CS 61A Lecture Notes Week 15

Topic: Logic programming

Reading: Abelson & Sussman, Section 4.4.1–3

This week's big idea is *logic programming* or *declarative programming.*

It's the biggest step we've taken away from expressing a computation in hardware terms. When we discovered streams, we saw how to express an algorithm in a way that's independent of the *order* of evaluation. Now we are going to describe a computation in a way that has no (visible) algorithm at all!

We are using a logic programming language that A&S implemented in Scheme. Because of that, the notation is Scheme-like, i.e., full of lists. Standard logic languages like Prolog have somewhat different notations, but the idea is the same.

All we do is assert facts:

```
> (load "~cs61a/lib/query.scm")
> (query)

;;; Query input:
(assert! (Brian likes potstickers))
```

and ask questions about the facts:

```
;;; Query input:
(?who likes potstickers)

;;; Query results:
(BRIAN LIKES POTSTICKERS)
```

Although the assertions and the queries take the form of lists, and so they look a little like Scheme programs, they're not! There is no application of function to argument here; an assertion is just data.

This is true even though, for various reasons, it's traditional to put the verb (the *relation*) first:

```
(assert! (likes Brian potstickers))
```

We'll use that convention hereafter, but that makes it even easier to fall into the trap of thinking there is a *function* called `likes`.

• Rules. As long as we just tell the system isolated facts, we can't get extraordinarily interesting replies. But we can also tell it *rules* that allow it to infer one fact from another. For example, if we have a lot of facts like

```
(mother Eve Cain)
```

then we can establish a rule about grandmotherhood:

```
(assert! (rule (grandmother ?elder ?younger)
               (and (mother ?elder ?mom)
                    (mother ?mom ?younger) )))
```

The rule says that the first part (the conclusion) is true *if* we can find values for the variables such that the second part (the condition) is true.

Again, resist the temptation to try to do composition of functions!

```
(assert! (rule (grandmother ?elder ?younger)         ;; WRONG!!!!
               (mother ?elder (mother ?younger)) ))
```

`Mother` isn't a function, and you can't ask for the mother of someone as this incorrect example tries to do. Instead, as in the correct version above, you have to establish a variable (`?mom`) that has a value that satisfies the two motherhood relationships we need.

In this language the words `assert!`, `rule`, `and`, `or`, and `not` have special meanings. Everything else is just a word that can be part of assertions or rules.

Once we have the idea of rules, we can do real magic:

```
;;;;;                          In file cs61a/lectures/4.4/logic-utility.scm
(assert! (rule (append (?u . ?v) ?y (?u . ?z))
               (append ?v ?y ?z)))

(assert! (rule (append () ?y ?y)))
```

(The actual online file uses a Scheme procedure `aa` to add the assertion. It's just like saying `assert!` to the query system, but you say it to Scheme instead. This lets you `load` the file. Don't get confused about this small detail—just ignore it.)

```
;;; Query input:
(append (a b) (c d e) ?what)

;;; Query results:
(APPEND (A B) (C D E) (A B C D E))
```

So far this is just like what we could do in Scheme.

```
;;; Query input:
(append ?what (d e) (a b c d e))

;;; Query results:
(APPEND (A B C) (D E) (A B C D E))

;;; Query input:
(append (a) ?what (a b c d e))

;;; Query results:
(APPEND (A) (B C D E) (A B C D E))
```

The new thing in logic programming is that we can run a "function" backwards! We can tell it the answer and get back the question. But the real magic is...

```
;;; Query input:
(append ?this ?that (a b c d e))

;;; Query results:
(APPEND () (A B C D E) (A B C D E))
(APPEND (A) (B C D E) (A B C D E))
(APPEND (A B) (C D E) (A B C D E))
(APPEND (A B C) (D E) (A B C D E))
(APPEND (A B C D) (E) (A B C D E))
(APPEND (A B C D E) () (A B C D E))
```

We can use logic programming to compute multiple answers to the same question! Somehow it found all the possible combinations of values that would make our query true.

How does the `append` program work? Compare it to the Scheme `append`:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b)) ))
```

Like the Scheme program, the logic program has two cases: There is a base case in which the first argument is empty. In that case the combined list is the same as the second appended list. And there is a recursive case in which we divide the first appended list into its `car` and its `cdr`. We reduce the given problem into a problem about appending `(cdr a)` to `b`. The logic program is different in form, but it says the same thing. (Just as, in the grandmother example, we had to give the mother a name instead of using a function call, here we have to give `(car a)` a name—we call it `?u`.)

Unfortunately, this "working backwards" magic doesn't always work.

```
;;;;;                          In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (reverse (?a . ?x) ?y)
               (and (reverse ?x ?z)
                    (append ?z (?a) ?y) )))

(assert! (reverse () ()))
```

This works for `(reverse (a b c) ?what)` but not the other way around; it gets into an infinite loop. We can also write a version that works *only* backwards:

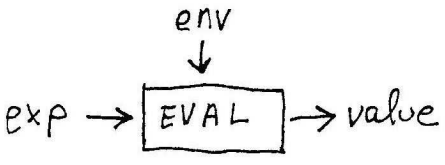
```
;;;;;                          In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (backward (?a . ?x) ?y)
               (and (append ?z (?a) ?y)
                    (backward ?x ?z) )))

(assert! (backward () ()))
```

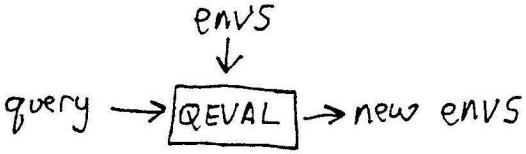
But it's much harder to write one that works both ways. Even as we speak, logic programming fans are trying to push the limits of the idea, but right now, you still have to understand something about the below-the-line algorithm to be confident that your logic program won't loop.

• Below-the-line implementation.

Think about `eval` in the MC evaluator. It takes two arguments, an expression and an environment, and it returns the value of the expression.

In logic programming, there's no such thing as "the value of the expression." What we're given is a query, and there may or may not be some number of variable bindings that make the query true. The query evaluator `qeval` is analogous to `eval` in that it takes two arguments, something to evaluate and a context in which to work. But the thing to evaluate is a query, not an expression; the context isn't just one environment but a whole collection of environments—one for each set of variable values that satisfy some previous query. And the result returned by `qeval` isn't a value. It's a new collection of environments! It's as if `eval` returned an environment instead of a value.



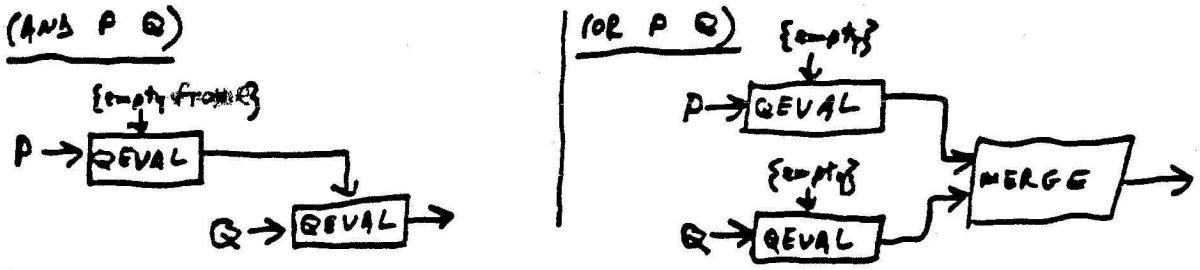


The "collection" of environments we're talking about here is represented as a stream. That's because there might be infinitely many of them! We use the stream idea to reorder the computation; what really happens is that we take one potential set of satisfying values and work it all the way through; then we try another potential set of values. But the program looks as if we compute all the satisfying values at once for each stage of a query.

Just as every top-level Scheme expression is evaluated in the global environment, every top-level query is evaluated in a stream containing a single empty environment. (No variables have been assigned values yet.)

If we have a query like `(and p q)`, what happens is that we recursively use `qeval` to evaluate `p` in the empty-environment stream. The result is a stream of variable bindings that satisfy `p`. Then we use `qeval` to evaluate `q` in that result stream! The final result is a stream of bindings that satisfy `p` and `q` simultaneously.

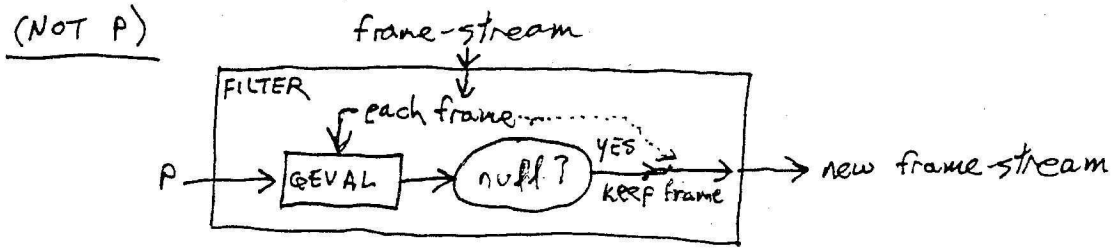
If the query is `(or p q)` then we use `qeval` to evaluate each of the pieces independently, starting in both cases with the empty-environment stream. Then we *merge* the two result streams to get a stream of bindings that satisfy either `p` or `q`.



If the query is `(not q)`, we can't make sense of that unless we already have a stream of environments to work with. That's why we can only use `not` in a context such as `(and p (not q))`. We take the stream of environments that we already have, and we `stream-filter` that stream, using as the test predicate the function

```
(lambda (env) (stream-null? (qeval q env)))
```

That is, we keep only those environments for which we *can't* satisfy `q`.



That explains how `qeval` reduces compound queries to simple ones. How do we evaluate a simple query? The first step is to *pattern match* the query against every assertion in the data base. Pattern matching is just like the recursive `equal?` function, except that a variable in the pattern (the query) matches anything in the assertion. (But if the same variable appears more than once, it must match the same thing each time. That's why we need to keep an environment of matches so far.)

The next step is to match the query against the *conclusions* of rules. This is tricky because now there can be variables in both things being matched. Instead of the simple pattern matching we have to use a more complicated version called *unification*. (See the details in the text.) If we find a match, then we take the condition part of the rule (the body) and use that as a new query, to be satisfied within the environment(s) that `qeval` gave us when we matched the conclusion. In other words, first we look at the conclusion to see whether this rule can possibly be relevant to our query; if so, we see if the conditions of the rule are true.

Here's an example, partly traced:

```
;;; Query input:
(append ?a ?b (aa bb))

(unify-match (append ?a ?b (aa bb))         ; MATCH ORIGINAL QUERY
             (append () ?1y ?1y)            ; AGAINST BASE CASE RULE
             ())                            ; WITH NO CONSTRAINTS

RETURNS: ((?1y . (aa bb)) (?b . ?1y) (?a . ()))
PRINTS:  (append () (aa bb) (aa bb))
```

Since the base-case rule has no body, once we've matched it, we can print a successful result. (Before printing, we have to look up variables in the environment so what we print is variable-free.)

Now we unify the original query against the conclusion of the other rule:

```
(unify-match (append ?a ?b (aa bb))                ; MATCH ORIGINAL QUERY
             (append (?2u . ?2v) ?2y (?2u . ?2z))  ; AGAINST RECURSIVE RULE
             ())                                   ; WITH NO CONSTRAINTS

RETURNS: ((?2z . (bb)) (?2u . aa) (?b . ?2y) (?a . (?2u . ?2v)))
         [call it F1]
```

This was successful, but we're not ready to print anything yet, because we now have to take the body of that rule as a new query. Note the indenting to indicate that this call to `unify-match` is within the pending rule.

```
    (unify-match (append ?2v ?2y ?2z)   ; MATCH BODY OF RECURSIVE RULE
                 (append () ?3y ?3y)    ; AGAINST BASE CASE RULE
                 F1)                    ; WITH CONSTRAINTS FROM F1

    RETURNS: ((?3y . (bb)) (?2y . ?3y) (?2v . ()) [plus F1])
    PRINTS:  (append (aa) (bb) (aa bb))

    (unify-match (append ?2v ?2y ?2z)                  ; MATCH SAME BODY
                 (append (?4u . ?4v) ?4y (?4u . ?4z))  ; AGAINST RECURSIVE RULE
                 F1)                                   ; WITH F1 CONSTRAINTS

    RETURNS: ((?4z . ()) (?4u . bb) (?2y . ?4y) (?2v . (?4u . ?4v))
              [plus F1])  [call it F2]

        (unify-match (append ?4v ?4y ?4z)   ; MATCH BODY FROM NEWFOUND MATCH
                     (append () ?5y ?5y)    ; AGAINST BASE CASE RULE
                     F2)                    ; WITH NEWFOUND CONSTRAINTS

        RETURNS: ((?5y . ()) (?4y . ?5y) (?4v . ()) [plus F2])
        PRINTS:  (append (aa bb) () (aa bb))

        (unify-match (append ?4v ?4y ?4z)                  ; MATCH SAME BODY
                     (append (?6u . ?6v) ?6y (?6u . ?6z))  ; AGAINST RECUR RULE
                     F2)                                   ; SAME CONSTRAINTS

        RETURNS: ()                                        ; BUT THIS FAILS
```