Prolog, pt 2: Unification; Resolution; Arithmetic

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. Review

- Prolog is a programming language based on logic.
  - At its core, it is a declarative language: Instead of writing a program, you write out the specification for what a legal output looks like.
  - To execute a query (e.g., answer the question "Is there a value that satisfies ...(property)...?"), Prolog does a backtracking search to find a value that satisfies the query.
  - For efficiency, Prolog does have some imperative features (ordering of the database of facts and rules; left-to-right search to satisfy the requirements of a rule).
  - To form predicates, Prolog builds up from primitive predicates $P(a_1, a_2, ..., a_n)$, (and $e_1 < e_2$, etc.).
    - A fact declares that some predicate is true. E.g., food(pizza).
    - A rule declares an implication. A rule has the form $q : - p_1, p_2, ..., p_n$. where $q$ is the conclusion and the $p$'s are the antecedents. The semantics are that if all the $p$'s are true, then $q$ is true; i.e., $p_1 \land p_2 \land ... \land p_n \rightarrow q$.
      - E.g., mammal(X) :- dog(X). % if X is a dog then X is a mammal; i.e., all dogs are mammals.

C. Atoms

- Atoms are basically symbols
  - Named constants like $x_yz$ or ‘xy z’ (spaces are ok if in single quotes). First symbol is lowercase letter; underscores and digits ok.
  - Numerals like 27. (Note ‘27’ is not a numeral.)
  - Variables like $X_yz$, _XYZ. First symbol is upper-case letter or underscore.
  - Note just plain underscore _ is treated differently. (Basically as a don't-care variable; we'll look at this later.)
  - Special symbols like @, ; Generally have some special meaning.

D. Unification in Prolog

- [Also see Learn Prolog Now 2.1]
Recall that unification is a textual operation where we try to match terms built up from constants, numerals, variables, compound terms (which have a "functor" name and parenthesized arguments, as in \( f(a, b) \)). (There are also lists and list operations yet to be discussed.)

Terms without variables have to match exactly: \( f(1, 2) = f(1, 2) \) but not \( f(2, 1) \).

A variable may have to be **instantiated** – given a value in order to get matching terms. The value can be a **ground term** (have no variables); it can also include variables.

- E.g., \( f(X, Y) = f(g(z), h(W)) \) by using \( X = g(z) \) (a ground term) and \( Y = h(W) \) (uses a variable).

If used multiple times in the same terms, a variable must be instantiated to the same value everywhere.

- E.g., \( f(X, X) = f(3, 3) \) is okay but \( f(X, X) = f(5, 3) \) is not.

Matching is syntactic, not semantic: \( 2 + 2 \neq 4 \) because they are different as pieces of text.

**Unification Algorithm**

We're given some pairs of terms (“equations”) and want to find a collection of substitutions that, when applied, makes each pair match simultaneously. E.g., \( X = g(Y) \), \( g(g(8)) = g(X) \) unify if \( X = g(8) \) and \( Y = 8 \) because then the equations become \( g(8) = g(8) \), \( g(g(W)) = g(g(8)) \).

Given two terms, we can check for unification

- Two named constants or two numerals must be exactly the same: \( xyz = xyz \) and \( 7 = 7 \).
- (Evaluation of expressions is not taken into account, so \( 6 + 1 \neq 7 \).)
- For two compound terms to unify, they must have the same function name (“functor,” in Prolog), the same number of arguments, and the arguments must match elementwise.
  - E.g., for \( f(\text{term}_1, \text{term}_2) = f(\text{term}_3, \text{term}_4) \), we need \( \text{term}_1 = \text{term}_3 \) and \( \text{term}_2 = \text{term}_4 \).
  - Similarly, \( f(t_1, t_2, t_3) = f(u_1, u_2, u_3) \) exactly when terms \( t_1 = u_1 \), etc.
- If we have a variable, say \( X = t \) (where \( t \) is a term), then we will add the substitution \( [X \mapsto t] \) to our set of substitutions. (This is true even if \( t \) is also a variable.)
  - However, before adding it to the set of substitutions, we apply \( [X \mapsto t] \) to all the substitutions we've built up so far and also to all the equations we're still trying to solve. We'll discuss this more in a moment.
  - If we don't have a variable, the only cases remaining can't unify because the two sides of the equation involve different kinds of terms. E.g., we might have a \( xyz = 17 \), which fails to unify, so the whole attempt fails.

**Before adding a new substitution to our solution**

- If we want to add a new substitution \( [X \mapsto t] \) to our solution, we have to apply it to all the substitutions we've built up so far and also to all the equations we're still trying to solve.
  - Applying \( [X \mapsto t] \) to the equations means applying it to each pair of terms in the problem set.
  - Applying \( [X \mapsto t] \) to the substitutions means applying it to the new term of each substitution.
• E.g., say we have the problem set \( \{ X \equiv 2, f(X) = f(Y), g(X) = g(X) \} \) and we currently have \( \{ \[ W \mapsto g(X) \] \} \) as our set of substitutions so far. The equation \( X \equiv 2 \) tells us to add the substitution \( \[ X \mapsto 2 \] \) to our solution but before we do that, we transform the (reduced) problem set \( \{ f(X) = f(Y), g(X) = g(X) \} \) to \( \{ f(2) = f(Y), g(2) = g(2) \} \) and the substitution set \( \{ \[ W \mapsto g(X) \] \} \) to \( \{ \[ W \mapsto g(2) \] \} \). Then we add \( \[ X \mapsto 2 \] \) to the substitutions and get \( \{ \[ W \mapsto g(2) \], \[ X \mapsto 2 \] \} \).

• This is done so that the leftover equations won't have any more occurrences of \( X \) to work on and so that the terms introduced by the substitutions won't introduce any occurrences of \( X \).

• If we left around uses of \( X \) in the problem, we'd have to use the \( X \equiv 2 \) substitution anyway to make sure that the uses of \( X \) are compatible. E.g., we'd eventually have to turn \( f(X) = f(Y) \) into \( f(2) = f(Y) \) in order to figure out that \( Y \equiv 2 \); we might as well do it now.

• The same argument applies to the substitution \( \[ W \mapsto g(X) \] \). We don't have a \( W \) in the problem any more because we removed it before we added \( \[ W \mapsto g(X) \] \) to our solution. E.g., say we originally had \( W \equiv g(X) \), so we changed it to the \( g(X) = g(X) \) that currently appears in the problem. If we leave \( g(X) = g(X) \) to solve later, we'll just have to change it to \( g(2) = g(2) \) anyway so we might as well do it now.

The Occurs Check

• [Also see Learn Prolog Now 2.1: Unification]

• There’s one situation where applying a new substitution could get us into trouble.

• Take the equation \( X \equiv f(X) \). If we just add \( \[ X \mapsto f(X) \] \) to our substitution list, then our eventual solution set of substitutions won’t remove all \( X \)’s when applied to the original problem.

• If we try to expand \( X \equiv f(X) \) to get rid of the \( X \) on the rhs of the equation, the only solution involves having an infinite number of \( f \)’s: \( X \equiv f(f(f(\ldots))) \). Clearly, we can’t build that whole term when trying to solve our problem because it’s an infinitely large term\(^1\).

• If we want to make sure we don’t try to build an infinite terms, before we add a substitution \( \[ X \mapsto t \] \) to our solution, we would have to do an occurs check: We would inspect \( t \) to make sure it doesn’t have any \( X \)’s in it.

• This process takes time linear in the size of \( t \), so depending on how many terms our program has, the total time spent doing occurs checks might be large.

• The usual thing done is to not do an occurs check and trust the programmer to not build infinite terms.

• SWI Prolog treats the equation \( X \equiv f(X) \) as its own solution. (Try running it in an SWI Prolog session.)

Proving Goals in Prolog — The Resolution Principle

• The general process for proving some compound term like \( g(t) \) is

---

\(^1\) A lazy language like Haskell can handle infinitely long terms: We do have to build them lazily and only work with some finite initial segment of the term, as in \( \text{ones} = 1 : \text{ones} \), which in Prolog terms might be \( \text{ones} = \text{cons}(1, \text{ones}) \).
• Step 1: Search the database for \( g(t) \) as a fact. More exactly, look for a fact that unifies with \( g(t) \). E.g., if we want to solve \( g(X) \) and find the fact \( g(12) \) in the database, then our search ends successfully with \( X \equiv 12 \) as the solution.

• Step 2: If \( g(t) \) doesn't match a fact, search through the database for a rule whose goal matches (i.e., unifies with) \( g(t) \). Say the rule is \( g(u) :- p_1, p_2, ..., p_n \). Then continue the proof process, but with \( p_1, p_2, ..., p_n \) as our list of goals.

• In the general case, we're trying to prove a list of goals \( g_1, g_2, ..., g_m \). We first try to prove \( g_1 \) as above, first as a fact, then as the goal of a rule \( g :- p_1, p_2, ..., p_n \). In that case, we remove \( g_1 \) from our list of goals and replace it with \( p_1, p_2, ..., p_n \). Our overall list of goals is now \( p_1, p_2, ..., p_n, g_2, ..., g_m \), which might be longer, but presumably the \( p \)'s will be easier to prove.

• This process of replacing a goal with some (hopefully) simpler goals is called the **Resolution Principle**.

**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).

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

  • (Assuming \( X \) 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.

  • Similarly, the query \( X \) is \( 2 + 2 \) calculates \( 2 + 2 \) and binds \( X \) to \( 4 \). The is operator isn't symmetric: for \( e_1 \) is \( e_2 \), Prolog evaluates \( e_2 \) and tries to unify the result with (the unevaluated) \( e_1 \).

  • So, \( 2 + 2 \) is \( X \) never succeeds. (Let \( e_1 \) be \( 2 + 2 \) and \( e_2 \) be \( X \).)

    • If \( X \) is not instantiated, then evaluating \( e_2 \) 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 probably don't want \( = \) because it tries to unify its two sides (\( 2 + 2 = 4 \) fails). (There's also a \( \nabla = \) operator which means “doesn't unify with”: \( 2 + 2 \nabla = 4 \) succeeds, but so does \( 2 + 2 \nabla = 5 \).)

  • For arithmetic equality, the operators are \( \equiv \) and \( \equiv \). (The equal signs on both sides are mnemonic that both operands get evaluated.) E.g., \( 2 + 2 \equiv 4 \) and \( 2 + 2 \equiv 5 \) are both true.

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