ to `s` and `t` is that if we replace one occurrence of variable with a term we have to replace all occurrences of this variable with the same term.

The final value of σ is the most general substitution that will make `s` and `t` equal. It is called the *most general unifier*.

4.5 Resolution

Below is an algorithm for finding a proof for a given query. The algorithm is nondeterministic. For each loop in the while statement there is a choice. Making a bad choice will lead to failure.

```
1  void resolve(RuleList rules, Term query) {
2    TermList resolvent = [query];
3    while (resolvent is not empty) {
4      let t be the first term in resolvent
5      choose a rule r = g :- terms with fresh variable names such that
6        t and g unifies with most general unifier σ.
7        if there is no such rule
8        then
9          exit with failure
10       else
11         replace t in resolvent with terms.
12       apply σ to resolvent
13   }
14 }
```

Implementing this algorithm we have to *backtrack* to the last choice point where an alternative was present if the current choice lead to failure.

Consider the query `?- grandfather(X, lena)` to the intial example of this chapter. Initially the resolvent just contains this term and there is just one rule that may be used, the `grandfather` rule. According to the algorithm we should use fresh variables in the rule. This is not strictly required in this case but doing so we get a new resolvent, `father(X, Y1), father(Y1, lena)`. Next `father(X, Y1)` will be unified with `father(gustav, eva)` returning the resolvent `father(eva, lena)` which will fail for all axioms.

We have to return to the last choice point where there are unexplored alternatives. Thus `father(X, Y1)` will be unified with `father(gustav, lena)` which will lead to the same kind of failure.

Finally unify with `father(oskar, gustav)` yilding the resolvent `father(gustav, lena)` wich will lead to success after an initial failure.

4.6 Prolog systems

If you want to try the examples in this chapter and some Prolog programs of your own there are a number of free and commercial Prolog systems.

1. Gnu prolog, `http://www.gprolog.org/`
2. Swi-prolog, `http://www.swi-prolog.org/`
3. Sicstus prolog, `http://www.sics.se/isl/sicstuswww/site/index.html`

You may build your own system as Assignment 3.

There are many text books on Prolog. The following is a particularly good one:

- L. Sterling, E. Shapiro: *The Art of Prolog: Advanced Programming Techniques* , MIT Press, 1993.

It seems to be out of print so you may have to use a library.