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)

1 INTRODUCTION

The best way to write a miniKanren program can depend on the search strategy. For example, with inteleaving depth-first search (iDFS), users sometimes need to order their conde clauses carefully to avoid starving later clauses. This is because iDFS spends half of computation effort on the first clause, a quater on the second, and so on. Seasoned miniKanreners usually know how to make use of this bias to optimize their programs. However, we believe search strategies less sensitive to conde clause order can also be useful. We propose two such search strategies, balanced interleaving DFS (biDFS) and breadth-first search (BFS), and observe how they affect the efficiency and answer order of some known miniKanren programs. The experiment is conducted within the miniKanren in *The Reasoned Schemer, 2nd Edition*.

Section 2 explains why iDFS biases toward left conde clauses. Section 2 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 likely being computed.

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

Unpublished working draft. Not for distribution.

```
48  (define (disj2 g1 g2)
49    (lambda (s)
50      (append-inf (g1 s) (g2 s)))))
51
52
53  (define (append-inf s-inf t-inf)
54    (cond
55      ((null? s-inf) t-inf)
56      ((pair? s-inf)
57       (cons (car s-inf)
58         (append-inf (cdr s-inf) t-inf)))
59      (else (lambda ()
60             (append-inf t-inf (s-inf))))))))
61
62
63
64
```

Fig. 1. disj2 and append-inf

```
65
66  (define (disj* gs)
67    (cond
68      [(null? gs) fail]
69      [(null? (cdr gs)) (car gs)]
70      [else
71       (split gs
72         (lambda (gs1 gs2)
73           (disj2 (disj* gs1)
74                  (disj* gs2))))]))
75
76  (define (split ls k)
77    (cond
78      [(null? ls) (k '() '())]
79      [else (split (cdr ls)
80                   (lambda (l1 l2)
81                     (k l2 (cons (car ls) l1))))]))
82
83
84
```

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`.

```
95   (defrel (repeato x out)
96     (conde
97       [(== '() out)]
98       [(fresh (res)
99         (== `(,x . ,res) out)
100         (repeato x res))]))
101
102
103
```

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 `xs`.

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 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 considerably. In

```
142  (define (append-inf s-inf t-inf)
143    (cond
144      ((null? s-inf) t-inf)
145      ((pair? s-inf)
146       (cons (car s-inf)
147         (append-inf (cdr s-inf) t-inf)))
148      (else (lambda ()
149             (append-inf t-inf (s-inf))))))
150
151
152                    Fig. 4. append-inf in mk-0
153
154
```

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. **??**).

```
161  > (run 12 q
162      (conde
163        [(repeato 'a q)]
164        [(repeato 'b q)]
165        [(repeato 'c q)]))
166  '(() () ()
167    (a) (b) (c)
168    (a a) (b b) (c c)
169    (a a a) (b b b) (c c c))
170
171
172
```

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 `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. 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.

```
189  (define (append-inf s-inf t-inf)
190    (append-inf^ #t s-inf t-inf))
191
192
193  (define (append-inf^ s? s-inf t-inf)
194    (cond
195      ((pair? s-inf)
196       (cons (car s-inf)
197         (append-inf^ s? (cdr s-inf) t-inf)))
198      ((null? s-inf) t-inf)
199      (s? (append-inf^ #f t-inf s-inf))
200      (else (lambda ()
201             (append-inf (t-inf) (s-inf)))))))
202
203
204
```

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 straightforwared 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.

```
236  (define (empty-inf) '(() . ()))
237  (define (unit-mature-inf v) `((,v) . ()))
238  (define (unit-immature-inf th) `(() . (,th)))
239
240  (define (append-inf s-inf t-inf)
241    (cons (append (car s-inf) (car t-inf))
242      (append (cdr s-inf) (cdr t-inf))))
243
244
245  (define (append-map-inf g s-inf)
246    (foldr append-inf
247      (cons '()
248        (map (lambda (t
249               (lambda () (append-map-inf g (t))))
250          (cdr s-inf)))
251      (map g (car s-inf))))
252
253
254  (define (null-inf? s-inf)
255    (and (null? (car s-inf))
256         (null? (cdr s-inf))))
257
258  (define (mature-inf? s-inf)
259    (pair? (car s-inf)))
260
261  (define (car-inf s-inf)
262    (car (car s-inf)))
263
264  (define (force-inf s-inf)
265    (let loop ((ths (cdr s-inf)))
266      (cond
267        ((null? ths) (empty-inf))
268        (else (let ((th (car ths)))
269                (append-inf (th)
270                  (loop (cdr ths))))))))
271
272
273
```

Fig. 6. Functions being aware of stream representation

5 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

```
283  (define (take-inf n s-inf)
284    (take-inf^ n (car s-inf) (cdr s-inf) '()))
285
286
287  (define (take-inf^ n vs P Q)
288    (cond
289      ((and n (zero? n)) '())
290      ((pair? vs)
291       (cons (car vs)
292         (take-inf^ (and n (sub1 n)) (cdr vs) P Q)))
293      ((and (null? P) (null? Q)) '())
294      ((null? P) (take-inf^ n vs (reverse Q) '()))
295      (else (let ([th (car P)])
296              (let ([s-inf (th)])
297                (take-inf^ n (car s-inf)
298                  (cdr P)
299                  (append (reverse (cdr s-inf)) Q)))))))
300
301
302
```

Fig. 7. take-inf in mk-3-1