SICP section 4.4

February 07, 2008 at 22:14 Tags SICP

Logic programming is a major topic and this chapter isn't easy to understand at first reading, especially if you're not already familiar with some of the material. As usual, I'm first reading the whole section and re-implementing the relevant code in Common Lisp. You can download the implementation here, and the test suite for it here

In this post I want to share a few insights on the implementation, and in particular highlight the points in which my code differs from the authors' code, mainly due to the translation to Common Lisp. I will leave the solutions of exercises for later posts – it should be much more convenient once we have a full working implementation we understand well.

Understanding the query evaluator

My re-implementation of the full query evaluator is close to 600 lines of code (including lots of comments). For Lisp – this is a lot of code[1]. However, not all of this code is equally important for understanding the *core*, and I want to emphasis a few salient points that might make this easier.

First of all, read section 4.4.2. Now, read it again, and once again please. This is the most important section – because it explains how the the query evaluator works, without getting into the nitty gritty details.

Make sure you understand the basic pattern matcher. To *really* understand it, play with its implementation – by providing patterns and data. Control question – why are frames needed at all ?

Now, once you know your pattern matcher rock solid (to prove it, re-implement it in a blank file, without looking back), get to unification. Unification is probably as *core* as you can get in the logic evaluator. If you understand the pattern matcher, you'see that unification is really a very reasonable extension. Read the "Unification" paragraph in section 4.4.2 a couple of times, and then section 4.4.4.4, and play with the unifier function a bit to get a feeling for how it works.

Next, comprehend how processing a query works. First, understand the simple case of a single frame that goes through the query and is matched versus each assertion in the database. The resulting non-failed frames are then viable instantiations[2] of the query, and will be given as the answer. Second, understand the extension of the input to be a stream of frames. Conceptually, this is similar – just repeat the same process of matching for every frame in the input stream, but why is this needed ? Simple – to implement compound queries. See the examples in the paragraph titled "Compound queries" in section 4.4.2 for a good explanation.

This should give you the direction to grasping the code of the query evaluator. If there are still specific points that aren't clear, feel free to post a comment with questions.

Some important points about my implementation

My implementation of the query evaluator is in Common Lisp, and hence has some minute differences from the authors' code, which is in Scheme.

First of all, string handling in Common Lisp is done a bit differently than in Scheme. Here is my `expand-question-mark`:

```
(defun expand-question-mark (symb)
  (let ((chars (string symb)))
    (if (equal (char chars 0) #\?)
      (list '? (intern (remove #\? chars :count 1)))
      symb)))
```

In Common Lisp, characters are objects with their own type, and can be manipulated as such. `string` turns a symbol into a string[3], which can then be examined using the `char` function. Note how I get to the *rest of the string, after the ?* – by using `remove` on `?` with `:count 1`. This works since I know at that point that the string begins with `?`. `intern` then turns the new string into a symbol.

Similarly, note the differences in `contract-question-mark`:

```
(defun contract-question-mark (var)
  (intern
    (concatenate 'string
      "?"
      (if (numberp (cadr var))
        (concatenate 'string
          (string (caddr var))
          "-"
          (write-to-string (cadr var)))
        (string (cadr var))))))
```

`write-to-string` is used to convert numbers to strings, and `concatenate` to stitch several strings together.

Another interesting point I want to explain is the usage of `assoc` in `binding-in-frame`:

```
(defun binding-in-frame (var frame)
    (assoc var frame :test #'equal))
```

Note the peculiar `:test #'equal` – why is it here ? This is an interesting story in bug-hunting, actually. Recall that `expand-question-mark` turns `?x` into the list `(? x)` to as to make pattern matching and unification more efficient. Therefore, bindings in frames are stored with lists as the keys. For example, `(? x)` bound to `joe`, and so on. Therefore, `assoc` must be able to compare list keys. It turns out that the default comparison function for `assoc` in Scheme is `equal?`, which works for lists. On the other hand, in Common Lisp the default for `assoc` is `eql`, which doesn't work for lists ! Therefore, it is essential to provide `equal` as the test function explicitly when calling `assoc`.

Another gotcha that stems from the Scheme-Common Lisp disparity is the usage of `lisp-value`. Since the evaluation of `lisp-value` calls the underlying Lisp's `apply` on the predicate:

```
(defun execute-exp (exp)
  (apply (eval (predicate exp)) (args exp)))
```

The predicate must be `apply`-able. Therefore, when using Common Lisp I must pass a function object. I.e. instead of:

```
(and (salary ?person ?amount)
     (lisp-value > ?amount 30000))
```

For the Scheme implementation the authors use, in my version this should be:

```
(and (salary ?person ?amount)
     (lisp-value #'> ?amount 30000))
```

Because in Common Lisp `#'>` is a function object and `>` isn't.

Finally, as with the Scheme evaluator written eariler in chapter 4, I prefer to have means to interpret expressions in a non-interactive way. So, I wrote `qinterpret`, which is similar to the authors' `query-driver-loop` from section 4.4.4.1, except for all the interactivity code:

```
(defun qinterpret (&rest exps)
  (dolist (exp exps)
    (let ((q (query-syntax-process exp)))
      (cond ((assertion-to-be-added? q)
             (add-rule-or-assertion! (add-assertion-body q)))
            (t
              (display-stream
                (stream-map
                  (lambda (frame)
                    (instantiate
                      q
                      frame
                      (lambda (v f)
                        (contract-question-mark v))))
                  (qeval q (singleton-stream '()))))))))
```

While at it, I added some functionality by allowing `qinterpret` to receive several expressions at the same time, and

interpreting them all. This is useful for providing a whole bunch of assertions for the database:

```
(qinterpret
  '(assert! (address bob (haifa malal 12)))
  '(assert! (address jane (tel-aviv rokah 3)))
  '(assert! (address joe (haifa hertzel 33/1))))
```

That's it. Now, we're ready to tackle the horde of exercises of section 4.4 –*bon voyage!*

[1] You can implement a whole logic programming interpreter with that ! Oh, wait…

[2] An instantiation of the query is simply substituting the variables it contains to the values they're assigned in the frame. For example:

```
(job ?x ?y)
```

Instantiated with the frame where `?x` is set to `joe` and `?y` is set to `(computer programmer)`, results in:

```
(job joe (computer programmer))
```

It can get just a bit more complicated than this, though, since in reality `?x` may be set to `?z` and `?z` to `joe` in the same frame. So, the instantiator works recursively, eventually resolving all variables to values.

[3] I could also use `symbol-name` for this purpose here, but `string` is more generic.

For comments, please send me an email.

Back to top