Prolog, pt 2: Unification; Resolution; Arithmetic

CS 440: Programming Languages and Translators, Spring 2020

4/9: posted

1. Sources for Study[1]

- Read Chapters 1 and 2 of *Learn Prolog Now*: www.learnprolognow.org/
- *SWI Prolog*: http://www.swi-prolog.org/

2. Review

- Prolog is a programming language based on logic.
  - Its core is as a **declarative** language: Instead of writing a program, you write out the specification for what a legal output looks like.
  - To execute a query (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). (Plus cuts, later today.)
- To form predicates, Prolog builds up from primitive predicates $P(a_1, a_2, \ldots, a_n)$, (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 `:-` $p_1, p_2, \ldots, p_n$`.` 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.
    - E.g,, `mammal(X) :- dog(X).` % if X is a dog then X is a mammal; i.e., all dogs are mammals.

3. Atoms

- Not mentioned last time (see Ch. 1 of Learn Prolog Now)
- Atoms are basically symbols
  - Named constants like `x_yz` or `'xy z'` (spaces are ok if in single quotes). First symbol is lower-case 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.

[1] I also used *Programming Language Pragmatics*, 4th ed., Michael L. Scott as a reference.

4. Unification in Prolog

- The same as unification in Lecture 16 and in the Final Project.[2]
  - Textual operation where we try to match terms built up from constants, numerals, variables, compound terms (have a "functor" name and parenthesized arguments, as in `f(a, b)`).
  - (Not seen yet: Prolog also has lists.)
- Terms without variables have to match exactly: `f(1, 2)` ≡ `f(1, 2)` but ≡ `f(2, 1)`.
- Variables inside terms may have to be given values to get matching ("instantiating" variables). These might be ground terms (with no variables) or might include other variables.
  - E.g., `f(X, Y)` ≡ `f(g(z), h(W))` by using `X` ≡ `g(z)` and `Y` ≡ `h(W)`.
- 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, 5) ≡ f(5, 3) is not.
- Matching is syntactic, not semantic: 2+2 ≢ 4 because they are different as pieces of text.

Unification Algorithm

- Review of the unification algorithm from Lecture 16:
- 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` ≢ `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`(*t1*, *t2*) ≡ `f`(*u1*, *u2*), we need *t1* ≡ *u1* and *t2* ≡ *u2*. (The *t*'s and *u*'s are italicized to help make it clearer that they stand for terms; they aren't terms themselves.)
    - Similarly, `f`(*t1*, *t2*, *t3*) ≡ `f`(*u1*, *u2*, *u3*) exactly when *t1* ≡ *u1*, etc.
  - If we have a variable, say `X` ≡ *t* (where *t* is a term), then we add a substitution [`X` ↦ *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` ↦ *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.

---

[2] Don't forget to look at the final project — the parser should be pretty easy to write. Pretty printing is annoying but not deep, conceptually. Unification is the hardest part.

Before adding a new substitution to our solution

- Recall that if we wanted to add a new substitution [X ↦ *t*] to our solution, we had 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 ↦ *t*] to the equations means applying it to each pair of terms in the problem set.
- Applying [X ↦ *t*] to the substitutions means applying it to the new term of each substitution.
- E.g., say we have the problem set {X ≡ 2, f(X) ≡ f(Y), g(X) ≡ g(X)} and we currently have {[W ↦ g(X)]} as our set of substitutions so far. The equation X ≡ 2 tells us to add the substitution [X ↦ 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 ↦ g(X)]} to {[W ↦ g(2)]}. Then we add [X ↦ 2] to the substitutions and get {[W ↦ g(2)], [X ↦ 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 ≡ 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 ≡ 2; we might as well do it now.
- The same argument applies to the substitution [W ↦ g(X)]. We don't have a W in the problem any more because we removed it before we added [W ↦ g(X)] to our solution. E.g., say we originally had W ≡ 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 (see Chapter 2 of Learn Prolog Now)

- There's one situation where applying a new substitution could get us into trouble.
- Take the equation X ≡ f(X). If we just add [X ↦ 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 ≡ 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 ≡ f(f(f(…))). Clearly, we can't build that whole term when trying to solve our problem because it's an infinitely large term.
  - Note Haskell can handle infinitely long terms as long as we build them lazily and only work with some finite initial segment of the term, as in `ones = 1 : ones`, which in Prolog terms might be `ones = cons(1, ones)`.
- If we want to make sure we don't try to build an infinite terms, before we add a substitution [X ↦ *t*] to our solution, we would have to do an **occurs check**: We'd 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 = 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
  - 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` ≡ `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*`) :-` $p_1, p_2, \ldots, p_n$. 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 $g_1, g_2, \ldots g_m$. We first try to prove g_1 as above, first as a fact, then as the goal of a rule g `:-` $p_1, p_2, \ldots, p_n$. In that case, we remove g_1 from our list of goals and replace it with $p_1, p_2, \ldots, p_n$. Our overall list of goals is now $p_1, p_2, \ldots, p_n, g_2, \ldots g_m$, 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

- Arithmetic terms are written in the usual way in Prolog (numerals, `+`, `-`, `*`, `/`, parentheses).
- There is a difference between unifying with an expression and evaluating an expression.
- The query `X = 2+2` succeeds and binds `X` to literally `2+2` because = means unification, which is a textual operation.
- The query `X is 2+2` also succeeds but calculates `2+2` and binds `X` to the result, `4`.
  - The expression to calculate must be on the right; the variable must be to the left.
  - The query `2+2 is X` fails with a message about `X` not being instantiated (and therefore doesn't look like an expression that Prolog can evaluate. On the other hand, `X = 2+2`, `Y is X` succeeds because it instantiates `X` to an expression and then evaluates the expression to yield `Y`.
    with `X = 2+2` and `Y = 4`, since X is instantiated to an expression.
  - You can have a numeral constant to the left of `is`, but not a complicated expression. E.g., `4 is 2+2` succeeds but `2+2 is 2+2` fails because the right hand 2+2 evaluates to `4`, which does not unify with the left-hand `2+2`.
- **Comparisons**
  - You can compare arithmetic expressions using and <, =<, >, and >= (where =< means ≤). Both sides of the operator are evaluated, e.g., `2+2 < 7*3` yields `true`.
  - For arithmetic equality, you probably don't want = or `\=` because these are unification operations, e.g., `2+2 \= 4` is `false` and `2+2 \= 4` is `true`.
  - For arithmetic equality, the operators are `=:=` and `=\=` . (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.

Activity Problems for Lecture 22

- To be posted.

