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 disjunct to share the same chance of being explored, as these clauses are written in parallel. The existing multi-arity 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), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

miniKanren programs, especially relational interpreters, have been proven to be useful in solving many problems [1]. A subtlety in writing relational programs involving a large `cond`^{e} expression, such as interpreters, is that the order of `cond`^{e} clauses can affect the speed considerably. This is due to the biased allocation of computational resource. Interleaving DFS, the search strategy of `cond`^{e} in miniKanren [2], allocates half resource to the first `cond`^{e} clause, a quarter to the second, an eighth to the third, and so on. Under the hood, it is the unfairness of `disj`.

We address the unfairness of `disj` in this work. We propose two new search strategies, balanced interleaving DFS (biDFS) and breadth-first search (BFS). biDFS is almost fair – the maximal ratio of resource among disjunctive goals is bounded by a constant factor. BFS is completely fair. We prove that our BFS is equivalent to the BFS proposed by Seres et al [3]. We also observe how new search strategies affect the efficiency and the answer order of known miniKanren programs.

2 FAIRNESS

A search strategy is *fair* if answers of lower costs always come first. The *cost* of an answer is the number of relation applications. Now we illustrate the costs of answers by running a miniKanren relation. Fig. 1 defines a relation `repeato` which relates a term `x` with a list whose elements are all `x`s.

Consider the following `run` of `repeato`.

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

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

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 surprising: 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, every search strategy looks fair. However, the following example exposes 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 `cond`^{e} case considerably. When every `cond`^{e} 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))
```

Running the same query with biDFS results in yet another answer list. biDFS essentially organize disjunctive goals into a balanced tree. There is no way to build a balanced and complete tree of size 3, so one clause is allocated more resource than the other two.

```
> (run 12 q
    (conde
      [(repeato 'a q)]
      [(repeato 'b q)]
      [(repeato 'c q)]))
'(() ()
  (b) ()
  (b b) (a)
  (b b b) (c)
  (b b b b) (a a)
  (b b b b b) (c c))
```

If one insert a `(nevero)` as the forth clause, this run would results in the same answer list as the run with BFS. However, just making every `cond`^{e} has 2^n clauses cannot turn biDFS to BFS.

```
> (run 12 q
    (conde
      [(repeato 'a q)]
      [(repeato 'b q)]
      [(repeato 'c q)]
      [(nevero)]))
'(() () ()
  (a) (b) (c)
  (a a) (b b) (c c)
  (a a a) (b b b) (c c c))
```


3 BALANCED INTERLEAVING DFS

Our first solution, balanced interleaving DFS (biDFS), like iDFS, is not fair . However, it is less sensitive to goal order in disjunct and is as efficient as iDFS.

The reason why iDFS's `disj` prioritizes its left goals considerably is that the `disj` applys `disj2` right associatively, and that `disj2` allocates resource evenly to its two sub-goals. If a disjunct is viewed as a binary tree where `disj2`s are nodes and sub-goals are leaves, the deeper a leaf locates, the lower resource it is shared. In iDFS, the tree is in one of the most unbalanced forms.

The key idea of biDFS is to make the tree balanced. Fig. 2 shows the difference between iDFS and biDFS. 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 the two sub-lists.

4 BREADTH-FIRST SEARCH

In this section, we describe our BFS and compare it with the one from Seres et al [3]. The first subsection is devoted to introducing BFS to miniKanren. This subsection results in a new version of miniKanren, `mk-1` (we call the original version `mk-0` for short). The second subsection describes the equivalent between `mk-1` and Silvija's BFS. In the third and last subsection, we optimize `mk-1` with the help of a queue, which results in `mk-2`, the final BFS version.

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

4.1 change search strategy from iDFS to BFS

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

For convenience, we call the `car`s of a stream as its *mature* part, and its last `cdr` as its *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 the rule when its first input stream, `s-inf`, has a non-trivial immature part. In this case, the resulting mature part contains only the mature part of `s-inf`. If we want to describe the cost information with thunks, the resulting mature part should also contain 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 indicates whether `s-inf` and `t-inf` haven't been swapped in the driver. `s-inf` and `t-inf` are swapped in the third `cond` clause, where `s?` is flipped accordingly.

`mk-1` is inefficient in two aspects. `append-inf` need to copy all `cons` cells of *both* input streams when the first stream is has a non-trivial immature part. Besides, `mk-1` computes answers of the same cost at once, even when only a small portion is queried. We solve the two problems in the next subsections.

```
(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

4.2 compare our BFS with Seres's

;; under construction

4.3 optimize breadth-first search

We avoid generating same-cost answers at once by expressing BFS with a queue, whose elements are thunks that return a new stream. Every `mk-1` stream has zero or one thunk, so it is uninteresting to manage them with the queue. Therefore, we change the representation of immature parts from thunks to thunk lists. As a side effect, it is no longer convenient to combine the mature and immature part with `append`, which would mix answers and thunks in the same list. We choose `cons` as an alternative to `append`.

After applying these two changes, stream representation becomes more complicated, which 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, except `take-inf` and its helper function (they are explained later).

The first three functions are constructors: `empty-inf` makes an empty stream; `unit` makes a stream with one mature solution; `step` makes a stream with one thunk.

`append-inf` combines each sub-parts with `append`.

`append-map-inf` ...

The next four functions are only depended on by `ifte` and `once`. `null-inf?` checks whether a stream is exhausted. `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 immature stream to do more computation.

`unit`, `append-map-inf`, `empty-inf` and `append-inf` form a *MonadPlus*, where they correspond to `unit`, `bind`, `mzero`, and `mplus` respectively.

The last interesting function is `take-inf` (Fig. 6). Its first parameter, `vs`, is a list of solutions. The next two parameters, `P` and `Q`, together represents a queue. The first two `cond` lines are very similar to their counterparts in mk-0 and mk-1. The third line runs when both the answer list `vs` and the queue are empty, which means we have found all the answers. The fourth line re-shape the queue. The last line invokes 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.

5 QUANTITATIVE EVALUATION

;; Kuang-Chen and Weixi plan to put '(I love you), quines, appendo, and reverso here

6 RELATED WORKS

;; under construction

Edward points out a disjunct would be 'fair' if its tree representation is balanced and full [4].

Silvija et al [3] also describe a breadth-first search strategy. We proof their BFS is equivalent to ours. However, ours looks simpler and runs about twice faster in comparison with a straightforward translation of their Haskell code.

7 CONCLUSION

7.1 others

We devise a new search strategy, balanced interleaving DFS. The key idea is to make disjunct trees balanced. Changing the search strategy from iDFS to biDFS is not hard: 2 new functions and 1 modified macro.

We also devise breadth-first search, whose intuition is similar to Seres's BFS. And we have proved their equivalence. We optimize our BFS with a queue.

ACKNOWLEDGMENTS

REFERENCES

[1] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

[2] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition.*

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

[4] Edward Z Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue 15* (2010), 11.

```
(define (empty-inf) '(() . ()))
(define (unit v) `((,v) . ()))
(define (step f) `(() . (,f)))

(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`