Introduction to Pure Prolog

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

October 28, 2020

Table of Contents

Table of Contents

You might be wondering, "haven't we been learning Prolog this whole time? Why is this lecture titled, 'Introduction to Pure Prolog?"'

Yes, we have been learning Prolog this whole time. You can run the examples shown thus far in this course using SWI-Prolog or other Prolog implementations.

But, you have been learning a small, core subset of Prolog.

In this lecture, we will cover additional pure Prolog features, such as arithmetic, and we will also cover Prolog's execution model.

Table of Contents

Let's recall the resolution algorithm we studied in our last lesson:

Resolution with Unification

Input: A goal G and a program P

Output: An *instance* of G that is a logical consequence of P, or *no* otherwise

Algorithm:

Initialize the resolvent to G.
while the resolvent is not empty *do*
    choose a goal A from the resolvent
    choose a (renamed) clause $A' \leftarrow B_1,\ldots,B_n$ from P
        such that A and A' unify with mgu θ
    (if no such goal and clause exist, exit the *while* loop)
    replace A by $B_1,\ldots,B_n$ in the resolvent
    apply θ to the resolvent and to G
If the resolvent is empty, *then* output G, *else* output *no*.

Figure 4.2 An abstract interpreter for logic programs

Figure: Resolution algorithm with unification [Sterling and Shapiro 1994, p. 93]

There are two decisions that Prolog implementations need to make:

There are two decisions that Prolog implementations need to make:

1. How to choose a goal A from the resolvent.

There are two decisions that Prolog implementations need to make:

1. How to choose a goal A from the resolvent.
2. How to replace A with $A' \leftarrow B_1, ..., B_n$ from the program.

- In Prolog, the resolvent is a stack.
- When the interpreter chooses a goal A from the resolvent, it pops it from the stack.
- When the interpreter chooses a $A' \leftarrow B_1, ..., B_n$ from the program:
  1. It chooses the first goal in the program that unifies with A. According to Sterling and Shapiro, "if no unifiable clause is found for the popped goal, the computation is unwound to the last choice made, and the next unifiable clause is chosen" [p. 120].
  2. It pushes the elements $B_1, ..., B_n$ onto the stack.

Rule Order in Prolog

- Because of how Prolog implementations handle resolvents, rule order in Prolog matters.
- To directly quote Sterling and Shapiro, "*the rule order determines the order in which solutions are found*" [p. 130, emphasis original].
- This is unimportant for pure Prolog programs since reordering rules will not affect the results of computations, but this will be very important once the cut feature is introduced next week.

Is it possible for a Prolog query to not terminate?

Is it possible for a Prolog query to not terminate? Yes, it is possible for a Prolog query to not terminate.

Non-termination in Prolog

- This is particularly a problem with left-recursive rules.
  - A *left-recursive* rule is one where the first goal in the body is recursive (e.g., `married(X,Y) :- married(Y,X).`).
- The best way to deal with possible non-termination in left-recursive rules is to avoid them by rewriting the rule in such a way to avoid left-recursion.
  - For example, we could define a new predicate, `are_married(Person1,Person2)` with the following rules:
    - `are_married(X,Y) :- married(X,Y).`
    - `are_married(X,Y) :- married(Y,X).`

Circular definitions are also problematic; make sure you avoid them.

The ordering that does matter considerably is the ordering of goals; i.e., each B_i in $A \leftarrow B_1, ..., B_n$ where $1 \leq i \leq n$.

The ordering that does matter considerably is the ordering of goals; i.e., each B_i in $A \leftarrow B_1, ..., B_n$ where $1 \leq i \leq n$.

Goal ordering is important not only for runtime efficiency reasons, but also for termination reasons.

Given the rule

```
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

What would happen if the goals were swapped, resulting in the rule

```
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

Given the rule

```
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

What would happen if the goals were swapped, resulting in the rule

```
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

This would introduce left-recursion to the rule, which means the rule would become non-terminating.

Table of Contents

System Predicates

- Not all operations in Prolog can be conveniently or efficiently expressed as pure logic programs.
  - For example, we want to be able to perform arithmetic using our computer's ALU and not by using custom functors.
- To provide such functionality, Prolog has system predicates.
  - This is analogous to Scheme's built-in functions such as `define` and `lambda`.

Arithmetic in Prolog

- Prolog provides built-in predicates for performing arithmetic.
- Examples include `+`, `-`, `*`, and `/`.
- We can perform queries using the following form: `Value is Expression?`.
- Examples of queries:
  - `(X is 3+5)?` $\Rightarrow$ `X=8`.
  - `(8 is 3+5)?` $\Rightarrow$ `yes`.

What does the query `(3+5 is 3+5)?` evaluate to?

What does the query `(3+5 is 3+5)?` evaluate to?

Actually, this query will fail.

The reason this query fail is because the left side of `is` expects a value, not an expression.

Comparisons in Prolog

- Prolog provides the following built-in comparison predicates: <, >, <=, and >=.
- We can perform queries using the following form: `A op B`, where `A` and `B` are arithmetic expressions and `op` is one of the above comparison predicates.
- Examples:
  - `(1 < 2)?` $\Rightarrow$ `yes.`
  - `(4*6 < 5*5)?` $\Rightarrow$ `yes.`
  - `(6/2 < 5-1)?` $\Rightarrow$ `no.`

What does the query `(N < 1)?` evaluate to?

What does the query `(N < 1)?` evaluate to?

This query results in an error since N is not an expression, but rather, a variable.

Arithmetic Queries

We can now write rules such as the following:

```
plus(X,Y,Z) :- Z is X+Y.
```

Arithmetic Queries

We can now write rules such as the following:

```
plus(X,Y,Z) :- Z is X+Y.
```

Unfortunately, we can't perform queries such as the following:

```
plus(3,X,8)?
```

In order to do so, we need to use *meta-logical predicates*, which will be covered in the next lecture and is discussed in Chapter 10 of the textbook.

Table of Contents

On Monday we will be covering more system predicates that deal with types in Prolog, and we will also be covering meta-logical predicates, which allow us to exploit the full power of arithmetic in our Prolog programs.

Class Updates

- I should finally have Project 1 grades finished no later than the end of this weekend (I apologize for the delay; I've been swamped lately).
- Lab 3 is due Thursday.
- I will be assigning Lab 4 and Project 2 on Monday. Lab 4 will consist of more exercises from *The Art of Prolog*, while Project 2 will consist of a significant Prolog programming exercise.
- Just three more lectures on Prolog (November 2, November 4, and November 9)! November 11 is a holiday (Veteran's Day), and November 16 will be the start of our third and final segment of the course, which will revisit object-oriented programming, using Smalltalk as our main language.