CSC324 Principles of Programming Languages

Lecture 9

November 16/17, 2020

Logic Programming

Our implementation of logic programming with `-<` and `?-` is not the only way

- Prolog is the most well-known logic programming language
- **miniKanren** is a relational programming language that we'll discuss

Let Lisa know if you're interested in a project course on miniKanren and are doing well in this course!

A student from last year wrote this paper: A Relational Interpreter for Synthesizing JavaScript

Relational Programming with miniKanren

Relations

In math, a *function* maps an input to a unique output.

A *relation* describes a set of values that satisfy the relation.

- Arithmetic relations ==, >, >=
- Mathematical relations $(x^2 + y^2 = 3)$

Functions and Relations in Math

We can define the relational form of functions

Function:

- `f(x) = x + 1`

Relation:

- `x + 1 = y` or `x + 1 - y = 0`

Functions and Relations

We can define the relational form of functions in the same way!

Function:

- `f x = x + 1`

Relation:

- `fo x y` is satisfied when `y = x + 1`

Example: The `appendo` relation

Function:

- `(append xs ys)` produces the output of the two lists concatenated together

Relation:

- `(appendo xs ys xsys)` is satisfied when `xsys` is the result of appending the two lists together

Querying relations

We can use miniKanren to find unknowns to satisfy a call to a relation, called a **goal**.

```
> (require "mk.rkt") ; from ex9 starter code
> (run 1   ; return at most 1 answer
       (x) ; the unknown that we are searching for
       (appendo '(1 2) x '(1 2 3 4 5))) ; the goal we wish
'((3 4 5))
```

The anatomy of a query

The unknown variables `x y` are called **logic variables**

```
> (run 1      ; return at most 1 answer
       (x y) ; the unknowns that we are searching for
       (appendo x y '(1 2 3 4 5))) ; the goal we wish to so
'((() (1 2 3 4 5)))
```

Find all answers

```
> (run* (x y) (appendo x y '(1 2 3 4 5)))
'((() (1 2 3 4 5))
  ((1) (2 3 4 5))
  ((1 2) (3 4 5))
  ((1 2 3) (4 5))
  ((1 2 3 4) (5))
  ((1 2 3 4 5) ()))
```

Search

Under the hood, there is an implicit search through all possible **terms** that the logic variable can take.

In the miniKanren world, a term can be either a:

- symbol, number, string, boolean, empty list (an atom)
- a pair (which can be used to construct a list)
- a logic variable

Example queries with logic variables

```
> (run* (x y) (appendo (cons x y) y '(1 1 1 1 1)))
'((1 (1 1)))
```

More complex queries: conjunctions

If we write multiple goals in a relation, miniKanren assumes that these calls to relations must *all* be satisfied:

```
> (run* (x y) (appendo x y '(1 2 3)) (appendo x x '(1 1)))
'(((1) (2 3)))
```

More complex queries: disjunction using conde

```
> (run* (x) (conde ((== x 1)) ((== x 2))))
'(1 2)
```

Condes are disjunctions of conjunctions

```
> (run* (x) (conde ((== x 1) (=/= x 1)) ((== x 2))))
'(2)
```

Repeated results

The same result can appear multiple times

```
> (run* (x) (conde ((== x 1)) ((== x 1))))
'(1 1)
```

Logic variables in results

Logic variables can appear in results

```
> (run* (x y) (== x y))
'((_.0 _.0))
```

Here, we see that `x` and `y` refers to the same logic variable. (Their actual values can be any term)

Type constraints in results

```
> (run 1 (x) (symbolo x))
'((_.0 (sym _.0)))
```

Type constraints like `symbolo`, `numbero` will add an extra constraint on logic variables.

Why does this fail?

```
> (run* (x y) (== x (car y)))
; car: contract violation
;   expected: pair?
;   given: '#((unbound) (scope) 12)
; [,bt for context]
```

Since `y` can be a logic variable, `car` won't always work. (In general miniKanren won't work with Racket functions)

We cannot *destruct* values, but we can *construct* values.

Using fresh logic variables

```
> (run* (x y) (fresh (rest) (== (cons x rest) y)))
'((_.0 (_.0 . _.1)))
```

Writing our first relation

Let's write `appendo`. Here's a version of the function `append`

```
(define (append xs ys)
  (cond [(equal? xs '())
         ys]
        [else
         (let* ([x (car xs)]
                [xs^ (cdr xs)]
                [xsys^ (append xs^ ys)])
           (cons x xsys^))]))
```

Writing appendo (part 1)

Instead of `cond`, we use a disjunction `conde`.

First handle the base case where `xs` is empty.

```
(define (appendo xs ys xsys)
  (conde ((== xs '())
          (== ys xsys))
         ...))
```

Need a side-by-side comparison

Writing appendo (part 2)

```
(define (appendo xs ys xsys)
  (conde ((== xs '())
          (== ys xsys))
         ((fresh (x xs^ xsys^)
            (== xs (cons x xs^))
            (== xsys (cons x xsys^))
            (appendo xs^ ys xsys^)))))
```

Writing Relations

Exercise: Write the relation membero

Breakout Group: 5 minutes

```
(define (member elem lst)
  (cond [(empty? lst) #f]
        [(equal? elem (first lst)) #t]
        [else (member elem (rest lst))]))
```

Write the relation membero (solution)

```
(define (membero elem lst)
  (fresh (first rest)
    (== lst (cons first rest))
    (conde
      ((== first elem))
      ((membero elem rest)))))
```

Association list

An **association list** is a list of key-value pairs:

```
> (define assoc '((a . 2) (b . 5)))
> (lookup assoc 'a)
2
```

Write the function `lookup`

(Breakout group: 5 minutes)

Write the function `lookup`

(Breakout group: 5 minutes)

```
(define (lookup lst key)
  (cond [(empty? lst) 'ERROR]
        [(equal? key (car (first lst)))
         (cdr (first lst))]
        [(lookup lst (rest lst))]))
```

Write the relation lookupo (attempt)

```
(define (lookupo lst key value)
```

Write the relation lookupo (attempt)

```
(define (lookupo lst key value)

(define (lookup lst key)
  (fresh (fkey fval rest)
    (== (cons (cons fkey fval) rest) lst)
    (conde ((== key fkey)
            (== value fval))
           ((lookupo rest key value)))))
```

This is **almost** correct, but there is an issue.

Duplicate results: lookupo

What happens if there are duplicate keys?

```
> (define assoc '((a . 2) (a . 5)))
> (lookup assoc 'a)
2
> (run* (v) (lookupo assoc 'a v))
'(2 5)
```

This feature is actually useful when dealing with variable shadowing.

Write the relation lookupo

```
(define (lookupo lst key value)
  (fresh (fkey fval rest)
    (== (cons (cons fkey fval) rest) lst)
    (conde ((== key fkey)
            (== value fval))
           ((=/= key fkey)
            (lookupo rest key value)))))
```

Side Note: Quasi-quote

Quasi-quote allows us to *unquote* part of the data structure

```
> (cons 'a (+ 1 2))
'(a 3)
> '(a (+ 1 2)) ; the entire data structure is quoted
'(a (+ 1 2))
> `(a ,(+ 1 2)) ; example of a quasi-quote
'(a 3)
```

A relational interpreter

An interpreter `eval` is a function too!

```
(define (eval expr env)
  ; ... an interpreter for some language
)
```

Can we write a relational interpreter `evalo`? Yes!

```
(define (evalo expr env value)
  ; ... a *relational* interpreter
)
```

Fun with `evalo`

Not so fun: evaluation

We can run the `evalo` relation *forward*, like the `eval` function

```
> (run 1 (value) (evalo '((lambda (x) x) '1) '() value))
1
```

Fun: running evalo backwards

```
> (run 5 (expr) (evalo expr '() 1))
'('1
  (((lambda (_.0) '1) '_.1) (sym _.0) (absento (closure _.1
  (((lambda (_.0) _.0) '1) (sym _.0))
  (((lambda (_.0) '1) (list)) (sym _.0))
  (((lambda (_.0) '1) (lambda (_.1) _.2)) (sym _.0 _.1)))
```

More fun: quines!

A **quine** is a program that evaluates to itself

```
> (run 1 (expr) (evalo expr '() expr))
'((((lambda (_.0) (list _.0 (list 'quote _.0)))
    '(lambda (_.0) (list _.0 (list 'quote _.0))))
   (=/= ((_.0 closure)))
   (sym _.0)))
```

Readable Quine in Racket

```
> ((lambda (x) (list x (list 'quote x))) '(lambda (x) (list
'((lambda (x) (list x (list 'quote x))) '(lambda (x) (list
```

Solving Problems

How would we solve sudoku from ex8?

- Write a function that **checks** if a solution is correct
- Write the relational form of that function
- Run it backwards using miniKanren

We won't do this, but you can see an example here:

https://github.com/gregr/experiments/blob/master/mk-misc/sudoku.scm

How would we solve the coins problem from ex8?

Generalized problem: given the *total* value of the change that we can make, and a list of *denominations* (possible coin values), and produce the possible ways that we can make the change with the coin denominations.

How would we solve the coins problem from ex8?

Generalized problem: given the *total* value of the change that we can make, and a list of *denominations* (possible coin values), and produce the possible ways that we can make the change with the coin denominations.

- Write a function that **checks** if a solution is correct
- Write the relational form of that function
- Run it backwards using miniKanren

1. Check if a solution is correct

```
(define (change? coins total denoms)
  (cond [(empty? coins) (equal? total 0)]
        [(member (first coins) denoms)
         (change? (rest coins) (- total (first coins)) deno
        [else #f]))

> (change? '(5 1 1 1 1) 9 '(5 2 1))
#t
```

2. Write the relational form of the function

One big issue. . . **miniKanren can't actually reason about Racket integers.**

However, we can use the relational arithmetic package `number.rkt`.

Version 1 of changeo

```
(require "number.rkt")

(define (changeo coins total denoms)
  (conde ((== coins '())
          (zeroo total))
         ((fresh (c coins^ subtotal)
            (== coins (cons c coins^))
            (membero c denoms)
            (pluso subtotal c total)
            (changeo coins^ subtotal denoms)))))
```

Issue: repeated results

```
(define result
        (run* (coins)
           (changeo coins
                    (build-num 13)
                    (list (build-num 5) (build-num 1)))))
> (map (lambda (coins) (map toint coins)) result)
'((5 5 1 1 1)
  (5 1 1 1 1 1 1 1 1)
  (1 5 5 1 1)
  (1 1 1 1 1 1 1 1 1 1 1 1 1)
  (1 5 1 1 1 1 1 1 1))
```