miniKanren with fair search

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 multiarity disjunctive operator in miniKanren, however, prioritize its clauses by the order of which these clauses are written down. We have devised two new search strategies that allocate computational effort fairly in all clauses.

(TODO: evaluation of performance)

ACM Reference Format:
Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. miniKanren with fair search. 1, 1 (March 2019), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With interleaving depth-first search, disjunctive operator allocates half computational effort to its first goal, and quater to the second, eighth to the third, and so on. Consequently, miniKanren programmers sometimes need to order their `conde` clauses carefully to prevent some clauses absorbing unproportional computational effort. For example, when a relational proof checker is used as a proof generator, the eliminational rules should comes later. Seasoned miniKanreners usually know how to make use of this unfair allocation to optimize their programs. However, we believe search strategies less sensitive to goal order can also be useful to little miniKanreners as well as seansoned ones. We propose two such search strategies, balanced interleaving DFS (biDFS) and breadth-first search (BFS), and observe how they affect the efficiency and query result of known miniKanren programs. The experiment is conducted with the miniKanren from *The Reasoned Schemer, 2nd Edition*.

(TODO: summarize following sections) Section 2 explains why iDFS biases toward left `conde` clauses. Section 3 is about the biDFS. Section 3 is about BFS.

2 WHY IDFS PUT MORE EFFORT COMPUTING LEFT CLAUSES

`conde` clauses are combined by `disj2`, right associatively. The core functionality of `disj2` is completed by `append-inf` (Fig. 1). When both input streams are infinite, the resulting mature part contains only the mature part of `s-inf`. The whole `t-inf` goes to the resulting immature part. However, `t-inf` and `s-inf` are swapped in the delayed recursive call. Hence the search strategy spend computational effort evenly in two appended streams. As `disj2` is right associative, the left clauses are nested in less calls to `append-inf`, which implies that they are more likeyly being computed.

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

```
(define (disj2 g1 g2)
  (lambda (s)
    (append-inf (g1 s) (g2 s))))

(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. 1. `disj2` and `append-inf`


```
(define (disj* gs)
  (cond
    [(null? gs) fail]
    [(null? (cdr gs)) (car gs)]
    [else
     (split gs
       (lambda (gs1 gs2)
         (disj2 (disj* gs1)
                (disj* gs2))))]))

(define (split ls k)
  (cond
    [(null? ls) (k '() '())]
    [else (split (cdr ls)
                 (lambda (l1 l2)
                   (k l2 (cons (car ls) l1))))]))
```

Fig. 2. `disj*`

3 BALANCED INTERLEAVING DFS

A `conde` expression looks like a binary tree, if we consider `disj2`s as internal nodes and `conde` clauses as a terminal nodes. In fact, each of them is one of the most unbalanced binary trees. The basic idea of our first solution, balanced interleaving DFS, is to make the tree balanced. We introduce a function `disj*` and its helper `split` (Fig. 2). `disj*` essentially construct a balanced `disj2` tree. `split` splits elements of `ls` into two even half and pass them to the continuation parameter `k`.

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

Fig. 3. `repeato`

4 BREADTH-FIRST SEARCH

In this section we change the search strategy to breadth-first search and optimize it. The whole process is completed in two steps. In the first step, from mk-0 to mk-1, BFS is introduced. In the second step, mk-1 to mk-3, BFS is optimized. The initial version, `mk-0`, is exactly the version in *The Reasoned Schemer, 2nd Edition*.

4.1 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. 3 defines the relation `repeato` that relates a term `x` with a list whose elements are all `x`s.

Consider the following `run` of `repeato`.

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

The above `run` generates 4 answers. All are lists of `*`s. The order of the answers reflects the order miniKanren discovers them: the first answer in the list is first discovered. 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. iDFS strategy is the cause, since it prioritizes the first `conde` case considerablely.

```
(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. 4. append-inf in mk-0

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


4.2 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 `cars` 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. 4) breaks cost respectiveness if its first input stream, `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. 5) 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, with `s?` flipped 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.

```
(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. 5. `append-inf` in `mk-1`

4.3 `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. 6 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` 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. 7). 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.

```
(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. 6. Functions being aware of stream representation

5 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

```
(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. 7. take-inf in mk-3-1