μKanren

A Minimal Functional Core for Relational Programming

Jason Hemann, Daniel P. Friedman; 2013

Lucas Stadler, Jakob Matthes, Eugen Rein

- Motivation
- Beispiele
- Konzepte
- Implementierung
- Zusammenfassung

Motivation

- relationale Programmierung
- Scheme (39 Zeilen Code)
- Einbettung in anderen Sprachen
- Erweiterbarkeit
- Eleganz

Scheme

```
(+ 1 2)
; 3
```

```
(+ 1 (* 3 4))
; 13
```

```
(append '(1 2 3) '(4 5 6))
; (1 2 3 4 5 6)
```

```scheme
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

```
(define (append l r)
  (cond
    ((null? l) r)
    (else (cons
            (car l)
            (append (cdr l) r)))))
```

Beispiele

```
(run* (q)
  (=== q 5))
; (5)
```

```
(run* (q)
  (=== q 5))
; (5)
```

```
(run* (q)
  (=== q q))
; (_.0)
```

```
(run* (q)
  (disj
    (=== q 5)
    (=== q 6)))
; (5 6)
```

```
(run* (q)
  (disj
    (=== q 5)
    (=== q 6)))
; (5 6)
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
; ()
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
; ()
```

```
(run* (q)
  (fresh (r)
    (conj
      (=== q r)
      (=== q 42))))
; 42
```

```
(run* (q)
  (fresh (r)
    (conj
      (=== q r)
      (=== q 42))))
; 42
```

```
(run* (q)
  (fresh (y)
    (conj
      (=== q `(5 ,y))
      (disj
        (=== y 6)
        (=== y 7)))))
; ((5 6) (5 7))
```

Konzepte

```
(run* (q)
  (=== q 5))
```

```
(run* (q)
  (=== q 5))
```

`q` ist eine Variable, die als `#(0)` angezeigt wird

```
(var 0)
; #(0)
```

State

```
(((#(0) . 5) (#(1) . #(0)) (#(2) . (1 2 3))) . 3)
;^ Substitution                           Counter ^
```

State

```
(((#(0) . 5) (#(1) . #(0)) (#(2) . (1 2 3))) . 3)
;^ Substitution                        Counter ^
```

State

```
(((#(0) . 5) (#(1) . #(0)) (#(2) . (1 2 3))) . 3)
;^ Substitution                         Counter ^
```

State

```
(((#(0) . 5) (#(1) . #(0)) (#(2) . (1 2 3))) . 3)
;^ Substitution                     Counter ^
```

State

```
(((#(0) . 5) (#(1) . #(0)) (#(2) . (1 2 3))) . 3)
;^ Substitution                         Counter ^
```

Goal

Funktion mit State als Argument

```
(=== (var 0) 5)
; <goal>
```

Stream

Liste von States

```
((=== (var 0) 5) '(() . 1))
; ((((#(0) . 5)) . 1))
```

Implementierung

```
(run* (q)
  (=== q 5))
```

```
(run* (q)
   (=== q 5))
```

```
(call/fresh
  (lambda (q)
    (=== q 5)))
```

```
((call/fresh
  (lambda (q)
    (=== q 5)))
 empty-state)
```

```
((call/fresh
  (lambda (q)
    (=== q 5)))
 empty-state)
; ((((#(0) . 5)) . 1))
```

```
(run* (q)
  (=== q 5))
```

```
(run* (q)
  (=== q 5))
```

```
((=== (var 0) 5) '(() . 1))
; ((((#(0) . 5)) . 1))
```

```
(===  (x          2          3)
      (1          y          3))
; ({x = 1, y = 2})
```

```
((=== `(,(var 0) 2         3)
      `(1         ,(var 1) 3)) '(() . 2))
```

```
((=== `(,(var 0) 2          3)
      `(1          ,(var 1) 3)) '(() . 2))
; (((#(0) . 1) (#(1) . 2)) . 2))
```

```
((=== (var 0) 6) `((,(var 0) . 5) . 1))
; ()
```

```
((=== (var 0) 6) `((,(var 0) . 5) . 1))
; ()
```

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

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

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

```
(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)))))
```

```
(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)))))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
; 6
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . 6)))
; 6
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
; 7
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7)))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

(walk #(2) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))

; 7
```

```
(walk #(1) ((#(0) . 5)
            (#(1) . #(2))
            (#(2) . 7))
; 7
```

```
(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)))))
```

```
(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)))))
```

```scheme
(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)))))
```

```
(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)))))
```

```
(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)))))
```

```
(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)))))
```

```
(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)))))
```

```scheme
(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)))))
```

```
(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)))))
```

```scheme
(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)))))
```

```
(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)))))
```

```
(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)))))
```

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

```
((=== `(,(var 0) 2 3) `(1 ,(var 1) 3)) '(() . 2))
; ((((#(0) . 1) (#(1) . 2)) . 2))
```

```
(run* (q)
  (disj
    (=== q 5)
    (=== q 6)))
```

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

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

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   (else (cons (car $1)
               (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   (else (cons (car $1)
               (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   (else (cons (car $1)
               (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    (else (cons (car $1)
                (mplus (cdr $1) $2)))))
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
```

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

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

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

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

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

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

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

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
```

```
(bind ((=== q 5) s/c)
      (=== q 6))
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
```

```
(bind ((((#(0) . 5)) . 1))
      (=== (var 0) 6))
```

```
(run* (q)
  (conj
    (=== q 5)
    (=== q 6)))
```

```
(bind ((((#(0) . 5)) . 1))
      (=== (var 0) 6))
```

```
((=== (var 0) 6) ((((#(0) . 5)) . 1)))
```

```
(define (fives x)
  (disj
    (=== x 5)
    (fives x)))
```

```
(define (fives x)
  (disj
    (=== x 5)
    (fives x)))
; (while (not end-of-the-universe)
;    ...)
```

```
(define (fives x)
  (disj
    (=== x 5)
    (lambda (s/c)
      (lambda ()
        ((fives x) s/c)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   (else (cons (car $1) (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   ((procedure? $1) (lambda () (mplus ($1) $2)))
   (else (cons (car $1) (mplus (cdr $1) $2)))))
```

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

```
(define (bind $ g)
  (cond
   ((null? $) mzero)
   ((procedure? $) (lambda () (bind ($) g)))
   (else (mplus (g (car $)) (bind (cdr $) g)))))
```

```
(define (fives x)
  (disj
    (=== x 5)
    (lambda (s/c)
      (lambda ()
        ((fives x) s/c)))))
```

```
(define (sixes x)
  (disj
    (=== x 6)
    (lambda (s/c)
      (lambda ()
        ((sixes x) s/c)))))
```

```
(run* (x)
  (disj
    (fives x)
    (sixes x)))
```

```
(run* (x)
  (disj
    (fives x)
    (sixes x)))
; only fives
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   ((procedure? $1) (lambda () (mplus ($1) $2)))
   (else (cons (car $1) (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   ((procedure? $1) (lambda () (mplus ($1) $2)))
   (else (cons (car $1) (mplus (cdr $1) $2)))))
```

```
(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   ((procedure? $1) (lambda () (mplus $2 ($1))))
   (else (cons (car $1) (mplus (cdr $1) $2)))))
```

```
(run* (x)
  (disj
    (fives x)
    (sixes x)))
; fives and sixes!
```

```
(run* (x)
  (disj
    (fives x)    ; (5 ...)
    (sixes x)))
```

```
(run* (x)
  (disj
    (fives x)
    (sixes x)))  ; (6 ...)
```

```
(run* (x)
  (disj
    (fives x)
    (sixes x)))
```

Grenzen

- Negation (`=/=` in cKanren)
- Zahlen (miniKanren, cKanren)
- "Interface" (**`run*`** und andere mit Macros)
- Performance (u.a. Tabling in miniKanren)

Zusammenfassung

- Eleganz
- Erweiterbarkeit (cKanren, αKanren, rKanren)
- Einbettung in anderen Sprachen (> 20)
- relationale Programmierung

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

(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 (=== 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 ,())

(define (unify uv vv s)
  (let ((u (walk uv s)) (v (walk vv 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)))))

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

(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)))

(define (mplus $1 $2)
  (cond
   ((null? $1) $2)
   ((procedure? $1) (lambda () (mplus $2 ($1))))
   (else (cons (car $1) (mplus (cdr $1) $2)))))

(define (bind $ g)
  (cond
   ((null? $) mzero)
   ((procedure? $) (lambda () (bind ($) g)))
   (else (mplus (g (car $)) (bind (cdr $) g)))))
```

Quellen

Jason Hemann und Daniel P. Friedman, µKanren: A Minimal Functional Core for Relational Programming, 2013.

Daniel P. Friedman, William E. Byrd und Oleg Kiselyov, The Reasoned Schemer, Oktober 2005.

William E. Byrd, Relational Programming in miniKanren (Part 1/2).

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd und Daniel P. Friedman, cKanren: miniKanren with Constraints, 2011.

William E. Byrd, Relational Programming in miniKanren: Techniques, Applications, and Implementations, September 2009.

```
(define + -)
```

```
(define + -)
; have fun!
```

```
(define + -)
; have fun!
; any questions?
```