LOGIC EVALUATOR: BELOW THE LINE 27

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

August 9, 2010

To load the logic evaluator, say:

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

1 Review of Above the Line

Take a look at the following problem from Practice Final #1 in the Reader. One technique for writing some logic programs is Translation:

```
(define (depth lst)
  (if (not (pair? lst))
      0
      (max (+ 1 (depth (car lst))) (depth (cdr lst)))))
```

If we now do not allow composition of functions except for `cons`, `car`, `cdr`, and `+`:

```
(define (depth lst)
  (if (not (pair? lst))
      0
      (let* ((dcar (depth (car lst)))
             (dcdr (depth (cdr lst)))
             (result (max (+ 1 dcar) dcdr)))
            result)))
```

There's now a natural translation towards Logic:

2 Below the Line

Disclaimer: For the purposes of the final, you are by and large not responsible for any material in this section. You are responsible for above the line, and this lecture serves to demystify that.

2.1 Introduction

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**.

2.2 Pattern Matching

For example, if we did something like:

```
(match '(?a 'hello ?b) '(well hello there))
```

We'd see that ?a is bound to well and ?b is bound to there. On the other hand, if we tried a query like:

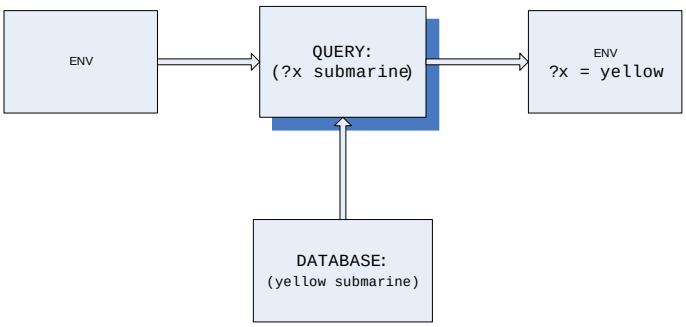
```
(match '(?a hello ?b) '(yellow submarine)
```

We'd see that this would fail.

The details on how pattern matching is implemented is outside the scope of this course, not because it's hard, but because it's not important. In short, use your human intuition, and you can see what the input and outputs should be. Feel free to ignore the internals.

Let's look at first the 'simple' query. This is one that will bind a query variable with a value. In other words, only one of the sides will have variables. The first step is to PATTERN MATCH the query against every assertion in the data base. Pattern matching is just like a 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.

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.) Thus, a query looks something like:

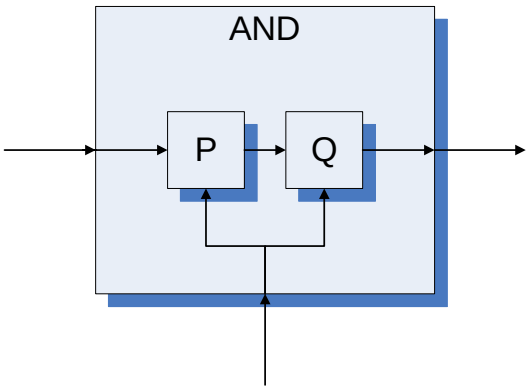


2.3 Complex Queries

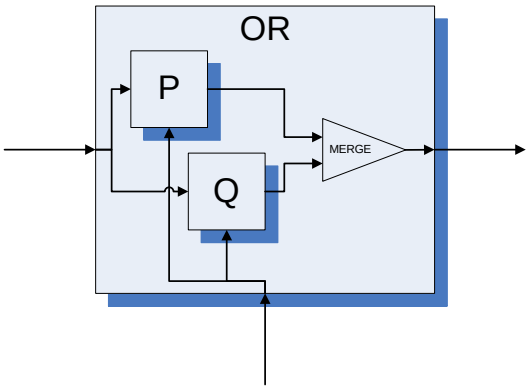
The "collection" of environments we talked about is represented as a stream. That's because there might be infinitely many of them! We use the stream idea to reorder the computation; what really happens is that we take one potential set of satisfying values and work it all the way through; then we try 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.

2.3.1 AND/OR

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. Remember, each query is capable of either adding a binding to a frame, or 'destroying' the frame, if nothing can be satisfied.



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`.

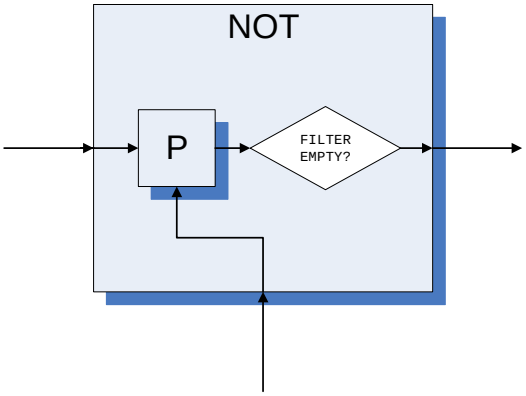


2.3.2 NOT

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`.



There's one important thing to note about what `not` really means. It is very different in particular, from the logical version of it. This version really deals with what can or cannot be deduced, as opposed to what is or is not true. In particular, it assumes that everything that it doesn't know about is false. You can see why that would be dangerous.

That explains how `qeval` reduces compound queries to simple ones. Notice in each case, our complex queries look outwardly the same as a simple query. We still input a stream of frames, as well as a database, and we output a stream of frames with some operation done on them.

2.4 Unification

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.

For example, if we wish to do something like:

```
(unify (?x ?x) ((a ?y c) (a b ?z))
```

We'd obtain:

```
?x = (a b c)
?y = b
?z = c
```

That wasn't bad. What about:

```
(unify (?x ?x) ((?y a ?w) (b ?v ?z))
```

Well...that's a bit tougher.

```
?y = b
?v = a
?w = ?z
?x = (b a ?w)
```

The best way to think about this is that it does it's best to merge two frames together. It's a lot like solving a linear system of equations.

But what about:

```
(unify (?x ?x) (?y (a . ?y))
```

Well, as humans, we can see:

```
?x = y
?y = (a a a a a ...)
```

Thus, the general case is harder. Basically, the program includes a check that will force the program to give up in some cases.

2.5 Quirks

By giving up all the control over the flow, we also potentially create problems. We'll see computation fail in Above the Line Logic programs because of Below the Line implementation.

Which of the following 2 works? Why?

```
(rule (outranked-by ?s ?b)
  (or (supervisor ?s ?b)
      (and (supervisor ?s ?m)
           (outranked-by ?m ?b))))
(rule (outranked-by ?s ?b)
  (or (supervisor ?s ?b)
      (and (outranked-by ?m ?b)
           (supervisor ?s ?m))))
```

Now let's look at more classical logic, and look at what I said earlier about `not`. Let's look at the Greeks!

```
(Greek Socrates) (Greek Plato)
(Greek Zeus)     (god Zeus)
```

```
(rule (mortal ?x)   (human ?x))
(rule (fallible ?x) (human ?x))

(rule (human ?x)
  (and (Greek ?x) (not (god ?x))))

(rule (address ?x Olympus)
  (and Greek ?x (god ?x)))
```

Again, let's extend this to include perfect beings:

```
(rule (perfect ?x)
  (and (not (mortal ?x))
       (not (fallible ?x))))
```

Now:

```
(and (address ?x ?y)
    (perfect ?x))
(and (perfect ?x)
     (address ?x ?y))
```

2.6 Example:

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
```