Login

Assignment 10: Introduction to Logic Programming

> Success is the ability to go from one failure to another with no loss of enthusiasm.

Preliminaries

This week we've begun our discussion of relational programming using miniKanren. Before you start the assignment, make sure you have mk.rkt and numbers.rkt.

```
(require "mk.rkt")
(require "numbers.rkt")
```

To further familiarize yourself with miniKanren, you should consult The Reasoned Schemer and your own notes from class.

Assignment

This assignment is composed of two parts, plus a brainteaser. To test your assignment, you can used the attached a10-student-tests.rkt

Notes

You may find the following notes on transforming Racket programs to miniKanren to be useful. You might also want to consult the documentation on match-let* [http://docs.racket-lang.org/reference/match.html?q=match-let#%28form._%28%28lib._racket%2Fmatch..rkt%29._match-let%2A%29%29].

You might also be interested in the following [http://www.cs.indiana.edu/l/www/classes/c311/miniaop.pdf] sequence [http://www.cs.indiana.edu/l/www/classes/c311/minirop.pdf] of notes [http://www.cs.indiana.edu/l/www/classes/c311/minilop.pdf] on logic programming, or these notes [http://www.cs.indiana.edu/l/www/classes/c311/miniunify.pdf] on unification.

Part I

Write the answers to the questions found in the a10.rkt file provided to you. **For each problem, explain how miniKanren arrived at the answer.** You will be graded on the quality and completeness of your explanation; a full explanation will require several sentences.

Part II

Here are the Racket procedures `assoc`, and `reverse`, `stutter`:

```
(define assoc
  (lambda (x ls)
    (match-let* ((`(,a . ,d) ls)
                 (`(,aa . ,da) a))
      (cond
        ((equal? aa x) a)
        ((not (equal? aa x)) (assoc x d))))))

(define reverse
  (lambda (ls)
    (cond
      ((equal? '() ls) '())
      (else
       (match-let* ((`(,a . ,d) ls)
                    (res (reverse d)))
         (append res `(,a)))))))

(define stutter
  (lambda (ls)
    (cond
      ((equal? '() ls) '())
      (else
        (match-let* ((`(,a . ,d) ls)
                     (res (stutter d)))
          `(,a ,a . ,res))))))
```

Take `assoc`, `reverse`, and `stutter`, and translate them into the equivalent miniKanren relations (`assoco`, and `reverseo`, `stuttero`) and put them at the end of your `a10.rkt` file. Your `numbers.rkt` file already contains an implementation of `appendo`.

The below tests are a guide. Your relations might not pass these tests, as implementing goals in a different order may cause the stream to return results in a different order. So it is possible that your code is correct though the tests fail.

Remember that you may need to rearrange some of the goals in your relations to ensure termination (and therefore to pass the tests). In general, recursive calls should come at the end of a sequence of goals, while explicit or implicit calls to == should come at the beginning.

Here are the tests.

```
> (require "a10.rkt")

> (run 1 q (stuttero q '(1 1 2 2 3 3)))
((1 2 3))

> (run* q (stuttero q '(1 1 2 2 3 3)))
((1 2 3))

> (run 1 q (fresh (a b c d) (== q `(,a ,b ,c ,d)) (stuttero a `(1 ,b ,c 2 3 ,d))))
(((1 2 3) 1 2 3))

> (run 1 q (fresh (a b c d) (== q `(,a ,b ,c ,d)) (stuttero `(,b 1) `(,c . ,d))))
((_0 _1 _1 (_1 1 1)))
```

```
> (run 1 q (fresh (e f g) (== q `(,e ,f ,g)) (stuttero `(,e . ,f) g)))
((_0 () (_0 _0)))

> (run 2 q (fresh (e f g) (== q `(,e ,f ,g)) (stuttero `(,e . ,f) g)))
((_0 () (_0 _0)) (_0 (_1) (_0 _0 _1 _1)))

> (run* q (assoco 'x '() q))
()

> (run* q (assoco 'x '((x . 5)) q))
((x . 5))

> (run* q (assoco 'x '((y . 6) (x . 5)) q))
((x . 5))

> (run* q (assoco 'x '((x . 6) (x . 5)) q))
((x . 6))

> (run* q (assoco 'x '((x . 5)) '(x . 5)))
(_0)

> (run* q (assoco 'x '((x . 6) (x . 5)) '(x . 6)))
(_0)

> (run* q (assoco 'x '((x . 6) (x . 5)) '(x . 5)))
()

> (run* q (assoco q '((x . 6) (x . 5)) '(x . 5)))
()

> (run* q (assoco 'x '((x . 6) . ,q) '(x . 6)))
(_0)

> (run 5 q (assoco 'x q '(x . 5)))
(((x . 5) . _0)
  ((_0 . _1) (x . 5) . _2)
  ((_0 . _1) (_2 . _3) (x . 5) . _4)
  ((_0 . _1) (_2 . _3) (_4 . _5) (x . 5) . _6)
  ((_0 . _1) (_2 . _3) (_4 . _5) (_6 . _7) (x . 5) . _8))

> (run 5 q (fresh (x y z)
               (assoco x y z)
               (== `(,x ,y ,z) q)))
((_0 ((_0 . _1) . _2) (_0 . _1))
  (_0 ((_1 . _2) (_0 . _3) . _4) (_0 . _3))
  (_0 ((_1 . _2) (_3 . _4) (_0 . _5) . _6) (_0 . _5))
  (_0 ((_1 . _2) (_3 . _4) (_5 . _6) (_0 . _7) . _8) (_0 . _7))
  (_0 ((_1 . _2) (_3 . _4) (_5 . _6) (_7 . _8) (_0 . _9) . _10) (_0 . _9)))

> (run* q (reverseo '() q))
(())

> (run* q (reverseo '(a) q))
((a))
```

```
> (run* q (reverseo '(a b c d) q))
((d c b a))

> (run* q (fresh (x) (reverseo `(a b ,x c d) q)))
((d c _0 b a))

> (run* x (reverseo `(a b ,x d) '(d c b a)))
(c)

> (run* x (reverseo `(a b c d) `(d . ,x)))
((c b a))

> (run* q (fresh (x) (reverseo `(a b c d) `(d . (,q . ,x)))))
(c)

> (run 10 q (fresh (x y) (reverseo x y) (== `(,x ,y) q)))
((() ())
  ((_0) (_0))
  ((_0 _1) (_1 _0))
  ((_0 _1 _2) (_2 _1 _0))
  ((_0 _1 _2 _3) (_3 _2 _1 _0))
  ((_0 _1 _2 _3 _4) (_4 _3 _2 _1 _0))
  ((_0 _1 _2 _3 _4 _5) (_5 _4 _3 _2 _1 _0))
  ((_0 _1 _2 _3 _4 _5 _6) (_6 _5 _4 _3 _2 _1 _0))
  ((_0 _1 _2 _3 _4 _5 _6 _7) (_7 _6 _5 _4 _3 _2 _1 _0))
  ((_0 _1 _2 _3 _4 _5 _6 _7 _8) (_8 _7 _6 _5 _4 _3 _2 _1 _0)))
```

Brainteaser

Chapter 7 of *The Reasoned Schemer* introduces a way to handle integers in relational programming. These are provided in the `numbers` suite.

Write `lengtho`, which associates its output with the length of a list as a binary number in reverse order (if you use the provided helpers, this is the normal representation of numbers). Here are some tests:

```
> (require "a10.rkt")

> (run 1 q (lengtho '() q))
((()))

> (run 1 q (lengtho '(a b) q))
((0 1))

> (run 1 q (lengtho '(a b c) q))
((1 1))

> (run 1 q (lengtho '(a b c d e f g) q))
((1 1 1))

> (run 1 q (lengtho q (build-num 0)))
((()))
```

```
> (run 1 q (lengtho q (build-num 5)))
((_0 _1 _2 _3 _4))

> (run 10 q (fresh (x y) (lengtho x y) (== `(,x ,y) q)))
((() ())
  ((_0) (1))
  ((_0 _1) (0 1))
  ((_0 _1 _2) (1 1))
  ((_0 _1 _2 _3) (0 0 1))
  ((_0 _1 _2 _3 _4) (1 0 1))
  ((_0 _1 _2 _3 _4 _5) (0 1 1))
  ((_0 _1 _2 _3 _4 _5 _6) (1 1 1))
  ((_0 _1 _2 _3 _4 _5 _6 _7) (0 0 0 1))
  ((_0 _1 _2 _3 _4 _5 _6 _7 _8) (1 0 0 1)))
```

Just Dessert

We saw how to implement a microKanren with ==, and we saw in lab how to build an miniKanren on top of that. But we haven't added the other constraints. Given the file micro-for-diseq.rkt add =/= to the microKanren implementation. You can make use of the file of helper macros mini.rkt if you would like a higher-level language from which to work.

lp-a2.txt · Last modified: 2018/04/08 09:21 by mvc