

c1

Towards a miniKanren with fair search strategies

KUANG-CHEN LU, Indiana University

WEIXI MA, Indiana University

DANIEL P. FRIEDMAN, Indiana University

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 a new, more efficient and simpler implementation 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 implementation is 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), 16 pages.
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

miniKanren is a family of relational programming languages. [2] Friedman et al. [3] introduce miniKanren and its implementation in *The Reasoned Schemer* and *The Reasoned Schemer, 2nd Ed* (TRS2). Byrd et al. [1] have

^{c1}LKC: submission deadline: Mon 27 May 2019

Unpublished working draft. Not for distribution

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 or distributed 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/2019/5-ART1 \$15.00

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

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

c2
c3 c4
c5
c6

A subtlety arises when a cond^e contains many clauses: not every clause has an equal chance of contributing to the result. As an example, consider the following relation repeat^o and its invocation.

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

Next, consider the following disjunction of invoking repeat^o with four different letters.

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

cond^e 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 cond^e in TRS2, however, generates a less expected result.

^{c2}DPF: [2,3] is the way to cite this, but I leave cleaning up this paragraph to you. Mostly the point is to also include a reference to the two dimitris and to the first OCamren (OCanren), whatever. It is up to you in this paragraph. You can refer to the two books like this “Friedman et al[2,3] ...”. This way the paragraph will have only one “Friedman”

^{c3}DPF: I have decided to agree with you that we should not use BFSs, so I am going to remove all occurrences. Also, we should say, “Our implementation of BFS” and then say, shortened to “Our BFS” and then you can use it in the table, so one column should be Our BFS and the other should be BFS.

^{c4}DPF: Occurrences of BFS (without Our) mean Seres et al. (We need only say that once, or very occasionally

^{c5}DPF: I think that subscripts i, bi, and f should match what is in the comments above code, so it should be italic, emph, or math, whichever was used before.

^{c6}DPF: Is the spelling of reverseo truly written without the last e? Shouldn't it be reverseo with the appropriate raising of o.

```
'((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_i), the cause of this unexpected result. With this search strategy, each *cond^e* 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_i is sometimes powerful for an expert. By carefully organizing the order of *cond^e* clauses, a miniKanren program can explore more “interesting” clauses than those uninteresting ones, and thus use computational resources efficiently.

DFS_i is not always the best choice. For instance, it might be less desirable for little miniKanreners – 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_i. 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 *cond^e* expression. When performing proof searches for complicated programs, the clauses for eliminators had better be at the top of the *cond^e* expression. With DFS_i, 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 *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_{bi}), a new search strategy with almost-fair *disj*.
- propose and implement **fair depth-first search** (DFS_f), a new search strategy with fair *disj*.
- implement in a new way the standard **breadth-first search** (BFS), a search strategy with fair *disj* and fair *conj*. We name the current implementation by Seres et al. and refer to “our BFS implementation as the improved one. We formally prove that the two BFS implementations are semantically equivalent, however, our BFS implementation runs faster in all benchmarks and is shorter.

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 *repeat^o*. We never use *run** in this paper because fairness is more interesting when we have an unbounded number of answers. However, it is perfectly fine to use *run** with any search strategies.

^{c1} LKC: The following paragraph is extended to include an explanation on *run**

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_i , the current 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 situation would be exacerbated if the cond^e would have contained more clauses.

```
;; DFSi (unfair disj)
> (run 12 q
  (conde
    ((repeato 'a q))
    ((repeato 'b q))
    ((repeato 'c q))
    ((repeato 'd 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))
```

Under the hood, the cond^e here is allocating computational resources to four trivially different search spaces. The unfair `disj` in DFS_i allocates many more resources to the first search space. On the contrary, fair `disj` would allocate resources evenly to each search space.

<pre>;; DFS_f (fair disj) > (run 12 q (cond^e ((repeat^o 'a q)) ((repeat^o 'b q)) ((repeat^o 'c q)) ((repeat^o 'd q)))) '((a) (b) (c) (d) (a a) (b b) (c c) (d d) (a a a) (b b b) (c c c) (d d d))</pre>	<pre>;; BFS (fair disj) > (run 12 q (cond^e ((repeat^o 'a q)) ((repeat^o 'b q)) ((repeat^o 'c q)) ((repeat^o 'd q)))) '((a) (b) (c) (d) (a a) (b b) (c c) (d d) (a a a) (b b b) (c c c) (d d d))</pre>
---	---

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

```

189 ;; DFSbi (almost-fair disj)
190 > (run 16 q
191   (conde
192     ((repeato 'a q))
193     ((repeato 'b q))
194     ((repeato 'c q))
195     ((repeato 'd q))
196     ((repeato 'e q))))
197 '((b) (c) (d) (a)
198   (b b) (c c) (d d) (e)
199   (b b b) (c c c) (d d d) (a a)
200   (b b b b) (c c c c) (d d d d) (e e))

```

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 cond^e has five clauses, 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. [6]. Their definition 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 cond^e).

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 (e.g. DFS_i, DFS_{bi}, DFS_f), however, results in many more lists of a than lists of other letters. And some letters are rarely seen. The situation would be exacerbated if cond^e were to contain more clauses. 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 cond^e clauses (including this case).

<pre> 236 ;; DFS_i (unfair conj) 237 > (run 12 q 238 (fresh (x) 239 (cond^e 240 ((≡ 'a x)) 241 ((≡ 'b x)) 242 ((≡ 'c x)) 243 ((≡ 'd x))) 244 (repeat^o x q))) 245 '((a) (a a) (b) (a a a) 246 (a a a a) (b b) 247 (a a a a a) (c) 248 (a a a a a a) (b b b) 249 (a a a a a a a) (d))) </pre>	<pre> 236 ;; DFS_f (unfair conj) 237 > (run 12 q 238 (fresh (x) 239 (cond^e 240 ((≡ 'a x)) 241 ((≡ 'b x)) 242 ((≡ 'c x)) 243 ((≡ 'd x))) 244 (repeat^o x q))) 245 '((a) (a a) (b) (a a a) 246 (a a a a) (b b) 247 (a a a a a) (c) 248 (a a a a a a) (b b b) 249 (a a a a a a a) (d))) </pre>	<pre> 236 ;; DFS_{bi} (unfair conj) 237 > (run 12 q 238 (fresh (x) 239 (cond^e 240 ((≡ 'a x)) 241 ((≡ 'b x)) 242 ((≡ 'c x)) 243 ((≡ 'd x))) 244 (repeat^o x q))) 245 '((a) (a a) (c) (a a a) 246 (a a a a) (c c) 247 (a a a a a) (b) 248 (a a a a a a) (c c c) 249 (a a a a a a a) (d))) </pre>
---	---	--

Under the hood, the cond^e and the call to repeat^o are connected by conj . The cond^e goal outputs a search space including four trivially different states. Applying the next conjunctive goal, $(\text{repeat}^o x q)$, 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,

```

257 ;; BFS (fair conj)
258 > (run 12 q
259   (fresh (x)
260     (conde
261       ((≡ 'a x))
262       ((≡ 'b x))
263       ((≡ 'c x))
264       ((≡ 'd x)))
265     (repeato x q)))
266 '((a) (b) (c) (d)
267   (a a) (b b) (c c) (d d)
268   (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” [6], 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.

^{c1}

^{c1}DPF: Was this supposed to be a note? “We never use run^* in this paper because fairness is only interesting when we have an unbounded number of answers. However, it is perfectly fine to use run^* with any search strategies.”

```

283 ;; naively fair conj
284 > (run 6 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 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) [6]. 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 $1 + 2 = 3$.

```

304 c1 c2
305
306 ;; BFS (fair conj)
307 > (run 12 q
308   (fresh (xs)
309     (conde
310       ((repeato 'a xs))
311       ((repeato 'b xs)))
312     (repeato xs q)))
313 '(((a)) ((b))
314   ((a) (a)) ((b) (b))
315   ((a a)) ((b b))
316   ((a) (a) (a)) ((b) (b) (b))
317   ((a a) (a a)) ((b b) (b b))
318   ((a a a)) ((b b b)))
319

```

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.

^{c1}DPF: Would the answer be the same if the list had been this $((a) (b))$ or did you mean structurally the same?

^{c2}LKC: I have rewritten the relevant sentences.

```

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_i (Part I)

3 INTERLEAVING DEPTH-FIRST SEARCH

In this section, we review the implementation of interleaving depth-first search (DFS_i). We focus on parts that are relevant to this paper. TRS2, chapter 10 and the appendix, “Connecting the wires”, provides 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 [∞] name search spaces. Fig. 1 shows the implementation of *disj*. The first function, *disj₂*, implements binary disjunction. It applies the two disjunctive goals to the input state *s* and composes the two resulting search spaces with *append[∞]*. The following syntax definitions say *disj* is right-associative. Fig. 2 shows the implementation of *conj*. The first function, *conj₂*, 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 *append-map[∞]* applies its input goal to states in its input search


```

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_i (Part II)

spaces and composes the resulting search spaces. It reuses `append∞` for search space composition. The following syntax definitions say `conj` is also right-associative.

^{c1} ^{c2}

4 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

Balanced interleaving DFS (DFS_{bi}) has an almost-fair `disj` and unfair `conj`. The implementation of DFS_{bi} differs from 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 `disj2s`, hence the name of this search strategy. The new helper, `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

^{c1}DPF: sure, go ahead and do it.

^{c2}LKC: I think we can simply remove `condc` code and its accompany description.

```

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
445

```

Fig. 3. implementation of DFS_{bi}

```

446 #| Goal × Goal → Goal |#
447 (define (disj2 g1 g2)
448   (lambda (s)
449     (append∞fair (g1 s) (g2 s))))
450
451 #| Space × Space → Space |#
452 (define (append∞fair s∞ t∞)
453   (let loop ((s? #t) (s∞ s∞) (t∞ t∞))
454     (cond
455       ((null? s∞) t∞)
456       ((pair? s∞)
457        (cons (car s∞)
458              (loop s? (cdr s∞) t∞)))
459       (s? (loop #f t∞ s∞))
460       (else (lambda ()
461                (loop #t (t∞) (s∞)))))))
462
463
464
465
466
467

```

Fig. 4. implementation of DFS_f

these two new base cases in the second clause and the third clause, respectively. In contrast, the `disj` in DFS_i constructs the binary tree in a particularly unbalanced form.

5 FAIR DEPTH-FIRST SEARCH

Fair DFS (DFS_f) has fair disj and unfair conj . The implementation of DFS_f differs from DFS_i 's in disj_2 (Fig. 4). The new disj_2 calls a new and fair version of append^∞ . $\text{append}_{fair}^\infty$ immediately calls its helper, loop , with the first argument, $s?$, set to $\#t$, which indicates that we haven't swapped s^∞ and t^∞ . 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^∞ and t^∞ are ended with a thunk, and that we have swapped them. In this case, we construct a new thunk. The new thunk swaps two spaces back in the recursive call to loop . This is unnecessary for fairness. We do it to produce answers in a more natural order.

^{c1} ^{c2}

6 BREADTH-FIRST SEARCH

BFS has both fair disj and fair conj . An easy implementation is based on DFS_f (not DFS_i): (1) replace append^∞ with $\text{append}_{fair}^\infty$ in append-map^∞ 's body (2) rename append-map^∞ to $\text{append-map}_{fair}^\infty$.

^{c3}

This implementation is improvable 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 our implementation, however, we manage cost information subtly—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 $\text{append}_{fair}^\infty$ and the $\text{append-map}_{fair}^\infty$ respects the cost information. Second, $\text{append}_{fair}^\infty$ 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.

^{c4} ^{c5}

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 BFS with BFS, the current implementation by Seres et al..

^{c6}

6.1 improved BFS

We make the cost information more clear by changing the type of search space, 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. 5. 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.

^{c1}DPF: When or where do you swap them back? This could be still clearer.

^{c2}LKC: Improved.

^{c3}DPF: I need clarification: Applying the above changes won't result in “our BFS,” but yet a third implementation of BFS?

^{c4}LKC: Should I point out that DFS_f also has the miserable space extravagance?

^{c5}DPF: probably

^{c6}DPF: Let's use one more line and far less horizontal space by use 3 linefeeds. They will still align nicely.

```

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

```

Fig. 5. New and changed functions in optimized 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       ((and (null? ss) f)
570        (step (lambda () (elim (f) kf ks))))
571       ((null? ss) (kf))
572       (else (ks (car ss) (cons (cdr ss) f))))))
573
574
575 #| Goal × Goal × Goal → Goal |#
576 (define (ifte g1 g2 g3)
577   (lambda (s)
578     (elim (g1 s)
579           (lambda () (g3 s))
580           (lambda (s0 s∞)
581             (append-map∞fair g2
582              (append∞fair (unit s0) s∞))))))
583
584
585 #| Goal → Goal |#
586 (define (once g)
587   (lambda (s)
588     (elim (g s)
589           (lambda () (none))
590           (lambda (s0 s∞) (unit s0))))))
591

```

Fig. 6. New and changed functions in improved BFS that implement impure features

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

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

Functions implementing impure features are in Fig. 6. The first function, `elim`, takes a space s^{∞} and two continuations `kf` and `ks`. When s^{∞} contains no answers, it calls `kf` with no argument. Otherwise, it calls `ks` with

^{c7}LKC: I do notice that in the below paragraph the leading sentence is not at the beginning (but at the second position). Putting the current first sentence at the beginning, however, looks more fun to me.

^{c1}DPF: I am not in love with the two calls to `null?`. I can think of at least two ways to fix it. Pick one.

^{c2}DPF: The use of the references to Fig 7 twice tells me that something has to change for the sake of clarity.

benchmark	size	DFS _i	DFS _{bi}	DFS _f	BFS _{imp}	BFS
very-recursive ^o	100000	579	793	2131	1438	3617
	200000	1283	1610	3602	2803	4212
	300000	2160	2836	-	6137	-
append ^o	100	31	41	42	31	68
	200	224	222	221	226	218
	300	617	634	593	631	622
revers ^o	10	5	3	3	38	85
	20	107	98	51	4862	5844
	30	446	442	485	123288	132159
quine-1	1	71	44	69	-	-
	2	127	142	95	-	-
	3	114	114	93	-	-
quine-2	1	147	112	56	-	-
	2	161	123	101	-	-
	3	289	189	104	-	-
'(I love you)-1	99	56	15	22	74	165
	198	53	72	55	47	74
	297	72	90	44	181	365
'(I love you)-2	99	242	61	16	66	99
	198	445	110	60	42	64
	297	476	146	49	186	322

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

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.

^{c3}

^{c4}

6.2 compare BFS_{imp} with BFS

In this subsection, we compare the pure subset of our 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 improved 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 BFS. And it runs slower in all benchmarks. Details about differences in efficiency are in Table 1.

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 memory.” The first two benchmarks are from TRS2. `reverso` is from Rozplokh and

^{c3}LKC: I swap `ks` and `kf` to mimic `ind-List`

^{c4}DPF: If we forget to use `after reverso`, the `o` ends up too near the next word. I fixed `reverso` on the second line of section 7, but there may be others.

Boulytchev [5]. 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 , absent^0). Our version generates de Bruijnized expressions and prevents closures from being inside a list. The two benchmarks differ in the cond^e clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to $'(I \text{ love } you)$. This benchmark is from Byrd et al. [1]. Again, the sibling benchmarks differ in the cond^e clause order of their relational interpreters. The first one has elimination rules (i.e. application, car , and cdr) at the end, while the other has them at the beginning. We conjecture that DFS_i would perform badly in the second case because elimination rules complicate the problem when synthesizing (i.e., our evaluation supports our conjecture.)

In general, only DFS_i and DFS_{bi} constantly perform well. DFS_f is just as efficient in all benchmarks except those of very-recursive^0 . Both implementations of BFS have obvious overhead in many cases. Among the three variants of DFS, which all have unfair conj , DFS_f is most resistant to clause permutation, followed by DFS_{bi} then DFS_i . Among the two implementation of BFS, our improved BFS constantly performs as well or better. Interestingly, every strategy with fair disj suffers in very-recursive^0 and DFS_f performs well elsewhere. Therefore, this benchmark might be a special case. Fair conj imposes considerable overhead constantly except in append^0 . The reason might be that strategies with fair conj tend to keep more intermediate answers in the memory.

8 RELATED WORKS

Yang [7] points out that a disjunct complex would be 'fair' if it were a full and balanced tree.

Seres et al. [6] 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.

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 (DFS_{bi}) and fair DFS (DFS_f)) and devise a new implementation of BFS. These strategies have different features in fairness: DFS_{bi} has an almost-fair disj and unfair conj . DFS_f has fair disj and unfair conj . BFS has both fair disj and fair conj .

Our quantitative evaluation shows that DFS_{bi} and DFS_f are competitive alternatives to DFS_i , the current search strategy miniKanren 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.

^{c1}DPF: There is still some clumsiness using our improved DFS, since at some point, we may want to make this clearer, so have DFS not be fonted is a little weird. I think both should always use ourDFS and macro expand it to <our improved DFS>. Then changes will be much easier.

^{c2}DPF: We might find the reference Ocanren in the two Dmitri's paper and say something here about it.

^{c3}DPF: Finish this paragraph with a clarification and you may place this anywhere in the conclusion. This may require some consultation with Weixi.

^{c4}LKC: I will do it later.

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] Dmitri Rozplokhas 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.
- [6] Silvija Seres, J Michael Spivey, and C. A. R. Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.
- [7] Edward Z. Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue* 15 (2010), 11.