miniKanren with fair search strategies

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 more fairly in all clauses.

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

1 INTRODUCTION

When every sub-goal of a disjunction produces infinite states, the existing disjunctive operator allocates half computational effort to its first goal, quater to the second, eighth to the third, and so on. The unfairness provides both opportunity and burden: miniKanren users can place more frequently used goal at the beginning to optimize their programs; however, it might be a catastrophe if a goal that generate many useless states is placed before more important goals. Seasoned miniKanreners usually know how to utilize the unfairness to optimize their programs. However, we believe search strategies that is 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 the answer order of known miniKanren programs. The experiment is conducted with the miniKanren from *The Reasoned Schemer, 2nd Edition*.

2 RELATED WORKS

3 FAIRNESS

A search strategy is *fair* if answers of lower costs always come first. The *cost* of an answer the number of relation applications needed to verify the answer. 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 `x`s.

Consider the following `run` of `repeato`.

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

XXXX-XXXX/2019/4-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

```
> (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 order is not suprising: to generate the first answer, `'()`, miniKanren needs to apply `repeato` only once and the later answers need more relation 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.

In the above example, all search strategies look fair. However, the following example points out that iDFS is not fair.

```
> (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))
```

With iDFS, `'(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. 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. In contrast, the same run with BFS produces answers in an expected order.

```
> (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))
```

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

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

(define-syntax disj
  (syntax-rules ()
    [(disj g ...) (disj* (list g ...))]))
```

Fig. 2. `balanced-disj`

4 BALANCED INTERLEAVING DFS

Our first solution, balanced interleaving DFS (biDFS), is not fair as well. However, it is less sensitive to goal order in disjucntion and is as efficient as iDFS. This search strategy is based on the observations that a disjuction can be viewed as a binary tree, where `disj2`s are nodes and sub-goals are leaves, and that the position of a leaf has direct relation with its priority – the deeper a leaf, the lower computational effort it is shared. In iDFS, the tree is in one of the most unbalanced forms, because `disj` applys `disj2` right associatively. Hence the goal order is very important. The reason why the tree shape can determine priority is that `disj2` allocates computational effort evenly to its two sub-goals.

The key idea of biDFS is to make the tree balanced (Fig. 2). We introduce a function `disj*` and its helper `split`, and change the `disj` macro to call `disj*` immediately. `disj*` essentially construct a balanced `disj2` tree. The `split` helper splits elements of `ls` into two lists of roughly the same length, then apply `k` to them.

5 BREADTH-FIRST SEARCH

Breadth-first search (BFS) is fair. Our BFS is similar to the one by Seres et al Seres et al. [1]. We conjecture that the two strategies produce answers in the same order. ;; need more comparision

In the first subsection, we change the search strategy from iDFS to BFS. In the second subsection we optimize the search engine by using a queue to manage the search order. The two subsections corresponds to two new version of miniKanren, `mk-1` and `mk-2`. We refer to the original version from TRS2 as `mk-0`.

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

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

5.1 change search strategy from iDFS to BFS

In both `mk-0` and `mk-1`, search spaces are represented by streams of answers. The thunky streams in `mk-0` denote delayed computation, however, they do not necessary mean an increment in cost. We use the same kind of stream in `mk-1` but only put thunk at those places where the cost of following answers is increased by one.

To ease discussion, we call the `cars` of an stream its *mature* part, and the last `cdr` the *immature* part. When the stream is definitely finite, its immature part is an empty list, otherwise, it is a thunk. We sometimes say a stream is immature to mean its mature part is empty.

Streams denote cost correctly when they are constructed by `==`, `succeed`, and `fail`. However, the `mk-0` version of `append-inf` (Fig. 3) breaks cost respectiveness if its first input stream, `s-inf`, is infinite. The resulting mature part contains only the mature part of `s-inf`. If we want to encode the cost information correctly, the resulting mature part should also contains the mature part of `t-inf`.

The `mk-1` version of `append-inf` (Fig. 4) gain fairness 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 *both* input streams when the first stream is possibly infinite. 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.

5.2 optimize breadth-first search

We avoid generating same-cost answers at once by expressing BFS with a queue, whose elements thunks that return a new stream. Every `mk-1` stream has zero or one thunk, so we cannot manage it with queue in interesting way. 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 and the rest of miniKanren. Listed in Fig. 5 are all functions being aware of the stream representation, but `take-inf` and its helper function, which are 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 too-early 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. 6). The parameter `vs` is a list of solutions. The next two parameters, `P` and `Q`, together represent a queue. 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.

6 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES

[1] Silvija Seres, J Michael Spivey, and CAR Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.

```
(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. 5. 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. 6. take-inf in mk-3-1