BFS search in miniKanren

KUANG-CHEN LU, Indiana University
WEIXI MA, Indiana University
DANIEL P. FRIEDMAN, Indiana University

The syntax of a programming language should reflect its semantics. When using a disjunction operator in relational programming, a programmer would expect all clauses of this disjunction to share the same chance of being explored, as these clauses are written in parallel. The existing disjunctive operators in miniKanren, however, prioritize their clauses by the order of which these clauses are written down. We have devised a new search strategy that searches evenly in all clauses. Based on our statistics, miniKanren slows down by a constant factor after applying our search strategy. (tested with very-recursiveo, need more tests)

ACM Reference Format:
Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. BFS search in miniKanren. 1, 1 (March 2019), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

miniKanren is a relational programming language embedable in many languages[cite miniKanren.org?].

The version of miniKanren in *The Reasoned Schemer, 2nd Edition* features an efficient and complete search strategy – interleaving depth-first search (iDFS). iDFS biases toward left conde lines. So miniKanren programmers sometimes need to organize their conde lines carefully. We proposed two search strategies and their implementations. The first strategy is breadth-first search. The second one is a modified iDFS.

OUTLINE:

(About miniKanren)

(Why the left clauses are explored more frequently?)

(How to solve the problem?)

(Summary of later sections)

2 COST OF ANSWERS

The *cost* of an answer is the number of relation applications needed to find the answer. This idea is borrowed from Silvija Seres's work [*]. Now we illustrate the costs of answers by running a miniKanren relation. Fig. 1 defines the relation `repeato` that relates a term `x` with a list whose elements are all `xs`.

Consider the following `run` of `repeato`.

```
> (run 4 q
    (repeato '* q))
'(() (*) (* *) (* * *))
```

Authors' addresses: Kuang-Chen LuIndiana University; Weixi MaIndiana University; Daniel P. FriedmanIndiana University.

XXXX-XXXX/2019/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(defrel (repeato x out)
  (conde
    [(== '() out)]
    [(fresh (res)
       (== `(,x . ,res) out)
       (repeato x res))]))
```

Fig. 1. `repeato`

The above `run` generates 4 answers. All are lists of `*`s. The order of the answers reflects the order miniKanren discovers them: the leftmost answer is the first one. This result is not suprising: to generate the first answer, `'()`, miniKanren needs to apply `repeato` only once and the later answers need more recursive applications. In this example, the cost of each answer is the same as one more than the number of `*`s: the cost of `'()` is 1, the cost of `'(*)` is 2, and so on.

A list of answer is in the *cost-respecting* order if no answer occurs before another answer of a lower cost. In the above example, the answers are cost-respecting. The iDFS search, however, does not generate cost-respecting answers in general. As an example, consider the following `run` of `repeato`.


```
> (run 12 q
    (conde
      [(repeato 'a q)]
      [(repeato 'b q)]
      [(repeato 'c q)]))
'(() (a) ()
  (a a) () (a a a)
  (b) (a a a a) (c)
  (a a a a a) (b b) (a a a a a a))
```

The results are not cost-respecting. For example, `'(a a)` occurs before `'(b)` while `'(a a)` is associated with a higher cost. The problem is that iDFS strategy prioritizes the first `conde` case considerablely. In general, when every conde case are equally productive, the iDFS strategy takes $1/2^i$ answers from the i-th case, except the last case, which share the same portion as the second last one.

For the above `run`, both search strategies produces answers in increasing order of costs, i.e. both of them are *cost-respecting*. In more complicated cases, however, interleaving DFS might not produce answers in cost-repecting order. For instance, with iDFS the `run` in Fig. **??** produces answers in a seemingly random order. In contrast, the same run with BFS produces answers in an expected order (Fig. **??**).

```
> (run 12 q
    (conde
      [(repeato 'a q)]
      [(repeato 'b q)]
      [(repeato 'c q)]))
'(() () ()
  (a) (b) (c)
  (a a) (b b) (c c)
```

```
(define (append-inf s-inf t-inf)
  (cond
    ((null? s-inf) t-inf)
    ((pair? s-inf)
     (cons (car s-inf)
       (append-inf (cdr s-inf) t-inf)))
    (else (lambda ()
            (append-inf t-inf (s-inf))))))
```

Fig. 2. `append-inf` in `mk-0`

```
  (a a a) (b b b) (c c c))
```

3 BREADTH-FIRST SEARCH

In this section we change the search strategy to breadth-first search and optimize it. The whole process is completed in three steps, corresponding to 3 versions of miniKanren. We start with, `mk-0`, the miniKanren in *The Reasoned Schemer, 2nd Edition*.

3.1 from `mk-0` to `mk-1`

In `mk-0` and `mk-1`, search spaces are represented by streams of answers. Streams can be finite or infinite. Finite streams are just lists. And infinite streams are improper lists, whose last `cdr` is a thunk returning another stream. We call the `car`s the *mature* part, and the last `cdr` the *immature* part.

Streams are cost respective when they are initially constructed by `==`. However, the `mk-0` version of `append-inf` (Fig. 2) breaks cost respectiveness when its first argument, `s-inf`, is infinite. The resulting mature part contains only the mature part of `s-inf`. The whole `t-inf` goes to the resulting immature part.

The `mk-1` version of `append-inf` (Fig. 3) restores cost-respectiveness by combining the mature parts in the fashion of `append`. This `append-inf` calls its helper immediately, with the first argument, `s?`, set to `#t`, which means `s-inf` in the helper is the `s-inf` in the driver. Two streams are swapped in the third `cond` clause, where `s?` is also changed accordingly.

`mk-1` is not efficient in two aspects. `append-inf` need to copy all `cons` cells of two input streams when the first stream has a non-trivial immature part. Besides, `mk-1` computes answers of the same cost at once, even when only a portion is queried. We solves the two problems in the next subsections.

3.2 `mk-3`, optimized breadth-first search

We avoid generating same-cost answers at once by expressing BFS with a queue. The elements of the queue are delayed computation, represented by thunks. Every `mk-1` stream has zero or one thunk, so we have no interesting way to manage it. Therefore we change the representation of immature parts from thunks to lists of thunks. As as consequence, we also change the way to combine mature and immature part from `append` to `cons`.

After applying this two changes, stream representation becomes more complicated. It motivates us to set up an interface between stream functions and the rest of miniKanren. Listed in Fig. 4 are all functions being aware of the stream representation, but `take-inf` and its helper function, which is explained later. The first three functions are constructors: `empty-inf` constructs an empty stream; `unit-mature-inf`

```
(define (append-inf s-inf t-inf)
  (append-inf^ #t s-inf t-inf))

(define (append-inf^ s? s-inf t-inf)
  (cond
    ((pair? s-inf)
     (cons (car s-inf)
       (append-inf^ s? (cdr s-inf) t-inf)))
    ((null? s-inf) t-inf)
    (s? (append-inf^ #f t-inf s-inf))
    (else (lambda ()
            (append-inf (t-inf) (s-inf))))))
```

Fig. 3. `append-inf` in `mk-1`

constructs a stream with one mature solution; `unit-immature-inf` constructs a stream with one thunk. The `append-inf` in `mk-3` is relatively straightforward compared with the `mk-1` version. `append-map-inf` is more tricky on how to construct the new immature part. We can follow the approach in mk-0 and mk-1 – create a new thunk which invoke append-map-inf recursively when forced. But then we need to be careful: if we construct the thunk when the old immature part is an empty list, the resulting stream might be infinitely unproductive. Beside, all solutions of the next lowest cost in `s-inf` must be computed when the thunk is invoked. However sometimes only a portion of these solutions is required to answer a query. To avoid the trouble and the advanced computation, we choose to create a new thunk for every existing thunk. The next four functions are used only by `ifte` and `once`. Uninterested readers might skip them. `null-inf?` checks whether a stream is exausted. `mature-inf?` checks whether a stream has some mature solutions. `car-inf` takes the first solution out of a mature stream. `cdr-inf` drops the first solution of a mature stream. Finally, `force-inf` forces an a immature stream to do more computation.

The last interesting function is `take-inf` (Fig. 5). The parameter `vs` is a list of solutions. The next two parameters together represents a functional queue in a typical way. The first two `cond` lines are very similar to their counterparts in mk-0 and mk-1. The third line runs when we exaust all solutions. The forth line re-shape the queue. The fifth and last line invoke the first thunk in the queue and use the mature part of the resulting stream, `s-inf`, as the new `vs`, and enqueuing `s-inf`'s thunks.

4 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

```
(define (empty-inf) '(() . ()))
(define (unit-mature-inf v) `((,v) . ()))
(define (unit-immature-inf th) `(() . (,th)))

(define (append-inf s-inf t-inf)
  (cons (append (car s-inf) (car t-inf))
    (append (cdr s-inf) (cdr t-inf))))

(define (append-map-inf g s-inf)
  (foldr append-inf
    (cons '()
      (map (lambda (t)
             (lambda () (append-map-inf g (t))))
           (cdr s-inf)))
    (map g (car s-inf))))

(define (null-inf? s-inf)
  (and (null? (car s-inf))
       (null? (cdr s-inf))))

(define (mature-inf? s-inf)
  (pair? (car s-inf)))

(define (car-inf s-inf)
  (car (car s-inf)))

(define (force-inf s-inf)
  (let loop ((ths (cdr s-inf)))
    (cond
      ((null? ths) (empty-inf))
      (else (let ((th (car ths)))
              (append-inf (th)
                (loop (cdr ths))))))))
```

Fig. 4. Functions being aware of stream representation

```
(define (take-inf n s-inf)
  (take-inf^ n (car s-inf) (cdr s-inf) '()))

(define (take-inf^ n vs P Q)
  (cond
    ((and n (zero? n)) '())
    ((pair? vs)
     (cons (car vs)
       (take-inf^ (and n (sub1 n)) (cdr vs) P Q)))
    ((and (null? P) (null? Q)) '())
    ((null? P) (take-inf^ n vs (reverse Q) '()))
    (else (let ([th (car P)])
            (let ([s-inf (th)])
              (take-inf^ n (car s-inf)
                (cdr P)
                (append (reverse (cdr s-inf)) Q)))))))
```

Fig. 5. take-inf in mk-3-1