Implementing a micro-Prolog System

John Easterday

April 19, 2020

What is Prolog?

Prolog is a declarative programming language based off of first-order logic. The program logic comes from programming relations between data rather than programming the steps to transform one piece of data into another. Data types in Prolog are called "terms" and consist of atoms, variables, and compound terms. Atoms are just names that start with a lowercase letter or digit, such as "atom". Variables must start with an uppercase letter or an underscore, such as "X". Compound terms look a lot like function calls from more conventional programming languages and can be thought of as the relationship between the data in the arguments of the compound term. These compound terms look like "relation(atom, X)." which defines the relationship "relation" between the atom "atom" and the variable "X". We can code more complex relationships by using what are called "rules" in Prolog. Rules follow the pattern of "lhs :- rhs" meaning "lhs" unifies if "rhs" is true. To give an example, let's say we wanted to make a relationship between two pieces of

data called "auntof". The relationship is named "auntof(X, Y)." which reads as "X is the aunt of Y". The corresponding rule would be:

```
auntof(X, Y) :-
  parentof(Z, Y),
  sisterof(X, Z).
```

This rules reads as "X is the aunt of Y if Z is the parent of Y and X is the sister of Z." So as you can see, these rules read very much like english. Combining these rules and data, the program we build up is essentially a database of facts that our system "knows". Following the analogy, if the program is a database then execution is simply querying the database. I will often refer to the program as a database and actions we are taking with our program are queries to the database.

One very interesting property of Prolog is because you are coding in relationships there are no set inputs and outputs of these relations. So, using our "auntof" example, we can get all the people who "wendy" is the aunt of by using the query "auntof(wendy, X)." However, using the same rule, we can get all the people who are the aunt of "john" by using the query "auntof(X, john)." In more conventional programming languages, having this property for functions would require an immense amount of code at best and would be impossible at worst.

The Algorithm

I alluded to this in the last section but Prolog works off what is called "unification" and when we ask our program something we are asking if the

query "unifies" with any of the facts we have in the database. There are three basic rules to unification:

1. An atom unifies with an atom if and only if the atoms have the same name.

2. A variable unifies with everything.

3. A compound term unifies with a compound term if and only if they have the same name and the same arity.

With these simple rules, you can get a basic Prolog system up and running. However, with these unification rules we don't get unification with Prolog rules, so we need something a little more sophisticated. In Prolog, when you query the database you lazily get back all of the variable bindings that make that query true. The "lazily" just means when the database gives you one of the bindings it doesn't find the next binding until you tell it to. Prolog can use many algorithms to search through the database. One of the simpler algorithms is your basic depth-first search but one of the more complicated algorithms is called constraint propogation. As the name states, with constraint propogation, the search is constrained so you aren't searching over things that don't meet the constraints.

The Implementation

I started by making sure a query is a valid query and follows the syntax I made a basic Prolog recursive descent parser which in turn built up an

abstract syntax tree. The abstract syntax tree helped with unification since I found it easier to tell is two trees unify rather than if two strings unify. To implement the unification algorithm, I started by following the three basic rules and I would loop through all the facts in the database and try to unify the query with the fact. However, I had to add that all the arguments of a compound term must unify too for the compound term to unify.

To get the unification search working with rules I added a "rule" ast node. So, if there is a rule in the database and I am trying to unify it with some query, they unify if the query unifies with the left node of the "rule" node and if the right node of the "rule" node unifies with some fact in the database. The recursive nature that unification takes when unifying trees means we already have the basic depth-first search. I didn't want to stop there, so when searching for the right hand side of the rule I apply the constraints that unifying with the left hand side produced and then do the search.

You may remember in the example from the "What is Prolog?" section that we had:

```
parentof(Z, Y) , sisterof(X, Z)
```

which means "Z is the parent of Y AND X is the sister of Z." This "," operator or "and" operator works the same as logical and where the whole conjunction unifies if both the left hand side and the right hand side are facts in the database. Similarly, we have the "or" operator for logical or and it looks like ";" and the whole disjunction unifies if either the left hand side or the right hand side are facts in the database.

The only part I didn't implement correctly is the lazy evaluation of the

algorithm, so when you make a query you get back all of the bindings that make it true at one time.

Ancestor Program

One basic Prolog program is an ancestry program or a family tree program. Running "./prolog family.pl" runs my family tree program. The example rule "auntof(X, Y)." is a query that works in this program along with all of the other common familial relationships. Note they all look like "[RELATIONSHIP]of(X, Y)" and read as "X is the [RELATIONSHIP] of Y". Something else to note about this program, if you ask the query:

```
siblingof(john, X).
```

You will get the bindings:

```
X = ashley
X = ashley
```

Obviously there isn't two different ashley's since an atom must unify with an atom but this comes from the rule being "Two people are siblings if they have the same parent." but Prolog finds both of my parents and matches with ashley with both of my parents.

Also, I didn't build into the system the ability to check if two things are equal so I had to manually input the equality and inequality relationships.

Towers of Hanoi

The Towers of Hanoi is a common program people implement when they learn a new language. A query to this program looks like "move(N, X, Y, Z)." where N is the number of disks, X is the "from" peg, Y is the "to" peg, and Z is the auxillary peg. So the query:

```
move(4, left, right, center).
```

Gives you the steps to move four disks from the left peg to the right peg.

Something to note, I never implemented the ability for the system to understand what numbers are so I had to implement numbers, subtraction, and greaterthan by hand. So, this algorithm only works for up to ten disks.