G6021 Comparative Programming

Part 5 - Logic Programming (Prolog)

Logic Programming Languages

- Use *logic* to express knowledge, describe a problem.
- Use *inference* to compute, manipulate knowledge, obtain a solution to a problem.

Based on this idea, several programming languages have been developed. The most popular is *Prolog*.

Prolog

- Logic: The *clausal fragment of classical first-order logic*.
- Inference System: *Resolution*

Prolog = Clausal Logic + Resolution + Control Strategy.

Impure Prolog: adds non-logical primitives (efficiency).

Advantages

- Knowledge-based programming: the program just describes the problem.
- It is a *declarative* style of programming: the program says *what* should be computed, rather than *how* it is computed (although this is not true for impure languages).
- Precise and simple semantics.
- The same formalism can be used to specify a problem, write a program, prove properties of the program.
- The same program can be used in many different ways.

The first points are shared with functional languages (which are also declarative languages) but the last point is specific to logic programming languages.

Disadvantages

- Inefficient: facilities to support efficient arithmetic, file handling, etc. are provided at the expense of the formalism's declarative semantics.

- Most logic languages are restricted to a fragment of classical first-order logic. There are some languages based on more powerful logics, but they are not widely available.

History

- Based on Unification, which is a key step in the Principle of Resolution.
- The unification algorithm was first sketched by Jacques Herbrand in his thesis (in the 1930's).
- In 1965 Alan Robinson introduced the Principle of Resolution and gave a unification algorithm.
- Finally, around 1974 Robert Kowalski, Alain Colmerauer and Philippe Roussel defined and implemented a logic programming language based on these ideas (Prolog).

Programs

- A *literal* is an atomic formula or a negated atomic formula. To build atomic formulas we use terms and predicates.
  Examples
  `p, odd(3), even(X), add(2,3,5), ¬raining`

- A Horn clause is built out of literals.

$$P_1 \;:-\; P_2, \ldots, P_n.$$

  and we read it as: "P_1 if P_2 and ... and P_n"
  If the clause contains just P_1, then it is a *Fact*. We write

$$P_1.$$

  If the clause contains only negative literals, we call it a *Goal* or *Query* and write

$$:-P_2, \ldots, P_n.$$

- A Prolog program is a set of *Horn clauses*.

Example 1 - propositions

```
r.
s.
p :- q,r.
q :- s.
:-p
```


- The first two clauses are facts, the next two are rules and the last is a goal (query).
- In Prolog, we usually write
  - `:-p` as `?p.`
  - `p :- q,r.` as $p \leftarrow q, r.$

  Here we will mix notations.
- Exercise: Is the Goal valid in this program?
- A Prolog program can be seen of a collection of facts and a query about these facts.

Example 2 - backtracking

```
r.
s.
a.
p :- a,b,c
p :- q,r.
q :- s.
?p
```


- There are two rules for `p`. Prolog will try them both: backtracking.
- This gives a tree structure (SLD-resolution tree) and some of the branches will succeed, some will fail
- Exercise: Is the Goal valid in this program?
- Compare SLD tree with reduction graph.

(Note: SLD = Selective Linear Definite clause)

Examples 3 - predicates

```
nat(0).
nat(s(X)) :- nat(X).
?nat(s(s(0))).
?nat(Y).

add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
?add(s(s(0)),0,A).
?add(s(0),B,s(s(0))).
?add(A,s(0),s(s(0))).
?add(A,B,s(0)).

even(0).
even(s(s(X))) :- even(X).
?even(s(s(s(0)))).
```

Substitution and Unification

We explain these concepts through examples:

```
nat(0).
nat(s(X)) :- nat(X).
?nat(s(s(0))).
?nat(Y).
```


- Attempt to understand the computation mechanism for the first query by thinking about functional programming and substitution. `?nat(s(s(0)))` will *match* against the second clause, and we generate a substitution (cf. β-reduction): `X = s(0)`. We now have to solve `nat(X)` where `X = s(0)`, (i.e. `nat(s(0))`), and so on.
- However, how do we explain `?nat(Y)`?

Substitution and Unification

```
nat(0).
nat(s(X)) :- nat(X).
?nat(Y).
```


`?nat(Y)` will match either (both) clause(s).

- `nat(0)` succeeds, giving an answer substitution: `Y=0`.
- `nat(s(X))` succeeds with `Y=s(X')` now we have to solve `nat(X')`....

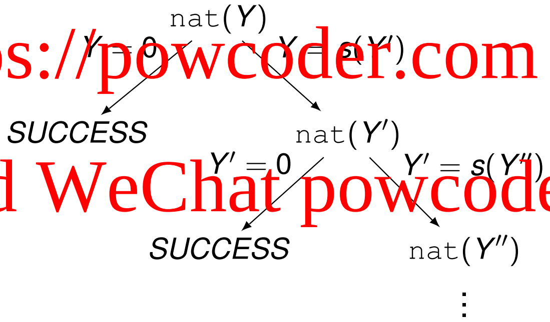
Repeating we get: `X'= 0, X'=s(X")`, so `Y=s(0)`, etc. which gives a tree of possible solutions.

Note that we have to go back and apply substitutions: this is unification (cf. type checking).

Unification will fail if we try to unify something like `Unify(X,nat(X))` (cf. unify A with $A \to A$). This is known as the *occur check*.

Example SLD tree

```
nat(0).
nat(s(X)) :- nat(X).
?nat(Y).
```

Thus, the solutions are 0, $s(0)$, $s(s(0))$, ...

Examples continued


```
basedOn(prolog,logic).
basedOn(haskell,maths).
likes(alice,maths).
likes(bob,logic).
likes(X,P) :- based(P,Y), likes(X,Y).
```


The first four clauses are facts, the last clause is a rule.

Compute the following Goal:


```
?likes(Z,prolog).
```

Control Mechanism

In its pure form, logic programming languages are declarative. They admit two interpretations:

- a *declarative interpretation*, in which the meaning of the program is defined with respect to a mathematical model (the Herbrand Universe). It corresponds to a *denotational semantics*.

- a *procedural interpretation*, which explains how the program is used in computations. This is the *operational semantics*.

The operational semantics of Prolog is based on the use of SLD-Resolution, with automatic backtracking. We resolve upon the leftmost literal each time: this strategy is complete for Horn clauses.

Exercise: think about alternative strategies.

Prolog supports interactive programming: The user can submit a query and ask for one or more solutions. This is analogous to the way functional programming languages are used:

- In functional languages the interaction is achieved by means of expressions that are evaluated using a collection of function definitions.
- In logic languages the interaction is achieved by means of queries that are resolved using a collection of predicate definitions.

Note that:

- The equations in a functional program define functions.
- The clauses in a logic program define predicates.

Lists in Prolog

In the program

```
append([],L,L).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```


`append(S,T,U)` expresses that the result of appending the list T onto the end of list S is the list U.

- The term `[X|T]` denotes a list where the first element is X (*the head of the list*) and T is the rest of the list (*the tail of the list*).
- `[]` denotes the empty list.
- We abbreviate `[X|[Y|[]]]` as `[X,Y]`.

Exercise: Compute Goals such as:

```
?append([0],[1,2],U)
?append(X,[1,2],U)
?append([1,2],X,[0])
```

Exercise: Try the first clause and compare with how you would write an append function in Haskell using accumulating parameters.

Let's try: `?append([0],[1,2],U)`

```
append([],L,L).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```


$$
\begin{array}{r}
\texttt{?append}([\texttt{0}],[\texttt{1},\texttt{2}],\texttt{U}) \\
\big\downarrow\ \texttt{X}'=\texttt{0},\texttt{L}'=[],\texttt{Y}'=[\texttt{1},\texttt{2}],\texttt{U}=[\texttt{X}'|\texttt{Z}'] \\
\texttt{?append}([],[\texttt{1},\texttt{2}],\texttt{Z}') \\
\big\downarrow\ \texttt{L}''=[\texttt{1},\texttt{2}],\texttt{L}''=\texttt{Z}' \\
\textit{SUCCESS}
\end{array}
$$

Applying substitutions we get: `U=[0,1,2]`

Another example: `?append([0,1],[1,2],U)`

```
append([],L,L).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```


$$?\mathtt{append}([0,1],[1,2],\mathtt{U})$$

$$\big\downarrow\ \mathtt{X}' = 0, \mathtt{L}' = [1], \mathtt{Y}' = [1,2], \mathtt{U} = [\mathtt{X}'|\mathtt{Z}']$$

$$?\mathtt{append}([1],[1,2],\mathtt{Z}')$$

$$\big\downarrow\ \mathtt{X}'' = 1, \mathtt{L}' = [], \mathtt{Y}'' = [1,2], \mathtt{Z}' = [\mathtt{X}''|\mathtt{Z}'']$$

$$?\mathtt{append}([],[1,2],\mathtt{Z}'')$$

$$\big\downarrow\ \mathtt{L}'' = [1,2], \mathtt{L}'' = \mathtt{Z}''$$

SUCCESS

Applying substitutions we get: `U=[0,1,1,2]`

Difference Lists

- In Prolog (and Haskell, etc.) you can access only the first element of a list.
- Prolog difference lists: represent a list as the difference of two lists: `A\B`. Examples: `[1,2,3]\[2,3]`, `[1|X]\X`

- Can write a constant time append clause:

  `append(A\B,B\C,A\C).`

- Compare with pointers. Standard lists allow access to the head (first element) of the list, and access to all other elements are through the head.

- Difference lists can be thought of as a data structure where we have a pointer to first and last elements of a list. This can be done in languages like C (structures and pointers) and Java (with objects). In both cases we need pointers to the first and last elements of a list so that append becomes a constant time operation.

Exercise: think how to do constant time list concatenation in Java, C, Haskell.

Difference lists example


```
append(A\B,B\C,A\C).

?append([1,2,3|V]\V, [4,5,6|W]\W, R)

A=[1,2,3|V], V=B, B=[4,5,6|W], C=W, R = A\C

R = [1,2,3,4,5,6|W]\W
```

The order of the clauses

Prolog will try to find all solutions, but the order of the clauses is important:


```
nat(0).
nat(s(X)) :- nat(X).
?nat(s(s(0))).
```



```
nat(s(X)) :- nat(X).
nat(0).
?nat(s(s(0))).
```

What about `?nat(Y)`?

- Termination of Prolog programs depends on a good order of the clauses.
  Alternatives? (Depth first vs. depth first?)
- Compare with overlapping patterns in Haskell.

Summary

- In logic programming, computation is *proof search*.
- Applications: very successful in several domains:
  - Artificial intelligence.
  - Deductive databases.

- Like Haskell, very useful for prototyping, list processing, etc.
- Exercises: See worked examples on web page
  Research topic: relationship between functional (proof normalisation) and logic programming (proof search).