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 (May 2019), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

c2 c3 c4

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.

[c1] *LKC*: submission deadline: Mon 27 May 2019
[c2] *DPF*: ... Perhaps it should be part of the title that we have improved on the the standard BFS implementation.
[c3] *DPF*: When you rewrite the abstract and drop the abbreviations, remember when you are rewriting the Introduction and introducing the abbreviations, that you should explain the ser of BFSser using a reference.
[c4] *LKC*: I have rewritten the abstract and explain the 'ser' in the ending paragraph of introduction, where we list contributions.

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

```
(defrel (repeat^o x out)
  (cond^e
    ((≡ `(,x) out))
    ((fresh (res)
       (≡ `(,x . ,res) out)
       (repeat^o x res)))))
> (run 4 q
    (repeat^o '* q))
'((*) (* *) (* * *) (* * * *))
```

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

```
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o '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.

```
'((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 $\text{DFS}_\text{i}$, 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 $\texttt{cond}^e$ 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 ($\text{DFS}_\text{bi}$), a new search strategy with almost-fair `disj`.
- propose and implement **f**air depth-first search ($\text{DFS}_\text{f}$), 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 name the current implementation by Seres et al. as $\text{BFS}_\text{ser}$, and our new implementation as $\text{BFS}_\text{imp}$. We prove formally that they are semantically equivalent. $\text{BFS}_\text{imp}$, however, runs faster in all benchmarks and is shorter, thus being an **imp**rovement.

## 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.

c1

Now we elaborate fairness by running more queries about $\texttt{repeat}^o$. 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. $\text{DFS}_\text{i}$, the current search strategy, however, results in many more lists of `a`s than lists of other letters. And some letters (e.g. `c` and `d`) are rarely seen. The situation would be exacerbated if the $\texttt{cond}^e$ would have contained more clauses.

```
;; DFSi (unfair disj)
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o '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))
```



c1 *LKC*: The following paragraph is extended to include an explanation on `run*`

Under the hood, the cond$^e$ here is allocating computational resources to four trivially different search spaces. The unfair `disj` in $\text{DFS}_\text{i}$ 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
    (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))
```

```
;; 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))
```

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

```
;; DFSbi (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))
```

$\text{DFS}_\text{bi}$ 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, $\text{DFS}_\text{bi}$ 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

c1

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).

```
;; DFSi (unfair conj)
> (run 12 q
    (fresh (x)
      (cond^e
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeat^o 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)
      (cond^e
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeat^o 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)
      (cond^e
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeat^o 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)
      (cond^e
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeat^o 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

[c1] *LKC*: I change '1/4 in the answer' to '1/4 in the answer list'.

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 c3

```
;; 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 $1 + 2 = 3$.
c4 c5



[c1] *DPF*: I think that should be said as directly as in your note.
[c2] *LKC*: I state this fact at the beginning of this section.
[c3] *LKC*: 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.
[c4] *DPF*: Next to the last line of the above paragraph the word same has too many meanings. It could be have the same cost or it could be failing to distinguish two list that would be the same if we substituted a b for an a. It needs to be clarified. It might help to work out the actual numbers
[c5] *LKC*: I have rewritten the relevant sentences.

```
;; 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.*

## 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, `disj2`, 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, `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`$^\infty$ 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.
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 `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

[c1] *DPF*: This came out fabulous. I think I spotted some missed opportunities later, which I will point out with additional notes.
[c2] *LKC*: I think we can simply remove `cond`$^e$ code and its accompany description.

```
#| 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)



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 first argument, `s?`, set to `#t`, which indicates that we haven't swapped `s`$^\infty$ and `t`$^\infty$. The swapping happens at the third `cond` clause in the helper, where `s?` is updated accordingly. The first two `cond` clauses essentially copy the `car`s 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`$^\infty$ and `t`$^\infty$ are ended with a thunk, and that we have swapped them. In this case, we construct a new thunk. The new thunk swaps two

```
#| 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)

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$^\infty$ with append$^\infty_{fair}$ in append-map$^\infty$'s body (2) rename append-map$^\infty$ to append-map$^\infty_{fair}$.

c1

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 this 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 append$^\infty_{fair}$ and the append-map$^\infty_{fair}$ respects the cost information.

c1 *DPF*: When or where do you swap them back? This could be still clearer.
c2 *LKC*: Improved.
c1 *LKC*: Applying the above changes won't result in $BFS_{ser}$, but yet another implementation of BFS.

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

(define-syntax disj+
  (syntax-rules ()
    ((disj+ () () g) g)
    ((disj+ (g_l ...) (g_r ...))
     (disj2 (disj+ () () g_l ...)
            (disj+ () () g_r ...)))
    ((disj+ (g_l ...) (g_r ...) g_0)
     (disj2 (disj+ () () g_l ... g_0)
            (disj+ () () g_r ...)))
    ((disj+ (g_l ...) (g_r ...) g_0 g_1 g ...))
     (disj+ (g_l ... g_0) (g_r ... g_1) g ...)))
```

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

```
#| Goal × Goal → Goal |#
(define (disj2 g_1 g_2)
  (lambda (s)
    (append^∞_fair (g_1 s) (g_2 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 $\text{DFS}_{\text{f}}$

Second, $\text{append}^{\infty}_{fair}$ 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 search spaces.
c2

[c2] *LKC*: Should I point out that $\text{DFS}_{\text{f}}$ also has the miserable space extravagance?

In the following subsections, we firstly describe $\text{BFS}_{\text{imp}}$, an improved BFS implementation which manages cost information in a more clear way and is less extravagant in memory usage. Then we compare $\text{BFS}_{\text{imp}}$ with $\text{BFS}_{\text{ser}}$, the current implementation by Seres et al..

## 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 $\text{BFS}_{\text{ser}}$ in our proof.

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

c1

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

Functions implementing impure features are in Fig. 6. 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.

c2

## 6.2 compare $\text{BFS}_{\text{imp}}$ with $\text{BFS}_{\text{ser}}$

c3

In this subsection, we compare the pure subset of $\text{BFS}_{\text{imp}}$ with $\text{BFS}_{\text{ser}}$. We focus on the pure subset because $\text{BFS}_{\text{ser}}$ is designed for a pure relational programming system. We proof in Coq that these two search strategies are are semantically equivalent, (i.e., `(run n ? g)` produces the same result (See supplements for the formal proof). To compare efficiency, we translate $\text{BFS}_{\text{ser}}$'s Haskell code into Racket (See supplements for the translated code). The translation is direct due to the similarity in two relational programming systems. The translated code is longer than $\text{BFS}_{\text{imp}}$. 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. $\texttt{revers}^{o}$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 two benchmarks differ in the $\text{cond}^{e}$ clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to `'(I love you)`. This benchmark is from Byrd et al. [1]. Again, the sibling benchmarks differ in the

c1 *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.

c2 *LKC*: I swap `ks` and `kf` to mimic `ind-List`

c3 *LKC*: I have rewritten this subsection.

```
#| → 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. 5. new and changed functions in optimized BFS that implements pure features

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

#| 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. 6. new and changed functions in improved BFS that implement impure features



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 $\text{DFS}_\text{i}$ 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 $\text{DFS}_\text{i}$ and $\text{DFS}_\text{bi}$ constantly perform well. $\text{DFS}_\text{f}$ is just as efficient in all benchmarks but `very-recursive`$^o$. Both implementations of BFS have obvious overhead in many cases. Among the three variants of DFS, which all all have unfair `conj`), $\text{DFS}_\text{f}$ is most resistant to clause permutation, followed by $\text{DFS}_\text{bi}$ then $\text{DFS}_\text{i}$. Among the two implementation of BFS, our improved $\text{BFS}_\text{imp}$ constantly performs as well or better. Interestingly, every strategy with fair `disj` suffers in `very-recursive`$^o$ and $\text{DFS}_\text{f}$ performs well elsewhere. Therefore, this benchmark might be a special case. Fair `conj` imposes considerable overhead constantly except in `append`$^o$. 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.

| benchmark | size | $DFS_i$ | $DFS_{bi}$ | $DFS_f$ | $BFS_{imp}$ | $BFS_{ser}$ |
|---|---|---|---|---|---|---|
| `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



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, and that BFS is less practical.

Our new implementation of 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 c2

[c1] *DPF*: Finish this paragraph with a clarification and you may place this anywhere in the conclusion. This may require some consultation with Weixi.

[c2] *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.