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 two new, more efficient and shorter 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 implementations are more efficient than the current one.
c2
c3

ACM Reference Format:
Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. Towards a miniKanren with fair search strategies. 1, 1 (May 2019), 16 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). Byrd et al. [1] have

c1 *LKC*: submission deadline: Mon 27 May 2019
c2 *LKC*: “simpler” -> “shorter” (more objective)
c3 *LKC*: We actually have two new BFS implementations, an easy one and an improved one (BFSimp/BFSopt). Both of them runs faster than the current BFS. I didn't name the easy one because I considered it just a stepstone to BFSimp. But now I think giving names to both of them would be more clear.

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

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

c4 c5 c6 c7

c8

c9

c10 c11

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 unsuspicious 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.

[c4] *DPF*: ... 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. ...
[c5] *LKC*: I have fixed the cite to TRS and TRS2.
[c6] *LKC*: What is the point of citing dmitris?
[c7] *LKC*: Are we citing OCanren to show that miniKanren is widely used? If yes, do we need to cite other implementations as well?
[c8] *DPF*: I have decided to agree with you that we should not use BFSser, 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.
[c9] *DPF*: Occurrences of BFS (without Our) mean Seres et al. (We need only say that once, or very occasionally
[c10] *DPF*: Is the spelling of reverso truly written without the last e? Shouldn't it be reverseo with the appropriate raising of o.
[c11] *LKC*: Yes, it is reverso instead of reverseo in Dmitris' paper.

```
'((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 resourses. 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 **b**alanced **i**nterleaving depth-first search (DFS$_{bi}$), a new search strategy with almost-fair `disj`.
- propose and implement **f**air depth-first search (DFS$_f$), a new search strategy with fair `disj`.
- implement in a new way the standard breath-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. [6] and refer to "our BFS implementation" ("our BFS" for short) as our new one. We formally prove that the two BFS implementations are semantically equivalent, however, our BFS 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.

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 as 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.

```
;; DFSf (fair disj)
> (run 12 q
    (conde
      ((repeato 'a q))
      ((repeato 'b q))
      ((repeato 'c q))
      ((repeato '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))
```

```
;; BFS (fair disj)
> (run 12 q
    (conde
      ((repeato 'a q))
      ((repeato 'b q))
      ((repeato 'c q))
      ((repeato '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))
```

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.

```
;; DFS_bi (almost-fair disj)
> (run 16 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))
      ((repeat^o 'e q))))
'((b) (c) (d) (a)
  (b b) (c c) (d d) (e)
  (b b b) (c c c) (d d d) (a a)
  (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 as 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).

```
;; DFSi (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato 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))
```

```
;; DFSf (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato 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))
```

```
;; DFSbi (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato 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))
```

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, (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,

```
;; BFS (fair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x 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))
```

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] [c2]

[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."
[c2] *LKC*: I initially wrote it down as a note. I might get it wrong, but I thought you suggested me to move it into the paper.

```
;; naively fair conj
> (run 6 q
    (fresh (xs)
      (cond^e
        ((repeat^o 'a xs))
        ((repeat^o 'b xs)))
      (repeat^o xs q)))
'(((a)) ((b))
  ((a a)) ((b b))
  ((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 $1 + 2 = 3$.
c1 c2

```
;; BFS (fair conj)
> (run 12 q
    (fresh (xs)
      (cond^e
        ((repeat^o 'a xs))
        ((repeat^o 'b xs)))
      (repeat^o xs q)))
'(((a)) ((b))
  ((a) (a)) ((b) (b))
  ((a a)) ((b b))
  ((a) (a) (a)) ((b) (b) (b))
  ((a a) (a a)) ((b b) (b b))
  ((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.*

[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 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.

```
#| Goal × Goal → Goal |#
(define (disj₂ g₁ g₂)
  (lambda (s)
    (append∞ (g₁ s) (g₂ s))))

#| Space × Space → Space |#
(define (append∞ s∞ t∞)
  (cond
    ((null? s∞) t∞)
    ((pair? s∞)
     (cons (car s∞)
       (append∞ (cdr s∞) t∞)))
    (else (lambda ()
            (append∞ t∞ (s∞))))))

(define-syntax disj
  (syntax-rules ()
    ((disj) (fail))
    ((disj g₀ g ...) (disj+ g₀ g ...))))

(define-syntax disj+
  (syntax-rules ()
    ((disj+ g) g)
    ((disj+ g₀ g₁ g ...) (disj₂ g₀ (disj+ g₁ g ...)))))
```

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 (≡). 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, disj$_2$, implements binary disjunction. It applies the two disjunctive goals to the input state s and composes the two resulting search spaces with append$^\infty$. The following syntax definitions say disj is right-associative. Fig. 2 shows the implementation of conj. The first function, 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 append-map$^\infty$ applies its input goal to states in its input search spaces and composes the resulting search spaces. It reuses append$^\infty$ for search space composition. The following syntax definitions say conj is also right-associative.

```
#| Goal × Goal ⟶ Goal |#
(define (conj₂ g₁ g₂)
  (lambda (s)
    (append-map∞ g₂ (g₁ s))))

#| Goal × Space ⟶ Space |#
(define (append-map∞ g s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞)
     (append∞ (g (car s∞))
       (append-map∞ g (cdr s∞))))
    (else (lambda ()
            (append-map∞ g (s∞))))))

(define-syntax conj
  (syntax-rules ()
    ((conj) (fail))
    ((conj g₀ g ...) (conj+ g₀ g ...))))

(define-syntax conj+
  (syntax-rules ()
    ((conj+ g) g)
    ((conj+ g₀ g₁ g ...) (conj₂ g₀ (conj+ g₁ g ...)))))
```

Fig. 2. implementation of DFS$_i$ (Part II)

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 `disj₂`s, 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 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 `disj₂` (Fig. 4). The new `disj₂` calls a new and fair version of append$^\infty$. append$^\infty_{fair}$ immediately calls its helper, `loop`, with the

```
(define-syntax disj
  (syntax-rules ()
    ((disj) fail)
    ((disj g ...) (disj+ (g ...) () ()))))

(define-syntax disj+
  (syntax-rules ()
    ((disj+ () () g) g)
    ((disj+ (gₗ ...) (gᵣ ...))
     (disj₂ (disj+ () () gₗ ...)
            (disj+ () () gᵣ ...)))
    ((disj+ (gₗ ...) (gᵣ ...) g₀)
     (disj₂ (disj+ () () gₗ ... g₀)
            (disj+ () () gᵣ ...)))
    ((disj+ (gₗ ...) (gᵣ ...) g₀ g₁ g ...)
     (disj+ (gₗ ... g₀) (gᵣ ... g₁) g ...))))
```

Fig. 3. implementation of DFS_{bi}

```
#| Goal × Goal → Goal |#
(define (disj₂ g₁ g₂)
  (lambda (s)
    (append∞fair (g₁ s) (g₂ s))))

#| Space × Space → Space |#
(define (append∞fair s∞ t∞)
  (let loop ((s? #t) (s∞ s∞) (t∞ t∞))
    (cond
      ((null? s∞) t∞)
      ((pair? s∞)
       (cons (car s∞)
         (loop s? (cdr s∞) t∞)))
      (s? (loop #f t∞ s∞))
      (else (lambda ()
              (loop #t (t∞) (s∞)))))))
```

Fig. 4. implementation of DFS_f

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

```
#| Goal × Space → Space |#
(define (append-map∞_fair g s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞)
     (append∞_fair (g (car s∞))
       (append-map∞_fair g (cdr s∞))))
    (else (lambda ()
            (append-map∞_fair g (s∞))))))
```

Fig. 5. implementation of our BFS (based on DFS$_f$)

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. Our first implementation (Figure 5) is based on DFS$_f$ (not DFS$_i$). It is so similar to DFS$_f$ that we only need to apply two changes: (1) rename append-map∞ to append-map$^\infty_{fair}$ and (2) replace append∞ with append$^\infty_{fair}$ in append-map∞'s body.
c3 c4

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 the above implementation, however, it is not very clear 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 append$^\infty_{fair}$ and the append-map$^\infty_{fair}$ respects the cost information. Second, append$^\infty_{fair}$ 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. DFS$_f$ also has the space extravagance.
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 improved BFS with BFS.
c6 c7

```
#| → Space |#
(define (none)
  `(() . #f))

#| State → Space |#
(define (unit s)
  `((,s) . #f))

#| (→ Space) → Space |#
(define (step f)
  `(() . ,f))

#| Space × Space → Space |#
(define (append∞_fair s∞ t∞)
  (cons (append (car s∞) (car t∞))
    (let ((t1 (cdr s∞)) (t2 (cdr t∞)))
      (cond
        ((not t1) t2)
        ((not t2) t1)
        (else (lambda ()
                (append∞_fair (t1) (t2)))))))))

#| Goal × Space → Space |#
(define (append-map∞_fair g s∞)
  (foldr
    (lambda (s t∞)
      (append∞_fair (g s) t∞))
    (let ((f (cdr s∞)))
      (step (and f (lambda () (append-map∞_fair g (f))))))
    (car s∞)))

#| Maybe Nat × Space → [State] |#
(define (take∞ n s∞)
  (let loop ((n n) (vs (car s∞)))
    (cond
      ((and n (zero? n)) '())
      ((pair? vs)
       (cons (car vs)
         (loop (and n (sub1 n)) (cdr vs))))
      (else
       (let ((f (cdr s∞)))
         (if f (take∞ n (f)) '())))))))
```

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

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. 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 $\texttt{append}^{\infty}_{fair}$ also fixes the miserable space extravagance—the use of `append` helps us to reuse the first bag of $\texttt{t}^{\infty}$.

Kiselyov et al. [4] have demonstrated that a *MonadPlus* hides in implementations of logic programming system. our improved BFS is not an exception: $\texttt{append-map}^{\infty}_{fair}$ is like `bind`, but takes arguments in reversed order; `none`, `unit`, and $\texttt{append}^{\infty}_{fair}$ correspond to `mzero`, `unit`, and `mplus`, respectively.
c1 c2

c3 c4

Functions implementing impure features are in Fig. 7. The first function, `elim`, takes a space $\texttt{s}^{\infty}$ and two continuations `kf` and `ks`. When $\texttt{s}^{\infty}$ 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.
c5 c6

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 are semantically equivalent, since `(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.
c7

[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*: Improved.
[c5] *LKC*: I have pointed out "DFS_f also has the space extravagance" at the end of the above paragraph.
[c6] *DPF*: Let's use one more line and far less horizontal space by use 3 linefeeds. They will still align nicely.
[c7] *LKC*: fixed
[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] *LKC*: fixed
[c3] *DPF*: The use of the references to Fig 7 twice tells me that something has to change for the sake of clarity.
[c4] *LKC*: I guess you are talking about Table 1 becasue there is no Fig 7. I have change the first reference.
[c5] *DPF*: If we forget to use after $\texttt{revers}^{o}$, the o ends up too near the next word. I fixed reverso on the second line of section 7, but there may be others.
[c6] *LKC*: I have checked all uses of similar macros by text search.
[c7] *LKC*: I ref the next section instead of the table

```
#| Space × (State × Space → Space) × (→ Space) → Space |#
(define (elim s∞ kf ks)
  (let ((ss (car s∞)) (f (cdr s∞)))
    (cond
      ((pair? ss) (ks (car ss) (cons (cdr ss) f)))
      (f (step (lambda () (elim (f) kf ks))))
      (else (kf)))))

#| Goal × Goal × Goal → Goal |#
(define (ifte g₁ g₂ g₃)
  (lambda (s)
    (elim (g₁ s)
      (lambda () (g₃ s))
      (lambda (s0 s∞)
        (append-map∞fair g₂
          (append∞fair (unit s0) s∞))))))

#| Goal → Goal |#
(define (once g)
  (lambda (s)
    (elim (g s)
      (lambda () (none))
      (lambda (s0 s∞) (unit s0)))))
```

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

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. $\texttt{revers}^o$ is from Rozplokhas and 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. $\neq$, $\texttt{absent}^o$). Our version generates de Bruijnized expressions and prevents closures from being inside a list. The sibling benchmarks differ in the $\texttt{cond}^e$ clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to `'(I love you)`. They are also modified from a similar test case in Byrd et al. [1] because of the same reason. Our version generates de Bruijnized expressions. Again, the sibling benchmarks differ in the $\texttt{cond}^e$ 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

benchmark	size	DFS_i	DFS_{bi}	DFS_f	our BFS	our improved BFS	BFS
`very-recursive`^{o}	100000	166	103	412	417	451	1024
	200000	283	146	765	766	839	1875
	300000	429	346	2085	2066	1809	4408
`append`^{o}	100	17	18	17	18	17	65
	200	145	137	137	137	142	121
	300	388	384	387	383	429	371
`revers`^{o}	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

second case because elimination rules complicate the problem when synthesizing (i.e., our evaluation supports our conjecture.)

c1

c2 c3

c4

In general, only DFS_i and DFS_{bi} constantly perform well. DFS_f is just as efficient in `append`^{o}, `revers`^{o}, and `quines`. All BFS implementations have obvious overhead in most cases. Among the three variants of DFS, which all have unfair `conj`, DFS_f is most resistant to clause permutation in `quines`, followed by DFS_{bi} then DFS_i. DFS_f, however, runs out of memory in `'(I love you)`s. In contrast, DFS_{bi} still performs more stable than DFS_i. Thus, we consider DFS_{bi} a competitive alternative to DFS_i, the current standard miniKanren search strategy. Among the three implementations of BFS, our BFSs constantly perform as well or better. our improved BFS only out-performs our BFS significantly in `very-recursive`^{o} 300000, while performing as well or worse elsewhere. The reason might be that the size of bags are usually too small to be worried about. It is hard to conclude how

[c1] *LKC*: I change the benchmark ’(I love you) such that it doesn’t prevent closures from getting into pairs because I consider this restriction not reasonable. As for quines, they are already very slow so I didn’t make the changes.

[c2] *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.

[c3] *LKC*: I guess you are talking about “our improved BFS”. I have added relevant macros.

[c4] *LKC*: I rewrote the following paragraph and the table to include the additional BFS and update the ’(I love you) benchmark.

Also, all running time are re-measured by running in command-line. I used to run benchmarks with DrRacket. Thus, the time measurements are smaller.

disj fairness affect performance based on these statistics. Fair conj, however, imposes considerable overhead constantly except in append^o. The reason might be that those strategies 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.

c1

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 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.

c2 c3

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.

c1 *DPF*: We might find the reference Ocanren in the two Dmitri's paper and say something here about it.
c2 *DPF*: Finish this paragraph with a clarification and you may place this anywhere in the conclusion. This may require some consultation with Weixi.
c3 *LKC*: I will do it later.