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

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))
'(() (*) (* *) (* * *))
```

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, etc.

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

Unpublished working draft. Not for distribution.

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

Fig. 1. repeato

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.

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. ?? 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))
```

```
95    (define (append-inf s-inf t-inf)
96      (cond
97        ((null? s-inf) t-inf)
98        ((pair? s-inf)
99         (cons (car s-inf)
100           (append-inf (cdr s-inf) t-inf)))
101       (else (lambda ()
102              (append-inf t-inf (s-inf))))))
103
```

Fig. 2. append-inf in mk-0

3 CHANGE SEARCH STRATEGY

Now we change the search strategy and optimize the system. The whole process is completed in three steps, corresponding to 4 versions of miniKanren. The initial version, mk-0, is exactly 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 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. 2), a composer of stream, can break 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 newly constructed thunk.

The mk-1 version of append-inf (Fig. 3) restores cost-respectiveness by combining the mature parts in the fashion of append. 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 stream when the first one is infinite. Besides, mk-1 generates answers of the same cost at once, even when only a small portion is queried. We solves the two problems in the next two subsections.

3.2 stepstone of optimization

(eqv to Seres's BFS ?)

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

3.3 optimization

The goal is to express BFS explicitly with queue, so that we don't 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.

```
(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

4 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES