LOGIC PROGRAMMING 25

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

August 4, 2010

1 Introduction

A computer does *exactly* what you tell it to do. This is one of the main reasons people get frustrated with their code or with their computer: the computer is doing exactly what you asked it to do, which can be entirely different from what you meant it to do. When mathematicians define, for example, the square root of a real number, they don't talk about how to *find* the square root; they talk about how the square root of a number x is the positive number y such that the square of y is x. A&S gives an implementation in pseudo-Scheme (pg. 22):

```
(define (sqrt x)
   (the y (and (>= y 0)
               (= (square y) x))))
```

We would like for the program above to work, but it does not quite work because we're not telling the computer how exactly to pick the `y`. We could potentially loop through all the possible integers and check to see if they satisfy the condition, but at this point in this course, you can instantly tell that this method is very inefficient. There are, of course, methods to find the square root of a number, such as Newton's method, but again, these methods tell the computer exactly *how to* find the square root of a number; they don't tell the computer *what* the square root of a number *is*.

Up until this point in the course, we have dealt with *imperative programming*, thinking about *how to* do things. Today's big idea is *logic programming* or *declarative programming*, thinking about *what* things *are*. It is the biggest step we have taken away from expressing a computation in hardware terms. When we discovered streams, we saw how to express an algorithm in a way that is independent of the order of evaluation. Now, we are going to describe a computation in a way that has no (visible) algorithm at all!

2 Logic Evaluator

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! (Jon likes omnom))
```

and ask questions about the facts:

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

;;; Query results:
(JON LIKES OMNOM)
```

Although the assertions and the queries take the form of lists, and so they look a little like Scheme programs, *they are not*! There is no application of function to argument here; an assertion is just data. This is true even though, for various reasons, it is traditional to put the verb (the relation) first:

```
(assert! (likes Jon omnom))
```

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

Notice that the language does an exact *pattern match*. As a result, if we add the following assertion:

```
;;; Query input:
(assert! (Jon likes om nom))
```

and ask questions about it:

```
;;; Query input:
(assert! (Jon likes ?what))

;;; Query results:
(JON LIKES OMNOM)
```

we only get the one assertion (so far) that matches the pattern exactly. To get all of the assertions, we use Scheme's dotted-tail notation, where everything after the dot is packed into one list.

```
;;; Query input:
(assert! (Jon likes . ?what))

;;; Query results:
(JON LIKES OM NOM)
(JON LIKES OMNOM)
```

Here, we are asking the query system for assertions with at *least* two words: `jon likes`.

Also notice that the system can only make assertions about what you tell it. Jon could like a ton of other stuff, but the system will not tell you about them because it does not know about these likes. (Thankfully.)

3 Rules

As long as we just tell the system isolated facts, we can not 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. In other words, we make a *logical inference* – if the *condition* is true, then we make a *conclusion*. In the above rule, we are encoding the logical inference that, if `?mom` is the mother of `?younger`, and if `?elder` is the mother of `?mom`, then it must be true that `?elder` is the grandmother of `?younger`.

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

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

`mother` is *not* a function, and you can not 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 is just like saying `assert!` to the query system, but you say it to Scheme instead. This lets you load the file. Do not 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. However...

```
;;; 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 does not 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. We will talk about the below-the-line implementation on Monday.