ucKanren: Micro Constraint Kanren

Joshua Cox
Indiana University
joshcox@indiana.edu

1. Introduction

Logic programming allows us to solve a variety of problems by allowing a user to tell what a solution should be rather than how to get a solution. Typically, logic programming languages use search to find the solutions for a particular description and use unification as a constraint solver. Unfortunately, this search and unification combination leads to slow execution. By adding a general constraint framework, we can prune the search tree, fail sooner, and perhaps intuit additional information given the current state of affairs.Here we present a constraint framework for microKanren.

<*> ::=

```
(require C311/trace C311/pmatch)
(provide (all-defined-out))
```

***<MicroKanren>* ::=**

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

(define ($-append-map g $)
  (cond
   ((procedure? $) (lambda () ($-append-
map g ($))))
   ((null? $) '())
   (else ($-append (g (car $)) ($-append-
map g (cdr $))))))

(define mzero (lambda () '()))

(define unit (lambda (x) (cons x (mzero))))
```

***<Goals>* ::=**

```
(define (call/fresh f)
  (lambdas (a : s/c d c)
    (let ((c (cdr s/c)))
      ((f (var c)) (make-a (cons (car s/c)
(+ 1 c)) d c)))))

(define (disj g1 g2) (lambdas (a : s/c d c) ($-
append (g1 s/c) (g2 s/c))))

(define (conj g1 g2) (lambdas (a : s/c d c) ($-
append-map g2 (g1 s/c))))

(define ==
  (lambda (u v)
    (goal-construct (==constraint u v))))

(define ==constraint
  (lambda (u v)
    (lambdas (a : s d c)
      (let ((s^ (unify u v (car s))))
        (if s^
            (if (eq? s^ (car s)) a (make-
a (cons s^ (cdr s)) d c))
            #f))))))
```

***<Solvers>* ::=**

```
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((eqv? 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 #f))))

(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
     ((var? v) (eq? v x))
     ((pair? v) (or (occurs? x (car v) s) (occur
     (else #f))))
```

***<Convenience>* ::=**

```
(define identity (lambdas(a) a))
(define composem
  (lambda (f f^)
    (lambdas (a)
```

```
      (let ((a (f a)))
        (and a (f^ a))))))
```

<*State*> ::=

```
(define make-a
  (lambda (s d c)
    (cons s (cons d c))))

(define-syntax lambdas
  (syntax-rules ()
    ((_ (a : s d c) body)
     (trace-lambda lambdas (a)
       (let ((s (car a)) (d (car (cdr a))) (c (cdr (cdr a))))
         body)))
    ((_ (a) body) (lambda (a) body))))

(define (empty-s) '(() . 0))
(define (empty-d) '())
(define (empty-c) '())
(define empty-state (lambda () (make-
a (empty-s) (empty-d) (empty-c))))

(define (var n) n)

(define (var? n) (number? n))

(define (ext-s x v s) (if (occurs? x v s) #f (cons (cons x v) s)))

(define (ext-d x fd d) (cons (cons x fd) d))
```

```
(define any/var?
  (lambda (x)
    (cond
      ((var? x) #t)
      ((pair? x) (or (any/var? (car x)) (any/var? (cdr x))))
      (else #f))))

(define (walk u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (walk (cdr pr) s) u)))

(define walk*
  (lambda (v a)
    (let ((v (walk v (car a))))
      (cond
        ((var? v) v)
        ((pair? v) (cons (walk* (car v) a) (walk* (cdr v) a)))
        (else v)))))
```

<*Constraint_Framework*> ::=

```
(define process-prefix (make-parameter #f))
(define enforce-constraints (make-parameter #f))
(define run-constraints (make-parameter #f))

(define goal-construct
  (lambda (f)
    (lambda (a)
      (cond
        ((f a) => unit)
        (else (mzero))))))
```

<*Reification*> ::=

```
(define (reify-var0 s/c)
  (let ((v (walk* (var 0) (car s/c))))
    (walk* v (make-a (reify-s v '()) '() '()))))

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

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

<*Running_MicroKanren*> ::=

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

(define (pull $) (if (procedure? $) (pull ($)) $

(define (take n)
  (lambda ($)
    (cond
      ((zero? n) '())
      (else
       (let (($ (pull $)))
         (cond
           ((null? $) '())
           (else
            (cons (car $)
              ((take (- n 1)) (cdr $)))))))))))
```

<*Syntactic_Sugar*> ::=

```
(define-syntax inverse-eta-delay
  (syntax-rules ()
    ((_ g) (lambda (s/c) (lambda () (g s/c))))))
```

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

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

(define succeed (lambda (s/c) (list s/c)))

(define fail (lambda (s/c) '()))

(define (ifte g0 g1 g2)
  (lambda (s/c)
    (let loop (($ (g0 s/c)))
      (cond
        ((procedure? $) (lambda () (loop ($))))
        ((null? $) (g2 s/c))
        (else ($-append-map g1 $))))))

(define (once g)
  (lambda (s/c)
    (let loop (($ (g s/c)))
      (cond
        ((procedure? $) (lambda () (loop ($))))
        ((null? $) '())
        (else (list (car $)))))))

(define (call/project x f)
  (lambda (s/c)
    ((f (walk* x (car s/c))) s/c)))


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

(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) (g0* g* ...) ...)
     (inverse-eta-delay
      (disj+ (conj+ g0 g ...) (conj+ g0* g* ...) ...)))))

(define-syntax run
  (syntax-rules ()
    ((_ n (q) g0 g ...)
     (map reify-var0
          ((take n)
           (call/empty-state (fresh (q) g0 g ...

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

(define-syntax ifte*
  (syntax-rules ()
    ((_ (g0 g ...)) (conj+ g0 g ...))
    ((_ (g0 g1 g ...) (h0 h ...) ...)
     (ifte g0 (conj+ g1 g ...) (ifte* (h0 h ...)

(define-syntax conda
  (syntax-rules ()
    ((_ (g0 g ...) (h0 h ...) ...)
     (inverse-eta-delay
      (ifte* (g0 g ... succeed) (h0 h ... succee

(define-syntax condu
  (syntax-rules ()
    ((_ (g0 g ...) (h0 h ...) ...)
     (conda ((once g0) g ...) ((once h0) h ...)
```