

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 (February 2019), 4 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

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. We use the miniKanren relation `repeato` in Fig. 1 to demonstrate the cost of answers. We borrow this idea from Silvija Seres's work [*]. `repeato` relates `x` with a list whose elements are all `xs`. The run expression uses `repeato` to generate 4 lists whose elements are all `*s`. The order of the answers reflects the order miniKanren discovers them. The leftmost answer is discovered first. The order here is not suprising: to generate the answer `'()`, miniKanren needs to apply `repeato` only once. And it needs more applications of `repeato` to find the later answers that are more complicated. In this example, the cost of each answer is the same as one more than the number of `*s`. No answer of zero cost exists.

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 does not always produces answers in cost-repecting order. For instance, with `iDFS` the `run` in Fig. 2 produces answers in a seemingly random order. In contrast, the same run with BFS produces answers in an expected order (Fig. 3).

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.
XXXX-XXXX/2019/2-ART \$15.00
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

2019-02-19 23:46. Page 1 of 1–4.

, Vol. 1, No. 1, Article . Publication date: February 2019.

```

48 (defrel (repeato x out)
49   (conde
50     [(== '() out)]
51     [(fresh (res)
52       (== '(,x . ,res) out)
53       (repeato x res))]))
54
55 > (run 4 q
56   (repeato '* q))
57 '(() (*) (* *) (* * *))
58
59

```

Fig. 1. repeato

```

62 > (run 12 q
63   (conde
64     [(repeato 'a q)]
65     [(repeato 'b q)]
66     [(repeato 'c q)]))
67
68 '(() (a) ()
69   (a a) () (a a a)
70   (b) (a a a a) (c)
71   (a a a a a) (b b) (a a a a a a))
72
73

```

Fig. 2. run a program with interleaving depth-first search

```

76 > (run 12 q
77   (conde
78     [(repeato 'a q)]
79     [(repeato 'b q)]
80     [(repeato 'c q)]))
81
82 '(() () ()
83   (a) (b) (c)
84   (a a) (b b) (c c)
85   (a a a) (b b b) (c c c))
86
87

```

Fig. 3. run a program with breadth-first search

Although all answers by iDFS is not in cost-respecting order, the answers from each case are in cost-respecting order. The problem is that iDFS strategy prioritizes the first `conde` case considerably. 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.

3 CHANGE SEARCH STRATEGY

Now we change the search strategy and optimize the system in three steps. The result of these steps are mk-1, mk-2, and mk-3 respectively. The initial miniKanren is named mk-0, which is the miniKanren in TRS2.

3.1 representation of search space

In mk-0, search space is represented by streams. Streams can be finite or infinite. Finite streams are just lists. And infinite streams are improper lists, whose last cdr is a procedure which takes no arguments and returns a stream when invoked. Cars of streams are (possibly incomplete) answers. We call the cars the mature part, and the last cdr the immature part.

(connect stream with cost)

3.2 initial version

(explain mk-0's append-inf)

```
(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))))))
```

To make the search BFS, we just need to have append-inf combines mature parts in the fashion of append, and combine non-empty immature parts by constructing a new procedure.

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

3.3 stepstone of optimization

Make irrelevant parts in mK representation-independent w.r.t. search space, and combine mature part and immature part with cons.

3.4 optimization

The goal is to express BFS explicitly with queue, so that we dont have to generate all answers of the same cost at once.

Interesting changes: (1) put thunks in a list; (2) change force-inf (introduced in 4.B) so that it can make progress in all thunks (3) use a queue to manage thunks in take-inf.

4 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

Unpublished working draft.
Not for distribution.