SICP sections 4.4.2 - 4.4.4

February 09, 2008 at 08:11 Tags SICP

Exercise 4.64

Let's follow the procedure for applying rules described in section 4.4.2, to see what happens when the badly rewritten rule `outranked-by` is applied to the query issued by De Witt Aull:

First, we'll unify the query:

```
(outranked-by (Bitdiddle Ben) ?who)
```

With the conclusion of the rule:

```
(outranked-by ?staff-person ?boss)
```

The unification succeeds with the bindings `?staff-person -> (Bitdiddle Ben)` and `?boss -> ?who`. Now, we will evaluate the query formed by the body of the rule:

```
(or (supervisor ?staff-person ?boss)
    (and (outranked-by ?middle-manager ?boss)
         (supervisor ?staff-person ?middle-manager))))
```

Relative to the extended frame. Consider what happens when we come to evaluate the sub-query `(outranked-by ?middle-manager ?boss)`. Again, this is a query that will match the `outranked-by` rule. Unifying it with the rule's conclusion succeeds, producing the bindings `?staff-person -> ?middle-manager` and `?boss -> ?who`. So we once again evaluate the rule's body, now with these bindings. And again, and again – this is an infinite loop.

Why didn't the original rule have this problem ?

Because the `(supervisor ?staff-person ?middle-manager)` sub-query came first in the `and`, and created bindings that narrowed the search when re-applying the `outranked-by` rule.

Exercise 4.65

Oliver Warbucks is listed four times because the rule matches four times over the database. I.e., there are four middle-managers of whom Oliver Warbucks is a manager. To find out who these are, we can unroll the rule:

```
(and (supervisor ?middle-manager ?person)
     (supervisor ?x ?middle-manager))
=>
(AND (SUPERVISOR (SCROOGE EBEN) (WARBUCKS OLIVER)) (SUPERVISOR (CRATCHET ROBERT) (SCROOGE EBEN)))
(AND (SUPERVISOR (BITDIDDLE BEN) (WARBUCKS OLIVER)) (SUPERVISOR (TWEAKIT LEM E) (BITDIDDLE BEN)))
(AND (SUPERVISOR (HACKER ALYSSA P) (BITDIDDLE BEN)) (SUPERVISOR (REASONER LOUIS) (HACKER ALYSSA P)))
(AND (SUPERVISOR (BITDIDDLE BEN) (WARBUCKS OLIVER)) (SUPERVISOR (FECT CY D) (BITDIDDLE BEN)))
(AND (SUPERVISOR (BITDIDDLE BEN) (WARBUCKS OLIVER)) (SUPERVISOR (HACKER ALYSSA P) (BITDIDDLE BEN)))
```

Exercise 4.66

Suppose that Ben will try to compute the sum of the salaries of the wheels in the organization:

```
(qinterpret
  '(and (wheel ?who)
        (salary ?who ?amount)))
=>
(AND (WHEEL (WARBUCKS OLIVER)) (SALARY (WARBUCKS OLIVER) 150000))
(AND (WHEEL (WARBUCKS OLIVER)) (SALARY (WARBUCKS OLIVER) 150000))
(AND (WHEEL (BITDIDDLE BEN)) (SALARY (BITDIDDLE BEN) 60000))
(AND (WHEEL (WARBUCKS OLIVER)) (SALARY (WARBUCKS OLIVER) 150000))
(AND (WHEEL (WARBUCKS OLIVER)) (SALARY (WARBUCKS OLIVER) 150000))
```

Obviously, if Ben tries to apply the `sum` on `?amount` over these results, he'll get too much because of the duplications.

One way to circumvent this problem is to install a *uniqueness filter* that will filter out all the duplicates from the resulting frames produced by `qeval`.

Exercise 4.67

```
???
```

Exercise 4.68

```
(qinterpret
  '(assert!
    (rule (reverse () ()))))
(qinterpret
  '(assert!
    (rule
      (reverse ?x ?y)
      (and
        (append-to-form (?car) ?cdr ?x)
        (append-to-form ?rev-cdr (?car) ?y)
        (reverse ?cdr ?rev-cdr)))))
```

This rule answers:

```
(qinterpret
  '(reverse ?x (1 2 3)))
=>
(REVERSE (3 2 1) (1 2 3))
```

But not:

```
(qinterpret
  '(reverse (1 2 3) ?x))
=>
*** - Program stack overflow. RESET
```

Exercise 4.69

```
(rule (ends-with-grandson ?x)
      (append-to-form ?head (grandson) ?x))

(rule ((great . ?rel) ?x ?y)
      (and
        (ends-with-grandson ?rel)
        (?rel ?sx ?y)
        (son ?x ?sx)))
```

Exercise 4.70

The problem with this alternative implementation is that `cons-stream` doesn't evaluate its second argument. Therefore, `THE-ASSERTIONS` will point to itself. In the original implementation, the purpose of the `let` was to force an evaluation of `THE-ASSERTIONS`, storing its pre-`set!` value in `old-assertions`.

It took me much longer than expected to solve this, because of one fact that puzzled me. I wrote a few unit tests for the logic evaluator, and ran them with the modified version. Everything worked. This was a surprise because I expected the self reference in the stream of assertions to cause an infinite loop.

The solution of this problem can teach us a lesson: complex systems are difficult to understand, test and debug. What actually happened is that all the assertions in my tests were *indexable*. Note that `store-assertion-in-index` in the implementation of `add-assertion!`, and the complementary `get-indexed-assertions` in `fetch-assertions` ? These do the actual job for most assertions, without ever getting to really searching in the big stream. To float this problem I wrote a few *unindexable* assertions[1]:

```
(assert! ((jay bird) sam 10))
(assert! ((jay bird) tom 12))
```

Indeed, these caused an infinite loop with the alternative implementation when searched. Phew. I'm rarely as glad to see a stack overflow in my program :-)

Exercise 4.71

These delays can postpone infinite looping in some cases, providing more meaningful answers to the user. Consider these assertions:

```
(qinterpret
  '(assert! (dummy a))
  '(assert! (dummy b))
  '(assert! (dummy c))
  '(assert! (rule (dummy ?x) (dummy ?x))))
```

The last one asserts an infinitely-recursive rule. Now, if we issue the query:

```
(qinterpret
  '(dummy ?who))
```

With the delays in place, we'll get:

```
(DUMMY C)
(DUMMY B)
(DUMMY A)
(DUMMY C)
(DUMMY B)
(DUMMY A)
(DUMMY C)
(DUMMY B)
(DUMMY A)
...
...
```

To understand why, look at this portion of `simple-query`:

```
      (stream-append-delayed
        (find-assertions query-pattern frame)
        (delay (apply-rules query-pattern frame))))
```

Note that the application of rules is delayed. In our case, there are valid assertions answering the query, but the rule is invalid, since trying to apply it causes an infinite loop.

Without the delay we get:

```
*** - Program stack overflow. RESET
```

Without any useful output, because `simple-query` deepens more and more into the recursive rule without ever printing results.

Thanks to Flavio Cruz for suggesting this solution

Exercise 4.72

As the hint suggests, this is done for exactly the same reason as the original `interleave` in section 3.5.3 – for handling infinite streams. Suppose that `stream-flatmap` is called on two streams. The first is infinite, for some (maybe valid) reason. Without using `interleave`, the elements of the second stream won't be reached, ever.

Exercise 4.74

a.

`simple-flatten` assumes that it has a stream of streams, in which each stream is either the empty stream or a singleton stream. It has to do the following:

1. Filter out all the empty streams 2. Extract the elements from the singleton streams 3. Stitch all those elements together into a new stream

Therefore the implementation is as follows:

```
(defun simple-stream-flatmap (proc s)
  (simple-flatten (stream-map proc s)))

(defun simple-flatten (stream)
  (stream-map
    #'stream-car
    (stream-filter
      (lambda (s)
        (not (stream-null? s)))
      stream)))
```

b.

I don't think this change will affect the system's behavior. It does not affect correctness, because the named functions indeed only generate either singleton or empty streams. Also, as far as I can see, all the tests pass.

I also don't see how it can change the order of assertions that are output from queries `flatten-stream` that uses `interleave` creates the same elements as those created by `simple-flatten`, in the same order.

Exercise 4.75

```
(defun uniquely-asserted (query frame-stream)
  "Return only those frames for whom the query
  produces a single match in the database"
  (simple-stream-flatmap
    (lambda (frame)
      (let ((eval-stream
              (qeval
                (query-of-unique query)
                (singleton-stream frame))))
        (if (singleton-stream? eval-stream)
          eval-stream
          the-empty-stream)))
    frame-stream))

(defun singleton-stream? (stream)
  (and
    (stream-car stream)
    (not (stream-cadr stream))))

(defun query-of-unique (uniq) (car uniq))

(put-op 'unique 'qeval #'uniquely-asserted)
```

Some tests with the database:

```
(qinterpret
  '(unique (job ?x (computer wizard))))
=>
(UNIQUE (JOB (BITDIDDLE BEN) (COMPUTER WIZARD)))

(qinterpret
  '(and (job ?x ?j) (unique (job ?anyone ?j))))
=>
(AND (JOB (AULL DEWITT) (ADMINISTRATION SECRETARY)) (UNIQUE (JOB (AULL DEWITT) (ADMINISTRATION SECRE
(AND (JOB (CRATCHET ROBERT) (ACCOUNTING SCRIVENER)) (UNIQUE (JOB (CRATCHET ROBERT) (ACCOUNTING SCRIV
(AND (JOB (SCROOGE EBEN) (ACCOUNTING CHIEF ACCOUNTANT))
 (UNIQUE (JOB (SCROOGE EBEN) (ACCOUNTING CHIEF ACCOUNTANT))))
(AND (JOB (WARBUCKS OLIVER) (ADMINISTRATION BIG WHEEL))
 (UNIQUE (JOB (WARBUCKS OLIVER) (ADMINISTRATION BIG WHEEL))))
(AND (JOB (REASONER LOUIS) (COMPUTER PROGRAMMER TRAINEE))
 (UNIQUE (JOB (REASONER LOUIS) (COMPUTER PROGRAMMER TRAINEE))))
(AND (JOB (TWEAKIT LEM E) (COMPUTER TECHNICIAN)) (UNIQUE (JOB (TWEAKIT LEM E) (COMPUTER TECHNICIAN))
(AND (JOB (BITDIDDLE BEN) (COMPUTER WIZARD)) (UNIQUE (JOB (BITDIDDLE BEN) (COMPUTER WIZARD))))

(qinterpret
  '(and (supervisor ?peon ?boss)
        (unique (supervisor ?others ?boss))))
=>
(AND (SUPERVISOR (CRATCHET ROBERT) (SCROOGE EBEN)) (UNIQUE (SUPERVISOR (CRATCHET ROBERT) (SCROOGE EB
(AND (SUPERVISOR (REASONER LOUIS) (HACKER ALYSSA P))
 (UNIQUE (SUPERVISOR (REASONER LOUIS) (HACKER ALYSSA P))))
```

Exercise 4.76

Here's `merge-frames` that merges a pair of frames:

```
(defun merge-frames (f1 f2)
  (if (null f1)
    f2
    (let ((var (caar f1))
          (val (cdar f1)))
      (let ((extension (extend-if-possible var val f2)))
        (if (equal extension 'failed)
          'failed
          (merge-frames (cdr f1) extension))))))
```

And this is `merge-frame-streams` that merges two streams of frames, using`merge-frames` for each pair:

```
(defun merge-frame-streams (s1 s2)
  "Tries to merge each frame of s1 with each frame
  of s2. Returns the stream of successful merges."
  (stream-flatmap
    (lambda (f1)
      (stream-filter
        (lambda (f) (not (equal f 'failed)))
        (stream-map
          (lambda (f2)
            (merge-frames f1 f2))
          s2)))
    s1))
```

`conjoin` can then be written thus:

```
(defun conjoin (conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
    frame-stream
    (merge-frame-streams
      (qeval (first-conjunct conjuncts) frame-stream)
      (conjoin (rest-conjuncts conjuncts) frame-stream))))
```

This works satisfactorily, in most cases. For example:

```
(qinterpret
  '(and
    (job ?x ?j)
    (supervisor ?x (bitdiddle ben))
    (salary ?x 25000)))
=>
(AND (JOB (TWEAKIT LEM E) (COMPUTER TECHNICIAN)) (SUPERVISOR (TWEAKIT LEM E) (BITDIDDLE BEN))
 (SALARY (TWEAKIT LEM E) 25000))

; the 'grandson' rule uses /and/
(qinterpret
  '(grandson ?d ?w))
=>
(GRANDSON MEHUJAEL LAMECH)
(GRANDSON IRAD METHUSHAEL)
(GRANDSON ENOCH MEHUJAEL)
(GRANDSON CAIN IRAD)
(GRANDSON ADAM ENOCH)
```

However, there are also problems. For example, consider this:

```
(qinterpret
  '(and
    (job ?x ?j)
    (not (supervisor ?x (bitdiddle ben)))))
=>
```

It should have printed out many items, but it doesn't. This is because of the problem with `not` described in section 4.4.3; `not` needs something to filter, and the new implementation of `conjoin` simply runs it on the full input frame stream, and merges the result.

A more serious error with this approach happens when resursive rules are used. When this rule is added to the Biblical hierarchy database:

```
(qinterpret
  '(assert!
    (rule (son ?m ?s)
          (and
            (wife ?m ?w)
            (son ?w ?s)))))
```

The evaluator goes into an infinite loop when matching `son` rules. This is because the `son` rule invokes another `son` rule in the `and` expression, and with the new implementation of `conjoin` this call is not on a reduced set of frames.

I suppose the authors didn't make a mistake with their suggestion, but it's rather a problem with my implementation. I'll be glad if someone points me to it.

[1] In the authors' implementation, an assertion is indexable if its first element is either a constant or a variable name, but not a list.

[2] Recall that a singleton stream is simply an element wrapped up as a stream.

For comments, please send me an email.

Back to top