

c1

# Towards a miniKanren with fair search strategies

KUANG-CHEN LU, Indiana University

WEIXI MA, Indiana University

DANIEL P. FRIEDMAN, Indiana University

c2

c3

c4

We describe fairness levels in disjunction and conjunction implementations. Specifically, a disjunction implementation can be fair, almost-fair, or unfair. And a conjunction implementation can be fair or unfair. We compare the fairness level of four search strategies: the standard miniKanren interleaving depth-first search, the balanced interleaving depth-first search, the fair depth-first search, and the standard breadth-first search. The two non-standard depth-first searches are new. And we present two new, more efficient and shorter implementations of the standard breadth-first search. Using quantitative evaluation, we argue that the two new depth-first searches are competitive alternatives to the standard one, and that our breadth-first search implementations are more efficient than the current one.

## ACM Reference format:

Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. Towards a miniKanren with fair search strategies. 1, 1, Article 1 (May 2019), 15 pages.

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

miniKanren is a family of relational programming languages. Friedman et al. [2, 3] introduce miniKanren and its implementation in *The Reasoned Schemer* and *The Reasoned Schemer, 2nd Ed* (TRS2). Although originally written in Scheme, miniKanren has been ported to many other languages (e.g. OCanren[5]). Byrd et al. [1] have

<sup>c1</sup>LKC: submission deadline: Mon 27 May 2019

<sup>c2</sup>DPF: Throughout the paper we have search spaces and search strategies. I can see leaving in the “search strategies”, but let’s try to remove virtually all of the uses of “search” in front of spaces.

<sup>c3</sup>LKC: BUT, I am thinking about going further — removing “our improved” from this paper. The first reason is that our BFS is actually more efficient than our improved BFS in most benchmarks (see Table 1), despite its theoretical space extravagance. The second reason is that DFSi, DFSf, and our BFS look nice when sitting together.

<sup>c4</sup>DPF: Does this mean that the comment about append goes away. I really thought that was so clever. I thought that the way you pointed out the trivial changes was very clear, so that did not bother me at all.

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 made or distributed for not-for-profit or commercial use, 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, or post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2019 Association for Computing Machinery.

XXXX-XXXX/2019/5-ART1 \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

demonstrated that miniKanren programs are useful in solving several difficult problems. miniKanren.org contains the seeds of many difficult problems and their solutions.

A subtlety arises when a `conde` contains many clauses: not every clause has an equal chance of contributing to the result. As an example, consider the following relation `repeato` and its invocation.

```
(defrel (repeato x out)
  (conde
    ((≡ `(,x) out))
    ((fresh (res)
      (≡ `(,x . ,res) out)
      (repeato x res))))))
> (run 4 q
  (repeato 'x q))
'((*) (* *) (* * *) (* * * *))
```

Next, consider the following disjunction of invoking `repeato` with four different letters.

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

`conde` intuitively relates its clauses with logical or. And thus an unsuspecting beginner would expect each letter to contribute equally to the result, as follows.

```
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

The `conde` in TRS2, however, generates a less expected result.

```
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

The miniKanren in TRS2 implements interleaving DFS (DFS<sub>i</sub>), the cause of this unexpected result. With this search strategy, each `conde` clause takes half of its received computational resources and passes the other half to its following clauses, except for the last clause that takes all resources it receives. In the example above, the a clause takes half of all resources. And the b clause takes a quarter. Thus c and d barely contribute to the result.

DFS<sub>i</sub> is sometimes powerful for an expert. By carefully organizing the order of `conde` clauses, a miniKanren program can explore more “interesting” clauses than those uninteresting ones, and thus use computational resources efficiently.

<sup>c5</sup> DPF: their BFS (or shortened to BFS). We need only say this once if we go with “our”, which I don’t like. I like “the new” or “the improved” and “the current”

DFS<sub>i</sub> is not always the best choice. For instance, it might be less desirable for novice miniKanren user—understanding implementation details and fiddling with clause order is not their first priority. There is another reason that miniKanren could use more search strategies than just DFS<sub>i</sub>. In many applications, there does not exist one order that serves all purposes. For example, a relational dependent type checker contains clauses for constructors that build data and clauses for eliminators that use data. When the type checker is generating simple and shallow programs, the clauses for constructors had better be at the top of the  $\text{cond}^e$  expression. When performing proof searches for complicated programs, the clauses for eliminators had better be at the top of the  $\text{cond}^e$  expression. With DFS<sub>i</sub>, these two uses cannot be efficient at the same time. In fact, to make one use efficient, the other one must be more sluggish.

The specification that gives every clause in the same  $\text{cond}^e$  equal “search priority” is fair disj. And search strategies with almost-fair disj give every clause similar priority. Fair conj, a related concept, is more subtle. We cover it in the next section.

To summarize our contributions, we

- propose and implement **balanced interleaving depth-first search** (DFS<sub>bi</sub>), a new search strategy with almost-fair disj.
- propose and implement **fair depth-first search** (DFS<sub>f</sub>), a new search strategy with fair disj.
- implement in two new ways the standard **breadth-first search** (BFS), a search strategy with fair disj and fair conj. We refer to “the current BFS implementation” (“BFS” for short) as the implementation by Seres et al. [7] and refer to “our BFS implementation” (“our BFS” for short) and “our improved BFS implementation” (“our improved BFS” for short) as our new ones. We also refer to “our BFSs” to both our BFS and our improved BFS. We argue that our BFSs are semantically equivalent and formally prove that our improved BFS and BFS are semantically equivalent, however, our BFSs run faster in all benchmarks and are shorter.

c1

## 2 SEARCH STRATEGIES AND FAIRNESS

In this section, we define fair disj, almost-fair disj and fair conj. Before going further into fairness, we give a short review of the terms: *state*, *search space*, and *goal*. A *state* is a collection of constraints. (Here, we restrict constraints to unification constraints.) Every answer corresponds to a state. A *space* is a collection of states. And a *goal* is a function from a state to a space.

Now we elaborate fairness by running more queries about  $\text{repeat}^o$ . We never use  $\text{run}^*$  in this paper because fairness is more interesting when we have an unbounded number of answers. It is perfectly fine, however, to use  $\text{run}^*$  with any search strategies.

### 2.1 fair disj

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the query result. DFS<sub>i</sub>, TRS2’s search strategy, however, results in many more lists of *a* than lists of other letters. And some letters (e.g. *c* and *d*) are rarely seen. The more clauses the worse the situation.

<sup>c1</sup>DPF: The citeteres1999algebra must be not broken: You can either rewrite the sentence or use a tilde called a tie that means don’t break the line. Sometimes using tilde does not work out elegant enough, so the first idea works best.

```

142 ;; DFSi (unfair disj)
143 > (run 12 q
144   (conde
145     ((repeato 'a q))
146     ((repeato 'b q))
147     ((repeato 'c q))
148     ((repeato 'd q))))
149 '((a) (a a) (b) (a a a)
150   (a a a a) (b b)
151   (a a a a a) (c)
152   (a a a a a a) (b b b)
153   (a a a a a a a) (d))

```

Under the hood, the  $\text{cond}^e$  here is allocating computational resources to four trivially different search spaces. The unfair  $\text{disj}$  in  $\text{DFS}_i$  allocates many more resources to the first search space. On the contrary, fair  $\text{disj}$  would allocate resources evenly to each search space.

<pre> 159 ;; DFS<sub>f</sub> (fair disj) 160 &gt; (run 12 q 161   (cond<sup>e</sup> 162     ((repeat<sup>o</sup> 'a q)) 163     ((repeat<sup>o</sup> 'b q)) 164     ((repeat<sup>o</sup> 'c q)) 165     ((repeat<sup>o</sup> 'd q)))) 166 '((a) (b) (c) (d) 167   (a a) (b b) (c c) (d d) 168   (a a a) (b b b) (c c c) (d d d)) </pre>	<pre> 159 ;; BFS (fair disj) 160 &gt; (run 12 q 161   (cond<sup>e</sup> 162     ((repeat<sup>o</sup> 'a q)) 163     ((repeat<sup>o</sup> 'b q)) 164     ((repeat<sup>o</sup> 'c q)) 165     ((repeat<sup>o</sup> 'd q)))) 166 '((a) (b) (c) (d) 167   (a a) (b b) (c c) (d d) 168   (a a a) (b b b) (c c c) (d d d)) </pre>
---	---

Running the same program again with almost-fair  $\text{disj}$  (e.g.  $\text{DFS}_{bi}$ ) gives the same result. Almost-fair, however, is not completely fair, as shown by the following example.

```

172 ;; DFSbi (almost-fair disj)
173 > (run 16 q
174   (conde
175     ((repeato 'a q))
176     ((repeato 'b q))
177     ((repeato 'c q))
178     ((repeato 'd q))
179     ((repeato 'e q))))
180 '((b) (c) (d) (a)
181   (b b) (c c) (d d) (e)
182   (b b b) (c c c) (d d d) (a a)
183   (b b b b) (c c c c) (d d d d) (e e))

```

$\text{DFS}_{bi}$  is fair only when the number of goals is a power of 2, otherwise, it allocates some goals twice as many resources as the others. In the above example, where the  $\text{cond}^e$  has five clauses,  $\text{DFS}_{bi}$  allocates more resources to the clauses of b, c, and d.

We end this subsection with precise definitions of all levels of `disj` fairness. Our definition of *fair disj* is slightly more general than the one in Seres et al. [7], which is only for binary disjunction. We generalize it to a multi-arity one.

**DEFINITION 2.1 (FAIR `disj`).** *A `disj` is fair if and only if it allocates computational resources evenly to search spaces produced by goals in the same disjunction (i.e., clauses in the same `conde`).*

**DEFINITION 2.2 (ALMOST-FAIR `disj`).** *A `disj` is almost-fair if and only if it allocates computational resources so evenly to search spaces produced by goals in the same disjunction that the maximal ratio of resources is bounded by a constant.*

**DEFINITION 2.3 (UNFAIR `disj`).** *A `disj` is unfair if and only if it is not almost-fair.*

## 2.2 fair `conj`

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer list. Search strategies with unfair `conj`: `DFSi`, `DFSbi`, and `DFSf`, however, results in many more lists of `a` than lists of other letters. And some letters are rarely seen. Here again, as the number of clauses grows, the situation worsens.

Although some strategies have a different level of fairness in `disj`, they have the same behavior when there is no call to a relational definition in `conde` clauses (including this case).

<pre>;; DFS<sub>i</sub> (unfair conj) &gt; (run 12 q     (fresh (x)       (cond<sup>e</sup>         ((= 'a x))         ((= 'b x))         ((= 'c x))         ((= 'd x)))       (repeat<sup>o</sup> x q))) '((a) (a a) (b) (a a a)   (a a a a) (b b)   (a a a a a) (c)   (a a a a a a) (b b b)   (a a a a a a a) (d))</pre>	<pre>;; DFS<sub>f</sub> (unfair conj) &gt; (run 12 q     (fresh (x)       (cond<sup>e</sup>         ((= 'a x))         ((= 'b x))         ((= 'c x))         ((= 'd x)))       (repeat<sup>o</sup> x q))) '((a) (a a) (b) (a a a)   (a a a a) (b b)   (a a a a a) (c)   (a a a a a a) (b b b)   (a a a a a a a) (d))</pre>	<pre>;; DFS<sub>bi</sub> (unfair conj) &gt; (run 12 q     (fresh (x)       (cond<sup>e</sup>         ((= 'a x))         ((= 'b x))         ((= 'c x))         ((= 'd x)))       (repeat<sup>o</sup> x q))) '((a) (a a) (c) (a a a)   (a a a a) (c c)   (a a a a a) (b)   (a a a a a a) (c c c)   (a a a a a a a) (d))</pre>
--	--	---

Under the hood, the `conde` and the call to `repeato` are connected by `conj`. The `conde` goal outputs a search space including four trivially different states. Applying the next conjunctive goal, `(repeato x q)`, produces four trivially different search spaces. In the examples above, all search strategies allocate more computational resources to the search space of `a`. On the contrary, fair `conj` would allocate resources evenly to each search space. For example,

```

236 ;; BFS (fair conj)
237 > (run 12 q
238   (fresh (x)
239     (conde
240       ((= 'a x))
241       ((= 'b x))
242       ((= 'c x))
243       ((= 'd x)))
244     (repeato x q)))
245 '((a) (b) (c) (d)
246   (a a) (b b) (c c) (d d)
247   (a a a) (b b b) (c c c) (d d d))

```

A more interesting situation is when the first conjunct produces an unbounded number of states. Consider the following example, a naive specification of fair conj might require search strategies to produce all sorts of singleton lists, but there would not be any lists of length two or longer, which makes the strategies incomplete. A search strategy is *complete* if and only if “every correct answer would be discovered after some finite time” [7], otherwise, it is *incomplete*. In the context of miniKanren, a search strategy is complete means that every correct answer has a position in large enough answer lists.

```

256 ;; naively fair conj
257 > (run 6 q
258   (fresh (xs)
259     (conde
260       ((repeato 'a xs))
261       ((repeato 'b xs))
262       (repeato xs q)))
263   '(((a)) ((b))
264     ((a a)) ((b b))
265     ((a a a)) ((b b b)))

```

Our solution requires a search strategy with *fair conj* to organize states in bags in search spaces, where each bag contains finite states, and to allocate resources evenly among search spaces derived from states in the same bag. It is up to a search strategy designer to decide by what criteria to put states in the same bag, and how to allocate resources among search spaces related to different bags.

BFS puts states of the same cost in the same bag, and allocates resources carefully among search spaces related to different bags such that it produces answers in increasing order of cost. The *cost* of an answer is its depth in the search tree (i.e., the number of calls to relations required to find them) [7]. In the following example, every answer is a list of a list of symbols, where inner lists in the same outer list are identical. Here the cost of each answer is equal to the length of its inner list plus the length of its outer list. For example, the cost of `((a) (a))` is  $1 + 2 = 3$ .

c1 c2

<sup>c1</sup>DPF: Can you express exactly how all costs are determined? I see that for this simple program, your notion of cost makes sense, but how does it generalize to all costs.

<sup>c2</sup>LKC: I mean for all answer in the answer list, the answer's inner lists are pairwise equal?. It is impossible to see a `((a) (b))` in the answer list.

```

283 ;; BFS (fair conj)
284 > (run 12 q
285   (fresh (xs)
286    (conde
287     ((repeato 'a xs))
288     ((repeato 'b xs)))
289    (repeato xs q)))
290 '(((a)) ((b)))
291 ((a) (a)) ((b) (b))
292 ((a a)) ((b b))
293 ((a) (a) (a)) ((b) (b) (b))
294 ((a a) (a a)) ((b b) (b b))
295 ((a a a)) ((b b b)))

```

We end this subsection with precise definitions of all levels of conj fairness.

**DEFINITION 2.4 (FAIR conj).** A conj is fair if and only if it allocates computational resources evenly to search spaces produced from states in the same bag. A bag is a finite collection of states. And search strategies with fair conj should represent search spaces with possibly unbounded collections of bags.

**DEFINITION 2.5 (UNFAIR conj).** A conj is unfair if and only if it is not fair.

### 3 INTERLEAVING DEPTH-FIRST SEARCH

In this section, we review the implementation of interleaving depth-first search ( $\text{DFS}_i$ ). We focus on parts that are relevant to this paper. TRS2, chapter 10 and the appendix, “Connecting the wires”, provide a comprehensive description of the miniKanren implementation but limited to unification constraints ( $\equiv$ ). Fig. 1 and Fig. 2 show parts that are later compared with other search strategies. We follow some conventions to name variables: ss name states; gs (possibly with subscript) name goals; variables ending with  $^\infty$  name (search) spaces. Fig. 1 shows the implementation of disj. The first function,  $\text{disj}_2$ , implements binary disjunction. It applies the two disjunctive goals to the input state  $s$  and composes the two resulting search spaces with  $\text{append}^\infty$ . The following syntax definitions say disj is right-associative. Fig. 2 shows the implementation of conj. The first function,  $\text{conj}_2$ , implements binary conjunction. It applies the *first* goal to the input state, then applies the second goal to states in the resulting search space. The helper function  $\text{append-map}^\infty$  applies its input goal to states in its input search spaces and composes the resulting search spaces. It reuses  $\text{append}^\infty$  for search space composition. The following syntax definitions say conj is also right-associative.

### 4 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

Balanced interleaving DFS ( $\text{DFS}_{bi}$ ) has an almost-fair disj and unfair conj. The implementation of  $\text{DFS}_{bi}$  differs from  $\text{DFS}_i$ ’s in the disj macro. We list the new disj with its helper in Fig. 3. When there are one or more disjunctive goals, disj builds a balanced binary tree whose leaves are the goals and whose nodes are  $\text{disj}_2$ s, hence the name of this search strategy. The new helper,  $\text{disj}^+$ , takes two additional ‘arguments’. They accumulate goals to be put in the left and right subtrees. The first clause handles the case where there is only one goal. In this case, the tree is the goal itself. When there are more goals, we partition the list of goals into two sublists of roughly equal lengths and recur on the two sublists. We move goals to the accumulators in the last clause. As we are moving two goals each time, there are two base cases: (1) no goal remains; (2) one goal remains. We handle these two new base cases in the second clause and the third clause, respectively. In contrast, the disj in  $\text{DFS}_i$  constructs the binary tree in a particularly unbalanced form.

```

330 #| Goal × Goal → Goal |#
331 (define (disj2 g1 g2)
332   (lambda (s)
333     (append∞ (g1 s) (g2 s))))
334
335 #| Space × Space → Space |#
336 (define (append∞ s∞ t∞)
337   (cond
338     ((null? s∞) t∞)
339     ((pair? s∞)
340      (cons (car s∞)
341            (append∞ (cdr s∞) t∞)))
342     (else (lambda ()
343              (append∞ t∞ (s∞))))))
344
345
346 (define-syntax disj
347   (syntax-rules ()
348     ((disj) (fail))
349     ((disj g0 g ...) (disj+ g0 g ...))))
350
351
352 (define-syntax disj+
353   (syntax-rules ()
354     ((disj+ g) g)
355     ((disj+ g0 g1 g ...) (disj2 g0 (disj+ g1 g ...))))
356
357

```

Fig. 1. implementation of DFS<sub>i</sub> (Part I)

## 5 FAIR DEPTH-FIRST SEARCH

Fair DFS (DFS<sub>f</sub>) has fair disj and unfair conj. The implementation of DFS<sub>f</sub> differs from DFS<sub>i</sub>'s in disj<sub>2</sub> (Fig. 4). The new disj<sub>2</sub> calls a new and fair version of append<sup>∞</sup>. append<sup>∞</sup><sub>fair</sub> immediately calls its helper, loop, with the first argument, s?, set to #t, which indicates that we haven't swapped s<sup>∞</sup> and t<sup>∞</sup>. The swapping happens at the third cond clause in the helper, where s? is updated accordingly. The first two cond clauses essentially copy the cars and stop recursion when one of the input spaces is obviously finite. The third clause, as we mentioned above, is just for swapping. When the fourth and last clause runs, we know that both s<sup>∞</sup> and t<sup>∞</sup> are ended with a thunk, and that we have swapped them. In this case, we construct a new thunk. The new thunk swaps back the two spaces in the recursive call to loop. This is unnecessary for fairness—we do it to produce answers in a more readable order.



```

377 #| Goal × Goal → Goal |#
378 (define (conj2 g1 g2)
379   (lambda (s)
380     (append-map∞ g2 (g1 s))))
381
382 #| Goal × Space → Space |#
383 (define (append-map∞ g s∞)
384   (cond
385     ((null? s∞) '())
386     ((pair? s∞)
387      (append∞ (g (car s∞))
388                (append-map∞ g (cdr s∞))))
389     (else (lambda ()
390              (append-map∞ g (s∞))))))
391
392
393 (define-syntax conj
394   (syntax-rules ()
395     ((conj) (fail))
396     ((conj g0 g ...) (conj+ g0 g ...))))
397
398
399 (define-syntax conj+
400   (syntax-rules ()
401     ((conj+ g) g)
402     ((conj+ g0 g1 g ...) (conj2 g0 (conj+ g1 g ...))))
403
404

```

Fig. 2. implementation of DFS<sub>i</sub> (Part II)

## 6 BREADTH-FIRST SEARCH

BFS has both fair disj and fair conj. Our first implementation (Figure 5) is based on DFS<sub>f</sub> (not DFS<sub>i</sub>). It is so similar to DFS<sub>f</sub> that we only need to apply two changes: (1) rename append-map<sup>∞</sup> to append-map<sup>∞</sup><sub>fair</sub> and (2) replace append<sup>∞</sup> with append<sup>∞</sup><sub>fair</sub> in append-map<sup>∞</sup>'s body.

c1 c2  
c3  
c4

<sup>c1</sup>DPF: I need clarification: Applying the above changes won't result in "our BFS," but yet a third implementation of BFS?

<sup>c2</sup>LKC: Improved.

<sup>c3</sup>DPF: Why are the two texttt's macros, again? Now, it hardly matters, but it is unfriendly to your other co-authors to have us reading long expressions when a newcommand would suffice. I read the paper with a pen marking it up, then I read the paper on the .tex screen. That's each time I read it.

<sup>c4</sup>DPF: You know that "bag" is a technical terms. It means that there is a structure where each element of the structure has an associated number, which tells you have many occurrences of that element are in the list. For example, the list (a b a c a d c) could be represented like this: (a<sup>3</sup> b<sup>1</sup> c<sup>2</sup> d<sup>1</sup>). If that is not what you mean, you might change the name to bucket.

```

424 (define-syntax disj
425   (syntax-rules ()
426     ((disj) fail)
427     ((disj g ...) (disj+ (g ...) () ())))))
428
429 (define-syntax disj+
430   (syntax-rules ()
431     ((disj+ () () g) g)
432     ((disj+ (gl ...) (gr ...))
433      (disj2 (disj+ () () gl ...)
434              (disj+ () () gr ...)))
435     ((disj+ (gl ...) (gr ...) g0)
436      (disj2 (disj+ () () gl ... g0)
437              (disj+ () () gr ...)))
438     ((disj+ (gl ...) (gr ...) g0 g1 g ...)
439      (disj+ (gl ... g0) (gr ... g1) g ...)))
440
441
442
443
444

```

Fig. 3. implementation of  $\text{DFS}_{bi}$ 

```

445 #| Goal  $\times$  Goal  $\rightarrow$  Goal |#
446 (define (disj2 g1 g2)
447   (lambda (s)
448     (append∞fair (g1 s) (g2 s))))
449
450 #| Space  $\times$  Space  $\rightarrow$  Space |#
451 (define (append∞fair s∞ t∞)
452   (let loop ((s? #t) (s∞ s∞) (t∞ t∞))
453     (cond
454       ((null? s∞) t∞)
455       ((pair? s∞)
456        (cons (car s∞)
457              (loop s? (cdr s∞) t∞)))
458       (s? (loop #f t∞ s∞))
459       (else (lambda ()
460                (loop #t (t∞) (s∞)))))))
461
462
463
464
465

```

Fig. 4. implementation of  $\text{DFS}_f$ 

This implementation can be improved in two ways. First, as mentioned in subsection 2.2, BFS puts answers in bags and answers of the same cost are in the same bag. In the above implementation, however, it is not obvious how we manage cost information—the cars of a search space have cost 0 (i.e., they are all in the same bag), and every thunk indicates an increment in cost. It is even more subtle that  $\text{append}^{\infty}_{\text{fair}}$  and the  $\text{append-map}^{\infty}_{\text{fair}}$

```

471 #| Goal × Space → Space |#
472 (define (append-map∞fair g s∞)
473   (cond
474     ((null? s∞) '())
475     ((pair? s∞)
476      (append∞fair (g (car s∞))
477                    (append-map∞fair g (cdr s∞))))
478     (else (lambda ()
479              (append-map∞fair g (s∞)))))))
480
481
482

```

Fig. 5. implementation of our BFS (based on DFS<sub>f</sub>)

respects the cost information. Second,  $\text{append}^{\infty}_{\text{fair}}$  is extravagant in memory usage. It makes  $O(n + m)$  new cons cells every time, where  $n$  and  $m$  are the “length”s of input spaces.  $\text{DFS}_f$  also has the space extravagance.

In the following subsections, we first describe our improved BFS implementation that manages cost information in a more clear and concise way and is less extravagant in memory usage. Then we compare our improved BFS with BFS.

## 6.1 improved BFS

We make the cost information more clear by changing the Space type, modifying related function definitions, and introducing a few more functions.

The new type is a pair whose car is a list of answers (the bag), and whose cdr is either #f or a thunk returning a search space. A falsy cdr means the search space is obviously finite.

We list functions related to the pure subset in Fig. 6. The first three functions are search space constructors. none makes an empty search space; unit makes a space from one answer; and step makes a space from a thunk. The remaining functions are as before. We compare these functions with BFS in our proof.

Luckily, the change in  $\text{append}^{\infty}_{\text{fair}}$  also fixes the miserable space extravagance—the use of append helps us to reuse the first bag of  $t^{\infty}$ .

Kiselyov et al. [4] have demonstrated that a *MonadPlus* hides in implementations of logic programming system. our improved BFS is not an exception:  $\text{append-map}^{\infty}_{\text{fair}}$  is like bind, but takes arguments in reversed order; none, unit, and  $\text{append}^{\infty}_{\text{fair}}$  correspond to mzero, unit, and mplus, respectively.

Functions implementing impure features are in Fig. 7. The first function, elim, takes a space  $s^{\infty}$  and two continuations kf and ks. When  $s^{\infty}$  contains no answers, it calls kf with no argument. Otherwise, it calls ks with the first answer and the rest of the space. Here ‘f’ means ‘fail’ and ‘s’ means ‘succeed’. This function is similar to an eliminator of search spaces, hence the name. The remaining functions are as before.

<sup>c1</sup>DPF: Let’s use one more line and far less horizontal space by use 3 linefeeds. They will still align nicely.

<sup>c2</sup>LKC: fixed

<sup>c3</sup>DPF: Font Space like in the code comments.

<sup>c4</sup>DPF: ks → sk and kf → fk and you don’t need to say f and s mean success continuation and failure continuation.

```

518 #|  $\rightarrow \text{Space}$  |#
519 (define (none)
520   `(() . #f))
521
522 #|  $\text{State} \rightarrow \text{Space}$  |#
523 (define (unit s)
524   `((,s) . #f))
525
526 #|  $(\rightarrow \text{Space}) \rightarrow \text{Space}$  |#
527 (define (step f)
528   `(() . ,f))
529
530
531 #|  $\text{Space} \times \text{Space} \rightarrow \text{Space}$  |#
532 (define (append∞fair s∞ t∞)
533   (cons (append (car s∞) (car t∞))
534         (let ((t1 (cdr s∞)) (t2 (cdr t∞)))
535           (cond
536             ((not t1) t2)
537             ((not t2) t1)
538             (else (lambda ()
539                     (append∞fair (t1) (t2)))))))
540
541
542 #|  $\text{Goal} \times \text{Space} \rightarrow \text{Space}$  |#
543 (define (append-map∞fair g s∞)
544   (foldr
545     (lambda (s t∞)
546       (append∞fair (g s) t∞))
547     (let ((f (cdr s∞)))
548       (step (and f (lambda () (append-map∞fair g (f))))))
549     (car s∞))
550
551
552 #|  $\text{Maybe Nat} \times \text{Space} \rightarrow [\text{State}]$  |#
553 (define (take∞ n s∞)
554   (let loop ((n n) (vs (car s∞)))
555     (cond
556       ((and n (zero? n)) '())
557       ((pair? vs)
558        (cons (car vs)
559              (loop (and n (sub1 n)) (cdr vs))))
560       (else (let ((f (cdr s∞)))
561               (if f (take∞ n (f)) '()))))))
562
563
564

```

Fig. 6. New and changed functions in our BFS that implements pure features

```

565 #| Space × (State × Space → Space) × (→ Space) → Space |#
566 (define (elim s∞ kf ks)
567   (let ((ss (car s∞)) (f (cdr s∞)))
568     (cond
569       ((pair? ss) (ks (car ss) (cons (cdr ss) f)))
570       (f (step (lambda () (elim (f) kf ks))))
571       (else (kf)))))
572
573
574 #| Goal × Goal × Goal → Goal |#
575 (define (ifte g1 g2 g3)
576   (lambda (s)
577     (elim (g1 s)
578           (lambda () (g3 s))
579           (lambda (s0 s∞)
580             (append-map∞fair g2
581               (append∞fair (unit s0) s∞))))))
582
583
584 #| Goal → Goal |#
585 (define (once g)
586   (lambda (s)
587     (elim (g s)
588           (lambda () (none))
589           (lambda (s0 s∞) (unit s0))))))
590

```

Fig. 7. New and changed functions in our BFS that implement impure features

## 6.2 compare our improved BFS with BFS

In this subsection, we compare the pure subset of our improved BFS with BFS. We focus on the pure subset because BFS is designed for a pure relational programming system. We prove in Coq that these two search strategies are semantically equivalent, since  $(\text{run } n \ ? \ g)$  produces the same result in both our BFS and BFS. (See supplements for the formal proof.) To compare efficiency, we translate BFS's Haskell code into Racket (See supplements for the translated code). The translation is direct due to the similarity of the two relational programming systems. The translated code is longer than our improved BFS. And it runs slower in all benchmarks. Details about differences in efficiency are in section 7.

## 7 QUANTITATIVE EVALUATION

In this section, we compare the efficiency of search strategies. A concise description is in Table 1. A hyphen means “running out of 500 MB memory.” The first two benchmarks are from TRS2. *revers<sup>o</sup>* is from Rozplokhas and Boulytchev [6]. The next two benchmarks about quine are modified from a similar test case in Byrd et al. [1]. The modifications are made to circumvent the need for symbolic constraints (e.g.  $\neq$ ,  $\text{absent}^o$ ). Our version generates de Bruijnized expressions and prevents closures from being inside a list. The sibling benchmarks differ in the *cond<sup>e</sup>* clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to  $'(I \text{ love } you)$ . They are also modified from a similar test case in Byrd et al. [1] because of

benchmark	size	DFS <sub>i</sub>	DFS <sub>bi</sub>	DFS <sub>f</sub>	our BFS	our improved BFS	BFS
very-recursive <sup>o</sup>	100000	166	103	412	417	451	1024
	200000	283	146	765	766	839	1875
	300000	429	346	2085	2066	1809	4408
append <sup>o</sup>	100	17	18	17	18	17	65
	200	145	137	137	137	142	121
	300	388	384	387	383	429	371
revers <sup>o</sup>	10	3	4	3	18	18	36
	20	35	35	33	3119	3070	3695
	30	329	333	315	76243	75079	107531
quine-1	1	13	14	10	-	-	-
	2	30	34	25	-	-	-
	3	41	48	33	-	-	-
quine-2	1	22	21	12	-	-	-
	2	51	46	24	-	-	-
	3	72	76	32	-	-	-
'(I love you)-1	99	10	37	-	40	40	96
	198	22	71	-	374	374	652
	297	24	292	-	1707	1668	3759
'(I love you)-2	99	485	179	-	38	44	94
	198	808	638	-	366	414	630
	297	1265	916	-	1674	1651	3691

Table 1. The results of a quantitative evaluation: running times of benchmarks in milliseconds

the same reason. Our version generates de Bruijnized expressions. Again, the sibling benchmarks differ in the  $\text{cond}^e$  clause order of their relational interpreters. The first one has elimination rules (i.e. application,  $\text{car}$ , and  $\text{cdr}$ ) at the end, while the other has them at the beginning. We conjecture that  $\text{DFS}_i$  would perform badly in the second case because elimination rules complicate the problem when synthesizing (i.e., our evaluation supports our conjecture.)

c1

c2

In general, only  $\text{DFS}_i$  and  $\text{DFS}_{bi}$  constantly perform well.  $\text{DFS}_f$  is just as efficient in  $\text{append}^o$ ,  $\text{revers}^o$ , and quines. All BFS implementations have obvious overhead in most cases. Among the three variants of DFS, which all have unfair  $\text{conj}$ ,  $\text{DFS}_f$  is most resistant to clause permutation in quines, followed by  $\text{DFS}_{bi}$  then  $\text{DFS}_i$ .  $\text{DFS}_f$ , however, runs out of memory in '(I love you)s. In contrast,  $\text{DFS}_{bi}$  still performs more stable than  $\text{DFS}_i$ . Thus, we consider  $\text{DFS}_{bi}$  a competitive alternative to  $\text{DFS}_i$ , the current standard miniKanren search strategy. Among the three implementations of BFS, our BFSs constantly perform as well or better. Among our BFSs, our improved BFS only performs better in very-recursive<sup>o</sup> 300000. The reason might be that the size of bags are usually too small to be worried about. It is hard to conclude how  $\text{disj}$  fairness affect performance based on these statistics.

<sup>c1</sup>LKC: I have changed the benchmarks about '(I love you), such that their relational interpreters doesn't prevent closures from getting into pairs. I did so because I consider this restriction doesn't make sense. Benchmarks about quine are already very slow so I didn't apply similar changes.

<sup>c2</sup>DPF: We don't have disequality constraints, so no  $\text{absento}$ . You might just say that.

Fair `conj`, however, imposes considerable overhead constantly except in `append`<sup>o</sup>. The reason might be that those strategies tend to keep more intermediate answers in the memory.

## 8 RELATED WORKS

Yang [8] points out that a disjunct complex would be ‘fair’ if it were a full and balanced tree.

Seres et al. [7] describe a breadth-first search strategy. We present another implementation. Our implementation is semantically equivalent to theirs. But, ours is shorter and performs better in comparison with a straightforward translation of their Haskell code.

Rozplokhos and Boulytchev [6] address the non-commutativity of conjunction, while our work about `disj` fairness addresses the non-commutativity of disjunction.

## 9 CONCLUSION

We analyze the definitions of fair `disj` and fair `conj`, then propose a new definition of fair `conj`. Our definition is orthogonal with completeness.

We devise two new search strategies (i.e. balanced interleaving DFS ( $\text{DFS}_{bi}$ ) and fair DFS ( $\text{DFS}_f$ )) and devise two new implementations of BFS. These strategies have different features in fairness:  $\text{DFS}_{bi}$  has an almost-fair `disj` and unfair `conj`.  $\text{DFS}_f$  has fair `disj` and unfair `conj`. BFS has both fair `disj` and fair `conj`.

Our quantitative evaluation shows that  $\text{DFS}_{bi}$  is a competitive alternative to  $\text{DFS}_i$ , the current miniKanren search strategy, and that BFS is less practical.

Our improved BFS is semantically equivalent to the original one. But, ours is shorter and performs better in comparison with a straightforward translation of their Haskell code.

<sup>c1</sup> <sup>c2</sup>

## 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).
- [2] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. The MIT Press.
- [3] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*.
- [4] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.
- [5] Dmitry Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).
- [6] Dmitri Rozplokhos and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. ACM, 18.
- [7] Silvija Seres, J Michael Spivey, and C. A. R. Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.
- [8] Edward Z. Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue* 15 (2010), 11.

<sup>c1</sup>DPF: Finish this paragraph with a clarification and you may place this anywhere in the conclusion. This may require some consultation with Weixi.

<sup>c2</sup>LKC: I will do it later.