Lecture 38: Declarative Programming (Under the Hood)

Announcements:

- Autograder running. As we fix glitches, expect multiple reports.
- Remember: you still have to provide your own tests!
- Submit your Project 4 contest entries as "`proj4-contest`." by next Wednesday. Assuming we get entries, we'll ask the class to judge these entries.
- Penultimate homework (13) to be released late tonight(?). Due date to be set appropriately.
- "Homework" 14 will be judging the contest.

Review: A "Schemish" Prolog

- A Scheme expression, e.g. `(ordered (0 1 2))` represents a logical assertion.
  - Its top-level operator (e.g., `ordered`) names a *predicate* (true/false function).
  - Its operands are the data for this predicate: unlike Scheme programs, they don't represent function calls—they are the literal data...
  - ...with the exception that *logical variables,* represented as symbols starting with underscore, stand for operands that may be replaced by other expressions.
- To define a predicate, we give rules for it:

  `(fact` *CONCLUSION)* means that *CONCLUSION* is to be taken as true, for any replacement of its logical variables.

  `(fact` *CONCLUSION HYPOTHESIS ...)* means that *CONCLUSION* is to be taken as true, assuming that the *HYPOTHESES* can all be shown to be true. Again, this is for all replacements of logical variables throughout the rule.

Review: Operational and Declarative Meanings

- Thus,

  ```
  (fact (eats _P _F) (hungry _P) (has _P _F) (likes _P _F))
  ```

  means that for any replacement of `_P` (e.g., 'brian') and `_F` (e.g., 'potstickers') throughout the rule:

  Declarative Meaning If brian is hungry and has potstickers and likes potstickers, then brian will eat potstickers.

  Operational Meaning To show that brian will eat potstickers, show that brian is hungry, then that brian has potstickers, and then that brian likes potstickers.

- The *declarative meaning* allows us to look at our Scheme-Prolog program as a logical specification of a problem for which the system is to find a solution.
- The *operational meaning* allows us to look at our Scheme-Prolog specification as an executable program for searching for a solution.
- *Closed Universe Assumption:* We make only positive statements. The closest we come to saying that something is false is to say that we can't prove it.

Review: Relations, not Functions

- We've "logified" functions. Instead of saying

  "the value of `(abs -5)` is `5`,"

  we recast the statement as

  "the value `5` stands in the 'absolute value of' relation to `-5`: `(abs -5 5)`."

- Given a value, `-5`, we can ask for its absolute value with a logical variable and then use it elsewhere with the help of logical variables:

  ```
  (?  (abs -5 _X) (add _X 4 _Y))
  ```

  specifies a replacement for `_Y` that makes it equal to `4+|-5|`.

How It's Done (I): Unification

- In general, our system, given a target expression involving a predicate to prove, must find a fact that might assert that target, given a suitable replacement of logical variables.
- To do this, we try to pattern-match the conclusions of all our facts against the target expression.
- The pattern matching is called *unification*, [J. A. Robinson].
- For example, we say that `(likes brian potstickers)` *unifies with* the expression `(likes _P _F)`, if we substitute `brian` for `_P` and `potstickers` for `_F`.
- Might think of this substitution—called a *unifier*—as a Python dictionary mapping logical variables to expressions.

Unification (II)

- The substitution has to be uniform:
  - Can unify `(le 0 1)` with `(le _X _Y)`
  - But cannot unify `(le 0 1)` with `(le _X _X)`
- Everything is symmetric: if A unifies with B, then B unifies with A. Logical variables can appear in one or both.
- It is possible for logical variables to be unified with each other:

  Unify `(likes _P _F)` with `(likes _Q potstickers)`.
- We substitute `potstickers` for `_F`, and choose either to substitute `_Q` for `_P` or vice-versa.
- The result in either case means that any person likes potstickers.

Implementing Unification

A simple tree recursion with side-effects:

```
def unify(E0, E1, env):
    """Returns True iff E0 and E1 can be unified by an extension
    of ENV.  ENV is modified to provide a suitable unifier."""
    def unify1(E0, E1):
        E0 = binding(E0, env); E1 = binding(E1, env)
        if scm_eqvp(E0, E1): return True
        if is_logical_var(E0):
            env[E0] = E1
            return True
        elif is_logical_var(E1):
            env[E1] = E0
            return True
        elif E0.atomp() or E1.atomp(): return False
        else:
            return unify1(E0.car, E1.car)
            and unify1(E0.cdr, E1.cdr)

    return unify1(E0, E1)
```

Using Unification to Search for Proofs

- The process of attempting to demonstrate an assertion (answer a query) is a systematic depth-first search of facts.

```
def query(targets, env):
    for fact in fact database:
        unify conclusion of fact with first target, \
           extending env
        if this succeeds:
           if query(hypotheses of fact, env) and \
              query(rest of targets, env):
                Success! return resulting env
    if we fail at any point, back up, undo changes to env and \
        try another fact.
```

Actual Code

```
def query(clauses, env):
    if scm_nullp(clauses):
        yield env
    else:
        for fact in facts_db:
            fact = fact.freshen({})
            env_head = EnvironFrame(env)
            if unify(fact.car, clauses.car, env_head):
                for env_rule in query(fact.cdr, env_head):
                    for r in query(clauses.cdr, env_rule):
                        yield r
```