Prolog, pt 3: Arithmetic, Data Structures, Cuts

CS 440: Programming Languages and Translators, Fall 2020

A. Sources for Study

- Learn Prolog Now: www.learnprolognow.org/
- SWI Prolog: http://www.swi-prolog.org/

B. Arithmetic

- [From Sections 5.1, 5.2, and 5.4 of Learn Prolog Now]

Arithmetic terms are written in the usual way in Prolog (numerals, +, -, *, /, parentheses).

- You can also write them in prefix form: *(2, +( 3, 4)) unifies with 2 * (3 + 4).

- There is a difference between unifying with an expression (using operator =) and evaluating an expression (using operator is).

  - If X is instantiated, because unification is a syntactic operation, the queries X = 2+2 and 2+2 = X will succeed iff the value of X is the term 2+2.
  - If X is uninstantiated, the queries X = 2+2 and 2+2 = X both succeed and bind X to literally 2+2 because = is a textual operation.

- The query X is 2+2 first calculates 2+2 to get 4.
  - If X is instantiated to the number 4, then X is 2+2 succeeds.
  - If X is instantiated to the symbol 4, then X is 2+2 fails. (For the same reason, '4' is 4 fails.)
  - If X is uninstantiated, then X is bound to the number 4 and X is 2+2 succeeds.

- The query e₁ is e₂ is not symmetric because only e₂ gets evaluated and Prolog tries to unify the unevaluated e₁ with the value of e₂.

- The query 2+2 is X never succeeds. (Let e₁ be 2+2 and e₂ be X.)
  - If X is not instantiated, then the query fails.
  - If X = 4, the test becomes 2+2 is 4, which fails because 2+2 = 4 fails.
  - If X = 2+2, then Prolog evaluates 2+2, gets 4, and then 2+2 = 4 fails.

- On the other hand, 4 is 2+2 succeeds; Prolog evaluates 2+2, gets 4, and 4 = 4 succeeds.

- Comparisons

  - For arithmetic equality, you don't want = because it tries to unify its two sides (2+2 = 4 fails).
  - There's also a \( \neq \) operator which means "doesn't unify with": \( 2+2 \neq 4 \) succeeds, but so does \( 2+2 \neq 5 \), so you definitely don't want to use \( \neq \) for arithmetic inequality.
• The operators for arithmetic equality and inequality are \( \equiv \) and \( =\neq \). (The equal signs on both sides are mnemonic that both operands get evaluated.) E.g., \( 2 + 2 = 4 \) and \( 2 + 2 = 5 - 1 \) are both true.

• You can also compare arithmetic expressions using and \( < \), \( =< \), \( > \), and \( >= \) (where \( =< \) means \( \leq \)). Both sides of the operator are evaluated, e.g., \( 2 + 2 < 7 \times 3 \) yields true.

C. Data Structures in Prolog

• Here are a couple of basic ways to encode data structure information about values being related by a property or function:
  - The property or function is used as the functor name, e.g., \( \text{road(\text{chicago, st\_louis})} \) to indicate graph connectivity or \( \text{add(2, 5, 7)} \) for arithmetic evaluation.
  - The property or function is used as an argument as part of a larger property, e.g., \( \text{cities(\text{road, chicago, st\_louis})} \) or \( \text{cities(\text{flight, chicago, st\_louis})} \) or \( \text{eval(\text{\textquote{\textquote{\textquote{+}}, 2, 5, 7})} \) (The apostrophes around + are optional.)
  - Making a property name an argument makes it easy to ask "Are X and Y related by a property?" E.g., as a query, \( \text{cities(\text{Method, \text{chicago, st\_louis})} \) asks if there's a value for Method that proves the query. In our example, Method = road and Method = flight both work.

• Lists are also available, as in \( [1, 2, 3, \text{alex, wombat}] \). (The values don't have to have the same type.) The cons operation is written with a vertical bar: \( [1 | [2, 3]] = [1, 2, 3] \).

• E.g., we can represent a path as a list of the nodes visited:

\[
\text{link(1, 2). link(2, 5). link(3, 1). \% \text{3 facts on one line.}}
\]

\[
\text{path(Start, End, [Start, End]) :- link(Start, End). \% \text{A link is a path}}
\]

\[
\text{path(Start, End, [Start, Neighbor | Remainder]) \% \text{Link to a neighbor, then to end?}}
\]

\[
\text{:- link(To, Neighbor), path(Neighbor, End, [Neighbor | Remainder]).}
\]

• The database above supports queries like:

\[
\text{path(3, 5, Path) \% \text{What paths are there from 3 to 8?}}
\]

\[
\text{path(3, End, Path) \% \text{What paths start at 3?}}
\]

\[
\text{path(Start, 5, Path) \% \text{What paths end at 5?}}
\]

\[
\text{path(X, X, Path) \% \text{Are there any cycles in the graph?}}
\]

• There are no cycles in the graph as given, but adding \( \text{link(5, 1)} \) will get us one.

\[
\text{link(1, 2). link(2, 5). link(3, 1). link(5, 1).}
\]

\[
\text{path(X, X, Path).}
\]

• But there's a problem: There are an infinite number of solutions because concatenating cycles gives you a cycle:

\[
P = [1, 2, 5, 1], X = 1
\]

\[
P = [1, 2, 5, 1, 2, 5, 1], X = 1
\]
Arguably, this is the correct behavior (they all are solutions, after all). But if you're only proving something true or false, there's no point in having an infinite number of ways to prove true, as in

\[ \text{cycle}(X) \leftarrow \text{path}(X, X, \_). \ % \_ \text{ is a don't care variable} \]

**Cuts, Green and Red**

- The way to avoid this problem is to use a "cut", which is a non-declarative (i.e., procedural) feature of Prolog that means "only look for one proof of the terms to the left". It's written as a term `!`. Going left-to-right, `!` is always provable, but going right-to-left (during backtracking), `!` always fails. We can add it to the `cycle` definition so that once we prove `cycle(X)` is true, we won't look for other proofs of it.

\[ \text{cycle}(X) \leftarrow \text{path}(X, X, \_), !. \]

- This cut is an example of a **green cut**, one that doesn't change the set of answers.

- A **red cut** is a cut that does change the set of possible answers. Here's an example that adds a red cut to the path rule.

\[ \begin{align*}
\text{path}(S, E, [S, E]) \leftarrow & \ \text{link}(S, E), !. \ % S = \text{start}, E = \text{end} \\
\text{path}(S, E, [S, N \mid R]) \leftarrow & \ \text{link}(T, N), \text{path}(N, E, [N \mid R]), !. \\
\end{align*} \]

% Link to a neighbor N, then to end using remainder of path

- With this definition of `path`, if we find a path, we won't look for another. This is fine if you only ever want to find one path, but if you're looking for multiple possible paths, then you wouldn't want to use this.

**Procedural (Non-Declarative) Parts of Prolog**

- Most of Prolog is non-declarative = purely logical, but there are some non-declarative = procedural parts:
  - **Cuts**, because they ban backtracking.
  - **Knowledge base order**: When looking for an applicable fact or rule, Prolog searches the database of facts and rules from top to bottom.
  - This implies that for recursive rules, you should put the base cases above the recursive cases otherwise you might get infinite recursion.
  - **Order of terms in a rule**: When a rule has multiple goals (result :- goal₁, goal₂, ...), Prolog tries to prove them from left to right. In normal situations, goal order isn't important. But if a rule needs to carry out a short-circuiting test, goal order is important. For example, the Prolog translation of the C/Java/etc. test \((0 \leq k && k < n && \text{array}[k] == 0)\) needs to take order into account.