μ Kanren

A Minimal Functional Core for Relational Programming

Jason Hemann, Dan Friedman

September 20, 2018

(lambda () 'taylskid)

- Taylor Skidmore
- Droit
- @SirSkidmore
- github.com/taylskid

Example

```
(call/fresh
  (lambda (q)
    (== q 'olive)))
```

Example

```
(call/fresh
  (lambda (q)
    (== q 'olive)))
```

$\Rightarrow$ `olive`, or, `'((((#(0) . olive)) . 1))`

Example Cont

```
(conj
 (call/fresh (lambda (a) (== a 'oil)))
 (call/fresh (lambda (b)
               (disj (== b 'olive)
                     (== b 'canola)))))
```

Example Cont

```
(conj
 (call/fresh (lambda (a) (== a 'oil)))
 (call/fresh (lambda (b)
              (disj (== b 'olive)
                    (== b 'canola)))))
```

$\Rightarrow$ `'(oil oil)`, or,

```
'((((#(1) . olive) (#(0) . oil)) . 2)
  (((#(1) . canola) (#(0) . oil)) . 2))
```

Bonus Example

```
(call/fresh
 (lambda (q)
   (call/fresh
    (lambda (x)
      (call/fresh
       (lambda (y)
         (conj (== `(,x ,y) q)
               (disj (conj (== x 'split)
                           (== y 'pea))
                     (conj (== x 'red)
                           (== y 'bean))))))))))
```

Bonus Example Cont

```
'((split pea) (red bean))

'((((#(2) . pea) (#(1) . split)
   (#(0) #(1) #(2))) . 3)
  (((#(2) . bean) (#(1) . red)
   (#(0) #(1) #(2))) . 3))
```

Goals

Logic programming equivalent of a predicate.
Goals succeed or fail (`#s` or `#u`), and could cause the state of the program to grow (i.e. variable 'bindings')

- `(== 5 5)` $\Rightarrow$ `#s`
- `(== 5 4)` $\Rightarrow$ `#u`
- `(== q 5)` $\Rightarrow$ `#s` (state grows)
- `(== 5 q)` $\Rightarrow$ `#s` (state grows)

Ext. Goal Example

```
(call/fresh
  (lambda (q)
    (conj (== q 'oil)
          (== q 'butter))))
```

$\Rightarrow$

```
#u
```

Streams

Scheme implementation of the List Monad.
Output of a μ Kanren program is a stream of states from 'successful' goals

- `'((((#(0) . olive)) . 1))`
- `'((((#(1) . olive) (#(0) . oil)) . 2) (((#(1) . canola) (#(0) . oil)) . 2))`

If I don't write something here Beamer formatting breaks

- Variables
- Streams
- Streams utils
- Goal constructors

Variables

Varaibles are vectors that track their De Bruijn index.

```
(define (var c) (vector c))
(define (var? x) (vector? x))
(define (var=? x1 x2) (= (vector-ref x1 0)
                         (vector-ref x2 0)))
```

Variables and State

```
(define (walk u s)
  (let ((pr (and (var? u)
                 (assp (lambda (v) (var=? u v)) s))))
    (if pr (walk (cdr pr) s) u)))

(define (ext-s x v s) `((,x . ,v) . ,s))

(define empty-state '(() . 0))
```

State Examples

- `(ext-s (var 0) 5 '())`
  $\Rightarrow$ `'((#(0) . 5))`
- `(walk (var 0) (ext-s (var 0) 5 '()))` $\Rightarrow$ 5
- `(ext-s (var 1) 5 (ext-s (var 0) (var 1) '()))`
  $\Rightarrow$ `'((#(1) . 5) (#(0) . #(1)))`
- `(walk (var 0) foo)` $\Rightarrow$ 5

Goal ≡

```
(define (== u v)
  (lambda (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))

(define (unit s/c) (cons s/c mzero))
(define mzero '())
```

unify

```
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      (else (and (eqv? u v) s)))))
```

unify / $\equiv$ example

```
((call/fresh (lambda (q) (== q 'oil))) empty-state)
```

$\Rightarrow$

```
(unify (var 0) 'oil '())
```

$\Rightarrow$

```
'(((#(0) . 'oil)) . 1)
```

call/fresh

```
(define (call/fresh f)
  (lambda (s/c)
    (let ((c (cdr s/c)))
      ((f (var c)) `(,(car s/c) . ,(+ c 1))))))
```

call/fresh example

```
((call/fresh (lambda (q) (== q 'oil)))
 empty-state)
```

state: `'(() . 0)` ⇒ `'(() . 1)`

```
((== (var 0) 'oil) '(() . 1))
```

disj / conj

```
(define (disj g1 g2)
  (lambda (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2)
  (lambda (s/c) (bind (g1 s/c) g2)))
```

mplus

```
(define (mplus s1 s2)
  (cond
    ((null? s1) s2)
    (else (cons (car s1) (mplus (cdr s1) s2)))))
```

bind

```
(define (bind s g)
  (cond
    ((null? s) mzero)
    (else (mplus (g (car s)) (bind (cdr s) g)))))
```

Conclusions

```
(conj
 (call/fresh (lambda (a) (== a 'oil)))
 (call/fresh (lambda (b)
              (disj (== b 'olive)
                    (== b 'canola)))))
```

$\Rightarrow$

```
'((((#(1) . olive) (#(0) . oil)) . 2)
  (((#(1) . canola) (#(0) . oil)) . 2))
```

Write some macros

```
(define-syntax Zzz
  (syntax-rules ()
    ((_ g) (lambda (s/c) (lambda () (g s/c))))))

(define-syntax conj+
  (syntax-rules ()
    ((_ g) (Zzz g))
    ((_ g0 g ...) (conj (Zzz g0) (conj+ g ...)))))

(define-syntax disj+
  (syntax-rules ()
    ((_ g) (Zzz g))
    ((_ g0 g ...) (disj (Zzz g0) (disj+ g ...)))))
```

Making μ Kanren usable

```
(run* (q)
  (fresh (x y)
    (== `(,x ,y) q)
    (conde
      ((== x 'split) (== y 'pea))
      ((== x 'red) (== y 'bean)))))
```

$\Rightarrow$

```
'((split pea) (red bean))
```

Write some macros

```
(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) ...)
     (disj+ (conj+ g0 g ...) ...))))

(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g ...) (conj+ g0 g ...))
    ((_ (x0 x ...) g0 g ...)
     (call/fresh
       (lambda (x0) (fresh (x ...) g0 g ...))))))
```

Macro Expansion

```
(fresh (x y)
  ... )
```

$\Rightarrow$

```
(call/fresh (lambda (x) (fresh (y) ...)))
```

$\Rightarrow$

```
(call/fresh (lambda (x)
  (call/fresh (lambda (y) ...))))
```

Halfway...

```
((call/fresh
   (lambda (q)
     (fresh (x y)
       (== `(,x ,y) q)
       (conde
         ((== x 'split) (== y 'pea))
         ((== x 'red) (== y 'bean))))))
 empty-state)
```

$\Rightarrow$

```
'((((#(2) . pea) (#(1) . split)
   (#(0) #(1) #(2))) . 3)
  (((#(2) . bean) (#(1) . red)
   (#(0) #(1) #(2))) . 3))
```

Write some functions

```
(define (mK-reify s/c*) (map reify-state/1st-var s/c*))
(define (reify-state/1st-var s/c)
  (let ((v (walk* (var 0) (car s/c))))
    (walk* v (reify-s v '()))))

(define (reify-s v s)
  (let ((v (walk v s)))
    (cond
      ((var? v)
       (let ((n (reify-name (length s))))
         (cons `(,v . ,n) s)))
      ((pair? v)
       (reify-s (cdr v) (reify-s (car v) s)))
      (else s))))
```

Functions. . .

```
(define (reify-name n)
  (string->symbol
   (string-append "_" "." (number->string n))))

(define (walk* v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) v)
      ((pair? v) (cons (walk* (car v) s)
                       (walk* (cdr v) s)))
      (else v))))

(define (call/empty-state g) (g empty-state))
```

Write some macros

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x ...) g0 g ...)
     (mK-reify
      (take n (call/empty-state
                (fresh (x ...) g0 g ...)))))))

(define-syntax run*
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (mK-reify
      (take-all (call/empty-state
                  (fresh (x ...) g0 g ...)))))))
```

And, finally...

```
(run* (q)
  (fresh (x y)
    (== `(,x ,y) q)
    (conde
      ((== x 'split) (== y 'pea))
      ((== x 'red) (== y 'bean)))))
```

⇒

```
'((split pea) (red bean))
```

Why?

Why is this cool? Why should we care?

- interpreters/type checkers
- Quine generation
- Register Allocation ("Four color problem")
- "Real applications"
  - Barliman
  - MediKanren

Why?

Why is this cool? Why should we care?

- interpreters/type checkers
- Quine generation
- Register Allocation ("Four color problem")
- "Real applications"
  - Barliman
  - MediKanren

Relational Interpreters

```
(run 1 (q) (evalo 5 q)
(run 1 (q) (evalo '((lambda (x) 5) 10) q))
```

$\Rightarrow$

5

Relational Interpreters

```
(run 2 (q) (evalo q 5))
```

$\Rightarrow$

```
'(5 ((lambda (_.0) 5) _.1))
```

Type Checkers

- `(run 1 (q) (typeo 5 q)` ⇒ `Int`
- `(run 1 (q) (typeo '(lambda (x) 5) q)`
  ⇒ `'(Any -> Int)`
- `(run 1 (q) (typeo q Int)` ⇒ `'5`
- `(run 1 (q) (typeo q '(Any -> Int))`
  ⇒ `(lambda (x) 5)`

Barliman: Program Synthesis

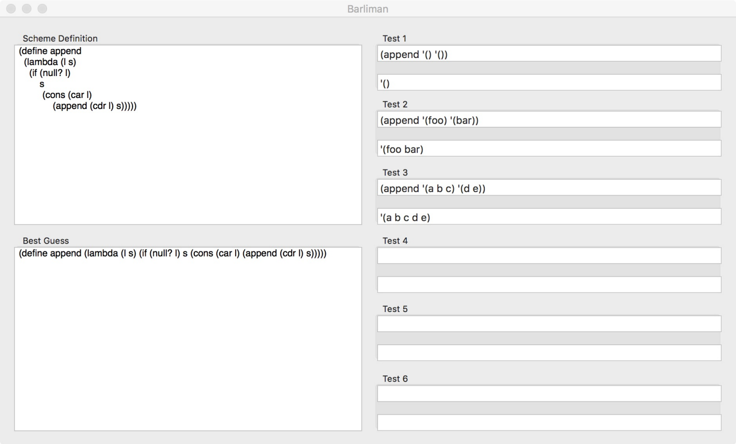

Figure: Barliman

https://github.com/webyrd/Barliman

Barliman: Under the hood

```
(run 1 (defn)
  (fresh (body)
    (absento 1 defn) (absento 2 defn) ...
    (== defn `(append (lambda (xs ys) ,body)))
    (evalo
      `(letrec (,defn)
         (list (append '() '())
               (append '(1) '(2))
               (append '(1 2) '(3 4))))
      '(() (1) (1 2) (1 2 3 4)))))
```

MediKanren

http://www.uab.edu/mix/stories/
a-high-speed-dr-house-for-medical-breakthroughs

MediKanren

Relations over SemMedDB

- diseases and symptoms
- drugs and symptoms
- drugs and diseases

https://github.com/webyrd/mediKanren