Concurrent Logic Programming

an exploration of miniKanren in FLENG PCN

About me

- Sjoerd Dost (github.com/deosjr)
- 10 years of Golang
- eBay, Northvolt
- Prolog hobbyist
- Lisp enthousiast

With thanks to

- Walter Schulze, for introducing me to miniKanren
- Ron Evans, for recommending this devroom
- Peter Saxton, for enduring a first draft
- Felix Winkelmann, for answering all my questions on IRC

> One interesting aspect of the miniKanren implementation is that the code is inherently thread-safe and—at least in theory—trivially parallelizable. Of course, parallelizing the code without making it slower is not trivial, given that each thread or process must be given enough work to make up for the overhead of parallelization. Still, this is an area of miniKanren implementation that I hope will receive more attention and experimentation.
>
> — Will Byrd[1]

---

[1] minikanren.org/minikanren-and-prolog.html

Language introductions

Languages

- FLENG
- μKanren
- Prolog

Prolog

- logic programming language
- declarative
- depth-first search
- pure core
- miniKanren implementations[1]

[1] github.com/mndrix/microkanren-prolog, aphyr.com/posts/354-unifying-the-technical-interview (via lisp)

Prolog meta-interpreter

```Prolog
prove(true).

prove( (Goal1, Goal2) ) :-
    prove(Goal1),
    prove(Goal2).

prove(Goal) :-
    clause(Goal, Body),
    prove(Body).
```

Prolog meta-interpreter

Prolog

```
prove(true).                        % disjunction

prove( (Goal1, Goal2) ) :-
    prove(Goal1),                   % conjunction
    prove(Goal2).

prove(Goal) :-
    clause(Goal, Body),             % builtin lookup
    prove(Body).
```

Prolog meta-interpreter

- very minimal, but useful
- easy access to modify goal evaluation (DFS -> BFS)
- obvious targets for parallelization (AND / OR)
- access to unification requires extention

μKanren

- functional
- substitutions, goals, streams
- no backtracking
- logically pure
- deals with infinite answers

μKanren

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

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

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

(define (unit st) (cons st mzero))
(define mzero '())

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

```scheme
(define (equalo u v)
  (lambda (st)
    (let ((sub (unify u v (car st))))
      (if sub (unit `(,sub . ,(cdr st))) mzero))))

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

(define (disj g1 g2) (lambda (st) (mplus (g1 st) (g2 st))))
(define (conj g1 g2) (lambda (st) (bind (g1 st) g2)))

(define (mplus str1 str2)
  (cond
    ((null? str1) str2)
    ((procedure? str1) (lambda () (mplus str2 (str1))))
    (else (cons (car str1) (mplus (cdr str1) str2)))))

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

µKanren

```Scheme
(() . 0 )                ; empty state

(((#(0) . 5)) . 1)   ; var 0 bound to 5, var counter at 1

(equalo x 5)             ; returns a goal, which is a function
; apply goal to empty state
(call/fresh (x) (equalo x 5) (() . 0))
; returns a stream of states as a list
> ( (((#(0) . 5)) . 1) )
```

Dealing with infinite answers

```Prolog
fives(X) :- X=5 ; fives(X).
sixes(X) :- X=6 ; sixes(X).
fives_and_sixes(X) :- fives(X) ; sixes(X).
```

example taken from J. Hemann, D. P. Friedman. *μKanren: a Minimal Functional Core for Relational Programming*, 2013[1]

[1] webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf

Dealing with infinite answers

```prolog
fives(X) :- X=5 ; fives(X).
sixes(X) :- X=6 ; sixes(X).
fives_and_sixes(X) :- fives(X) ; sixes(X).
```

Prolog

```cli
> fives(X).
X = 5 ;
X = 5 ;
X = 5 ;
...
```

cli

Dealing with infinite answers

```prolog
fives(X) :- X=5 ; fives(X).
sixes(X) :- X=6 ; sixes(X).
fives_and_sixes(X) :- fives(X) ; sixes(X).
```

Prolog

```cli
> sixes(X).
X = 6 ;
X = 6 ;
X = 6 ;
...
```

cli

Dealing with infinite answers

```prolog
fives(X) :- X=5 ; fives(X).
sixes(X) :- X=6 ; sixes(X).
fives_and_sixes(X) :- fives(X) ; sixes(X).
```

Prolog

```cli
> fives_and_sixes(X).
X = 5 ;
X = 5 ;     % not what we want!
X = 5 ;
...
```

cli

Dealing with infinite answers

```prolog
fives(X) :- fives(X) ; X=5.
sixes(X) :- X=6 ; sixes(X).
fives_and_sixes(X) :- fives(X) ; sixes(X).
```

Prolog

```cli
> fives_and_sixes(X).
ERROR: Stack limit (1.0Gb) exceeded
...
```

cli

Dealing with infinite answers

```Scheme
(define (fives x) (disj+ (equalo x 5) (fives x)))
(define (sixes x) (disj+ (equalo x 6) (sixes x)))
(define (fives_and_sixes x) (disj+ (fives x) (sixes x)))
```

```cli
> (run 5 (x) (fives_and_sixes x))
(5 6 5 6 5)
```

Dealing with infinite answers

```scheme
(define (fives x) (disj+ (fives x) (equalo x 5)))
(define (sixes x) (disj+ (equalo x 6) (sixes x)))
(define (fives_and_sixes x) (disj+ (fives x) (sixes x)))
```

```cli
> (run 5 (x) (fives_and_sixes x))
(5 6 5 6 5)     ; no problem!
```

Dealing with infinite answers

```Scheme
(define (fives x)  (disj+ (equalo x 5) (fives x)))
(define (sixes x)  (disj+ (equalo x 6) (sixes x)))
(define (sevens x) (disj+ (equalo x 7) (sevens x)))
(define (test x) (disj+ (fives x) (sixes x) (sevens x)))
```

```cli
> (run 9 (x) (test x))
(5 6 5 7 5 6 5 7 5)
```

Dealing with infinite answers

```Scheme
(define-syntax Zzz
  (syntax-rules ()
    ((_ g) (lambda (st) (lambda () (g st))))))
```

- streams as lists, immature streams as thunks
- disj+ wraps disj in Zzz and generalizes
- binary trampolining
- Go programmer thinks: channels![1]

[1] github.com/awalterschulze/gominikanren

FLENG

- concurrent logic programming
- guarded horn clauses
- committed choice
- supports multiple high-level languages (FGHC, Strand[1], PCN, KL1)
- single-assignment logic vars
- synchronisation using delay
- AND-parallelism
- in the public domain[2]

---

[1] gitlab.com/b2495/fleng/-/blob/master/doc/strand-book.pdf

[2] gitlab.com/b2495/fleng/-/blob/master/doc/LICENSE

FLENG

```
-initialization(main).
```

```
producer(N, Out) :-
    N > 0 |
    Out = [N|Out2],
    N1 is N - 1,
    producer(N1, Out2).

producer(_, Out) :-
    otherwise |
    Out = [].
```

```
consumer([X|More]) :-
    writeln(X),
    consumer(More).

consumer([]).

main :-
    producer(10, Stream),
    consumer(Stream).
```

FLENG

FLENG PCN

```
main() {||
    producer(10, stream);
    consumer(stream)
}
producer(n, out) {?
    n > 0 -> {
        out = [n|out2];       // create new stream element
        producer(n - 1, out2)
    },
    default -> out = []       // close stream
}
consumer(in) {?
    in ?= [x|more] -> {
        writeln(x);           // writes 10, 9, 8, ..., 1
        consumer(more)
} }
```

Strand

Pseudocode

```
interpreter()
    for each initial process P
        put_process(P)                          { put P in process pool }
    repeat
        P := get_process()                      { get a process from pool }
        if (is_predefined(P)) execute(P)        { predefined process }
            else reduce(P)                      { otherwise, do reduction }
    until(empty pool)

reduce(P)
    COMMIT := False                             { initialize Flags }
    repeat
        R  := pick_untried_rule(P, S)           { get a rule from S }
        R1 := fresh_copy(R)                     { copy the rule to R1 }
        M  := CMatch(P,R1)                      { execute match/guard }
        if (M = Theta) then                     { CMatch succeeds? }
            COMMIT := True                      { finished looking }
            spawn_body(R1, Theta)               { add processes to pool }
        until (COMMIT) or (all_rules_tried(P))  { reduced or done }
        if (not COMMIT) then put_process(P)     { return process to pool }
```

flengKanren

flengKanren

- immature streams as unbound logic variables
- stream as pair of str-fill, two logic vars
- generalised alternating goals in disj (m++)
- OR-parallelism using buffered stream resolution

Demo Time