Logic-Oriented Programming (5/11/2004)

Daniel P. Friedman, David W. Mack, William E. Byrd
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography Center,
Monterey, CA 93943, USA

This is a presentation of *Logic-Oriented Programming*. We have chosen the name *logic* because the three ideas that we use in this development were the "killer apps" that got Logic Programming its early recognition: executing programs backwards, using difference lists, and solving elementary logic puzzles. We assume the reader is familiar with *Outcome-Oriented-Programming* and *Relation-Oriented-Programming*.

We examine our first concept by working with `len`, which in Scheme is called `length`. Let's first look at the conventional function.

```
(define len
  (lambda (ls)
    (cond
      [(null? ls) 0]
      [else (+ (len (cdr ls)) 1)])))

> (len '(a b c u v))
5
> (answers (q) (== q (len '(a b c u v))))
q :: 5

#f
```

But, we can write the relation `len`, which naturally takes one additional argument, since we say that `len` of the list `ls` is some number `n`.

```
(define len
  (extend-relation (a b)
    (fact () '() 0)
    (relation (h t n)
      (to-show `(,h . ,t) n)
      (exists (nt)
        (len t nt)
        (project (nt)
          (== n (+ 1 nt)))))))
```

```
> (answers (n) (len '(a b c u v) n))
n :: 5

#f

> (answers (ls) (len ls 5))
ls :: (h.0 h.1 h.2 h.3 h.4)
```

The fact that we only get one answer for each invocation of `answers` should not matter. In fact, the second one loops indefinitely after it finds the first answer. The point is that we asked for a list of length five and we got one. This kind of reasoning is very foreign to the way we normally think about programming. We call feeding the output and asking for the input *executing backwards*.

We next re-examine executing backwards using Scheme's `append`. Since we shall be defining `append` globally, we change its name to `concat` to avoid a conflict with Scheme's `append`. We start by first defining `concat` as a Scheme procedure.

```
(define concat
  (lambda (x* y*)
    (cond
      [(null? x*) y*]
      [else (cons (car x*) (concat (cdr x*) y*))])))
```

```
> (concat '(a b c) '(u v))
(a b c u v)
```

or the equivalent

```
> (answers (q) (== q (concat '(a b c) '(u v))))
q :: (a b c u v)

#f
```

And we can write the corresponding relation over *three* lists,

```
(define concat
  (extend-relation (a1 a2 a3)
    (fact (x*) '() x* x*)
    (relation (x x* y* z*)
      (to-show `(,x . ,x*) y* `(,x . ,z*))
      (concat x* y* z*))))
```

```
> (answers (q) (concat '(a b c) '(u v) q))
q :: (a b c u v)

#f
```

which determines that there is only one answer and which shows if we concatenate `(a b c)` and `(u v)`, we get `(a b c u v)`. But, we can move `q` to another position!

```
> (answers (q) (concat '(a b c) q '(a b c u v)))
q :: (u v)

#f
```

This time we determine that `q` should be `(u v)`, which is *not* possible with `concat` as a function. Similarly, we can determine that `q` is `(a b c)`.

```
> (answers (q) (concat q '(u v) '(a b c u v)))
q :: (a b c)

#f
```

But the fun has only begun. What if we include another variable? Then, we can get more, but still a finite number of answers.

```
> (answers (q r) (concat q r '(a b c u v)))
q :: ()
r :: (a b c u v)

q :: (a)
r :: (b c u v)

q :: (a b)
r :: (c u v)

q :: (a b c)
r :: (u v)

q :: (a b c u)
r :: (v)

q :: (a b c u v)
r :: ()

#f
```

We get all the ways that we might concatenate two lists to form `(a b c u v)`.

Now, what if we include yet another variable?

```
> (answers (q r s) (concat q r s))
q :: ()
r :: x*.0
s :: x*.0

q :: (x.0)
r :: x*.0
s :: (x.0 . x*.0)
```

```
q :: (x.0 x.1)
r :: x*.0
s :: (x.0 x.1 . x*.0)

q :: (x.0 x.1 x.2)
r :: x*.0
s :: (x.0 x.1 x.2 . x*.0)
```

and a lot more

Here we see that the empty list and any list yield that list. Then we get all sorts of constructed lists with the first N elements of the list chosen as variables of that length. There is no bound on the number of answers.

Now, that we see just how easy it is to write programs that yield an unbounded number of answers, we revise `answers` to `answer*`, which allows us to put a bound on the number of answers we want to see.

```
(def-syntax (answer* max-num (id ...) ant)
  (let ([n max-num])
    (if (> n -1)
      (let loop ([ans (run (id ...)
                        (all ant (trace-vars "::" (id ...)))
                        fk subst fk #f)]
                 [n n])
        (if (not (or (zero? n) (not ans)))
          (loop (ans) (sub1 n))
          (if (not ans) #f #t))))))
```

We can even get an unbounded number of answers with only two variables.

```
> (answer* 4 (q r) (concat q '(u v) `(a b c . ,r)))
q :: (a b c)
r :: (u v)

q :: (a b c x.0)
r :: (x.0 u v)

q :: (a b c x.0 x.1)
r :: (x.0 x.1 u v)

q :: (a b c x.0 x.1 x.2)
r :: (x.0 x.1 x.2 u v)

#t
```

The first answer is the one we expect, where `q` binds `(a b c)` and `r` binds `(u v)`. But, then we discover that `q` and respectively `r` could be a bit longer.

An alternative uses `run` directly, but relies on a function that strips off the first n values from a stream.

```
(define stream-prefix
  (lambda (n strm)
    (if (null? strm) '()
      (cons (car strm)
        (if (zero? n) '()
          (stream-prefix (- n 1) ((cdr strm))))))))

> (stream-prefix 3
    (run (q r) (concat q '(u v) `(a b c . ,r))
      fk subst
      (cons `(,q ,r) fk) '()))
(((a b c) (u v))
 ((a b c x.0) (x.0 u v))
 ((a b c x.0 x.1) (x.0 x.1 u v))
 ((a b c x.0 x.1 x.2) (x.0 x.1 x.2 u v)))
```

The only subtle aspect is that the second argument to `cons` is `fk` instead of `(fk)`. This places the burden on `stream-prefix` to invoke the failure continuation, which is does with `((cdr strm))`. If we had used `(fk)` instead of `fk`, then we would never enter `stream-prefix`, since there would be an unbounded number of answers.

And here is an unbounded number of answers with a single variable. Again, the first answer seems logical enough, since concatenating the empty list to the empty list yields the empty list. But, also concatenating the empty list to any list yields that list.

```
> (answer* 4 (q) (concat q '() q))
q :: ()

q :: (x.0)

q :: (x.0 x.1)

q :: (x.0 x.1 x.2)

#t
```

A program like `concat` is what excited the logic programming world. It was called `append` because of the use of that name in Lisp.

Our second concept is difference lists[1], which improve efficiency by allowing us to have data structures that are not fully specified.

[1] Examples in this section taken from William F. Clocksin. 1997. *Clause and Effect: Prolog Programming for the Working Programmer*. Berlin, Heidelberg, and New York: Springer-Verlag.

A difference list is a segment of a list denoted by pointers to the *front* and *back* of the segment. For instance, consider the difference list `(a b c)` represented by the variables Z_f and Z_b. The variable Z_f points to the front of the list and Z_b points to the back. Thus, the whole difference list would be represented as `(Diff` Z_f Z_b`)` where Z_f is `(a b c .` Z_b`)`. In other words, Z_f represents the beginning of this segment of the difference list and Z_b handles everything else.

Observe another difference list `(Diff` Y_f Y_b`)` where Y_f is `(u v .` Y_b`)`. We can concatenate the two difference lists in constant time by tying the back of Z to the front of Y. Thus, the concatenate operation of the two difference lists is `(Diff` Z_f Y_b`)` where Z_f is `(a b c .` Y_f`)`.

Next, we write a new `concat` relation for difference lists that demonstrates how efficient this operation is. This definition is exactly what we expect from the discussion in the previous paragraph.

```
(define concat
  (relation (Zf Zb Yf Yb)
    (to-show `(Diff ,Zf ,Zb) `(Diff ,Yf ,Yb) `(Diff ,Zf ,Yb))
    (== Zb Yf)))
```

By removing the unification and changing some variable names, we can make the definition even simpler.

```
(define concat
  (fact (Zf Zb Yb) `(Diff ,Zf ,Zb) `(Diff ,Zb ,Yb) `(Diff ,Zf ,Yb)))
```

```
> (answer* 1 (Zb Yb res)
    (concat `(Diff (a b c . ,Zb) ,Zb) `(Diff (u v . ,Yb) ,Yb) res))
zb :: (u v . yb.0)
yb :: yb.0
res :: (diff (a b c u v . yb.0) yb.0)

#f
```

Thus, we perform the same concatenate operation mentioned before and the system yields the expected result.

We define `answer`, which abstracts the pattern `(answer* 1 ...)`.

```
(def-syntax (answer (id ...) ant)
  (run-once (id ...) (all ant (trace-vars "::" (id ...))) (void)))
```

Now let's try some other variations:

```
> (answer (Zb Yb q)
    (concat `(Diff (a b c . ,Zb) ,Zb) q `(Diff (a b c u v . ,Yb) ,Yb)))
zb :: (u v . yb.0)
yb :: yb.0
q :: (diff (u v . yb.0) yb.0)
```

```
> (answer (q r Yb) (concat q r `(Diff (a b c u v . ,Yb) ,Yb)))
q :: (diff (a b c u v . yb.0) zb.0)
r :: (diff zb.0 yb.0)
yb :: yb.0
```

As with the definition of `concat` at the beginning, our new difference list version can compute in many directions. The difference here is that the operations are all occurring in constant time!

It is possible to *rectify* a difference list. This process converts the difference list into a proper Scheme list. We rectify the difference list Z from above and then show that we can use `rectify` to perform the reverse operation as well.

```
> (define rectify (fact (Yf) `(Diff ,Yf ()) Yf))
> (answer (ls) (exists (Zb) (rectify `(Diff (a b c . ,Zb) ,Zb) ls)))
ls :: (a b c)

> (answer (Z) (rectify Z `(a b c)))
z :: (diff (a b c) ())
```

Now that we know how to convert difference lists to proper lists and back, we can use them to perform computation on proper lists. Consider the process of *linearizing* or flattening a list of elements. Assume we have a list `(a b (c (d)) e)`. Its linearized result would be `(a b c d e)`. Consider an implementation of `flatten` that inputs and outputs normal Scheme lists, but uses difference lists internally for computation. This implementation is considerably more efficent than the standard Scheme recursive function. Examine the `flatpair` relation. Its first clause handles the empty list, so an empty difference list is returned. The second case flattens the head and tail of the list. The third clause handles elements.

```
(define flatten
  (relation (x y)
    (to-show x y)
    (flatpair x `(Diff ,y ()))))

(define flatpair
  (extend-relation (a1 a2)
    (fact (X) '() `(Diff ,X ,X))
    (relation (h t Xf Xb)
      (to-show `(,h . ,t) `(Diff ,Xf ,Xb))
      (exists (Y)
        (flatpair h `(Diff ,Xf ,Y))
        (flatpair t `(Diff ,Y ,Xb))))
    (fact (x Yb) x `(Diff (,x . ,Yb) ,Yb))))

> (answer (y) (flatten '(a b (c (d)) e) y))
y :: (a b c d e)
```

We can also write logic programs that linearize binary tree structures. For this example, we take a binary tree like `((5 6) (d (0 e)))` and build a list of integers found in leaves of the tree: `(5 6 0)`. The `lintree` relation performs this operation, taking the tree as `x` and the linearized list as `y`. The first clause of `lindiff` handles nodes, the second takes care of leaves that are integers (observe the use of `project` and `predicate`), and the last case throws away non-integers.

```
(define lintree
  (relation (x y)
    (to-show x y)
    (lindiff x `(Diff ,y ()))))

(define lindiff
  (extend-relation (a1 a2)
    (relation (a b l1 l3)
      (to-show `(,a ,b) `(Diff ,l1 ,l3))
      (exists (l2)
        (lindiff a `(Diff ,l1 ,l2))
        (lindiff b `(Diff ,l2 ,l3))))
    (relation (x l)
      (to-show x `(Diff (,x . ,l) ,l))
      (project (x)
        (predicate (integer? x))))
    (fact (x l) x `(Diff ,l ,l))))
```

```
> (answer (y) (lintree '((5 6) (d (5 e))) y))
y :: (5 6 5)
```

We come to our last concept, solving logic puzzles. One of the most well-known logic puzzles is "The Zebra Problem." You can tell from the statement of the original problem below that it is old. There are several more modern versions of this that are not nearly so biased, but there is something to learn from seeing the original. For example, at the time of the problem's statement, people were smoking cigarettes, some of whose brands are no longer available. Here is the specification of the problem:

1. There are five houses in a row, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
2. The Englishman lives in the red house.
3. The Spaniard owns a dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is directly to the right of the ivory house.
7. The Old Golds smoker owns snails.
8. Kools are being smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.

11. The Chesterfields smoker lives next to the fox owner.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strikes smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.
16. Who owns the zebra and who drinks the water?

The solution is nearly as obvious as the statement of the problem. The trick is knowing how to translate the problem statement into a relation.

We start with the driver `zebra` below, and then explain the help relations. Here, we use a global variable, generally known as the anonymous variable or the wildcard. It matches against any term, but has the nice property that the substitution remains unchanged.

The first antecedent in the `all!` captures information from lines 1, 9, and 10. The value of `houses` becomes a list of five houses, each of which is a five-element list. By placing `norwegian` where we have, it should be clear that our representation in each five-element list assumes that the nationality is in the first position. Furthermore, by placing `milk` where we have, we are saying that each drink is in the third position. Finally, these five-element lists are ordered, so we place `milk` in the third position of the *third* house. The anonymous variable is used rather subtly, too. When it appears by itself, it is representing a five-element list, but when it appears in a five-element list, it represents either some nationality, some cigarette, some drink, some pet, or some color. What then becomes critical is that each help relation, and there are only three of them, takes `houses` as its last argument.

```
(define zebra
  (relation (houses)
    (to-show houses)
    (ef/only
      (all!
        (== houses `((norwegian ,_ ,_ ,_ ,_) ,_ (,_ ,_ milk ,_ ,_) ,_ ,_))
        (memb `(englishman ,_ ,_ ,_ red) houses)
        (on-right `(,_ ,_ ,_ ,_ ivory) `(,_ ,_ ,_ ,_ green) houses)
        (next-to `(norwegian ,_ ,_ ,_ ,_) `(,_ ,_ ,_ ,_ blue) houses)
        (memb `(,_ kools ,_ ,_ yellow) houses)
        (memb `(spaniard ,_ ,_ dog ,_) houses)
        (memb `(,_ ,_ coffee ,_ green) houses)
        (memb `(ukrainian ,_ tea ,_ ,_) houses)
        (memb `(,_ luckystrikes oj ,_ ,_) houses)
        (memb `(japanese parliaments ,_ ,_ ,_) houses)
        (memb `(,_ oldgolds ,_ snails ,_) houses)
        (next-to `(,_ ,_ ,_ horse ,_) `(,_ kools ,_ ,_ ,_) houses)
        (next-to `(,_ ,_ ,_ fox ,_) `(,_ chesterfields ,_ ,_ ,_) houses))
      (all (memb `(,_ ,_ water ,_ ,_) houses) (memb `(,_ ,_ ,_ zebra ,_) houses))
      (fail))))
```

Understanding three help relations is all that is left. Let's start with the simplest, `next-to`.

```
(define next-to
 (relation (house-a house-b houses)
   (to-show house-a house-b houses)
   (any (on-right house-a house-b houses)
        (on-right house-b house-a houses))))
```

Thus, given two houses and all the houses, either the first house is on the right of the second house or the second house is on the right of the first house. We don't have to know anything about `on-right`. We just have to assume that `on-right` works correctly.

We next define `on-right`.

```
(define on-right
  (extend-relation (a1 a2 a3)
    (fact (left-h right-h) left-h right-h `(,left-h ,right-h . ,_))
    (relation (left-h right-h cdr-houses)
      (to-show left-h right-h `(,_ . ,cdr-houses))
      (on-right left-h right-h cdr-houses))))
```

Here we have two relations. The first one is the base relation. It says that of the remaining houses, `right-h` is on the right of `left-h`. We don't know how many houses are left in the list and we don't care. That is why we write "`. ,_`" It says that we are ignoring the rest of the list. Now, if the first relation fails, we try the second relation. So, we just do a recursion on the `cdr`, leaving `left-h` and `right-h` unchanged. To remind ourselves that we really are doing all this in Scheme, here is a simpler version of `on-right` that acknowledges that `left-h` and `right-h` never change. (By the way, it uses fewer bytes this way.)

```
(define on-right
  (lambda (left-h right-h houses)
    (letrec
      ([on-right (extend-relation (a1)
                   (fact () `(,left-h ,right-h . ,_))
                   (relation (cdr-houses)
                     (to-show `(,_ . ,cdr-houses))
                     (on-right cdr-houses)))])
      (on-right houses))))
```

We have only `memb`, which is like `member`, left to write.

```
(define memb
  (relation (item ls)
    (to-show item ls)
    (any
      (== ls `(,item . ,_))
      (exists (rest)
        (ef/only (== ls `(,_ . ,rest)) (memb item rest) (fail))))))
```

Here we show recursion using a single relation within an `any`. The base relation succeeds if `ls` is some list whose `car` matches the item we are looking for. Otherwise, we introduce a variable to bind the `cdr`, and then recur on the `odr`.

Finally, we would like to know something about the resources consumed when we run `zebra`. Specifically, we want to see how much time it takes and how much memory it uses.

```
> (time (answer (h) (zebra h)))
h :: ((norwegian kools water fox yellow)
      (ukrainian chesterfields tea horse blue)
      (englishman oldgolds milk snails red)
      (spaniard luckystrikes oj dog ivory)
      (japanese parliaments coffee zebra green))

(time (answer (h) (zebra h)))
    no collections
    0 ms elapsed cpu time
    0 ms elapsed real time
    67640 bytes allocated
```

The computation is reasonably fast, but in full Kanren, it is much faster. Also, it allocated 67640 bytes.