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 shorter implementation of the standard breadth-first search. Using quantitative evaluation, we argue that each depth-first search is a competitive alternative to the standard one, and that our improved 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), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

miniKanren is a family of relational programming languages. Friedman et al. [3, 4] introduce miniKanren and its implementation in *The Reasoned Schemer* and *The Reasoned Schemer, 2nd Ed* (TRS2). Although originally written in Scheme, miniKanren has been implemented in many other languages, including multiple ones using the same language. (See miniKanren.org.) This has come about because of Hemann et al. [5]'s work on microKanren, a core of miniKanren comprised of only 54 LOC. Also, miniKanren has been ported to Ocaml, and named OCanren[7]). Byrd et al. [2] have demonstrated that miniKanren problems that at first reading seem impossible are indeed possible by using a Scheme interpreter written in miniKanren, thereby having access to logic variables within Scheme. miniKanren.org contains the seeds of many interesting problems and their solutions.

c2 c3

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

c1 *LKC*: submission deadline: Mon 27 May 2019
c2 *DPF*: I thought we agreed to use BFSser throughout? In that sense it had emerged a few days ago, and now it has re-emerged. It is what we are going to use instead of "current".
c3 *LKC*:

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

XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(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 novice miniKanren users—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. Boskin et al. [1] propose and implement a means to eliminate or re-order disjunctive clauses to accommodate varying search needs such as these.
c1

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) by Seres et al. [9], a search strategy with fair `disj` and fair `conj`. We refer to "the current BFS" as their implementation and "the improved BFS" as our new one. We formally prove that the two implementations are semantically equivalent, however, the improved 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*, *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. It is perfectly fine, however, to use `run`* with any search strategy.

2.1 fair `disj`

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer list. DFS$_i$, TRS2's 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 more clauses, the worse the situation.

```
;; 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))
```

c1 *LKC*: I have cited Boskin in the above paragraph.

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

```
;; 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))
```

```
;; 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. 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 with 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. [9], which is only for binary disjunction. We generalize it to a multi-arity one.

Definition 2.1 (fair `disj`). *A* `disj` *is fair if and only if it allocates computational resources evenly to 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 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: DFS_i, DFS_{bi}, and DFS_f, however, results in many more lists of a's than lists of other letters. And some letters are rarely seen. Here again, as the number of clauses grows, the situation worsens.

Although some strategies have a different level of fairness in disj, they have the same behavior when there is no call to a relational definition in 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 space including four trivially different states. Applying the next conjunctive goal, (repeato x q), produces four trivially different spaces. In the examples above, all search strategies allocate more computational resources to the space of a. On the contrary, fair conj would allocate resources evenly to each 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 search strategy is *complete* if and only if "every correct answer would be discovered after some finite time" [9], 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.

```
;; 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 buckets in spaces, where each bucket contains finite states, and to allocate resources evenly among spaces derived from states in the same bucket. It is up to a search strategy designer to decide by what criteria to put states in the same bucket, and how to allocate resources among spaces related to different buckets.

BFS puts states of the same cost in the same bucket, and allocates resources carefully among spaces related to different buckets 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 relational definitions required to find the answer) [9]. In the above examples, the cost of each answer is equal to their lengths because we need to apply `repeat`^{o} n times to find an answer of length n. 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 lists plus the length of its outer list. For example, the cost of `((a) (a))` is $1 + 2 = 3$.

```
;; 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 spaces produced from states in the same bucket. A bucket is a finite collection of states. And search strategies with fair* `conj` *should represent spaces with possibly unbounded collections of buckets.*

Definition 2.5 (unfair `conj`). *A* `conj` *is unfair if and only if it is not fair.*

```
#| 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", provide 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: `s`s name states; `g`s (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 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 space. The helper function `append-map`$^\infty$ applies its input goal to states in its input space and composes the resulting spaces. It reuses append$^\infty$ for 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

The strategy **b**alanced **i**nterleaving 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. When there are one or more disjunctive goals, the new `disj` builds a balanced binary tree whose leaves are the goals and whose nodes are `disj`$_2$s, hence the name of this search strategy. In contrast, the `disj` in DFS$_i$ constructs the binary tree in a particularly unbalanced form. We list the new `disj` with its helper in Fig. 3. 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.

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$^\infty$. 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+ (g_l ...) (g_r ...))
     (disj_2 (disj+ () () g_l ...)
             (disj+ () () g_r ...)))
    ((disj+ (g_l ...) (g_r ...) g_0)
     (disj_2 (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 DFS_{bi}

```
#| Goal × Goal → Goal |#
(define (disj_2 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 DFS_f

first argument, s?, initialized to #t, which indicates that we haven't swapped s^∞ and t^∞. The swapping happens at the third cond clause in loop, 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. stepping-stone toward the improved BFS (based on DFS_f)

above, is just for swapping. When the fourth and last clause runs, we know that both $\texttt{s}^\infty$ and $\texttt{t}^\infty$ are ended with a thunk, and that we have swapped them. In this case, we construct a new thunk. The new thunk swaps back the two spaces in the recursive call to `loop`. This is unnecessary for fairness—we do it to produce answers in a more readable order.

6 BREADTH-FIRST SEARCH

BFS has both fair `disj` and fair `conj`. Our first BFS implementation (Fig. 5) serves as a "stepping-stone" toward the improved BFS. It is so similar to DFS_f (not DFS_i) that we only need to apply two changes: (1) rename $\texttt{append-map}^\infty$ to $\texttt{append-map}^\infty_{fair}$ and (2) replace $\texttt{append}^\infty$ with $\texttt{append}^\infty_{fair}$ in $\texttt{append-map}^\infty_{fair}$'s body.

This implementation can be improved in two ways. First, as mentioned in subsection 2.2, BFS puts answers in buckets and answers of the same cost are in the same bucket. In the above implementation, however, it is not obvious how we manage cost information—the `car`s of a space have cost 0 (i.e., they are all in the same bucket), and every thunk indicates an increment in cost. It is even more subtle that $\texttt{append}^\infty_{fair}$ and the $\texttt{append-map}^\infty_{fair}$ respects the cost information. Second, $\texttt{append}^\infty_{fair}$ is extravagant in memory usage. It makes $O(n + m)$ new `cons` cells every time, where n and m are the sizes of the first buckets of two input spaces. DFS_f is also space extravagant.

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

6.1 improved BFS

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

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

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

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

```
#| → 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 the improved BFS that implements pure features

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

#| 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 the improved BFS that implement impure features

Kiselyov et al. [6] have demonstrated that a *MonadPlus* hides in implementations of logic programming systems. the improved BFS is not an exception: append-map$^{\infty}_{fair}$ is like bind, but takes arguments in reversed order; none, unit, and append$^{\infty}_{fair}$ correspond to mzero, unit, and mplus, respectively.

Functions implementing impure features are in Fig. 7. The first function, elim, takes a space s^{∞} and two continuations fk and sk. When s^{∞} contains no answers, it calls fk. Otherwise, it calls sk with the first answer and the rest of the space. This function is similar to an eliminator of spaces, hence the name. The remaining functions are as before.

6.2 compare the improved BFS with the current BFS

In this subsection, we compare the pure subset of the improved BFS with the current BFS. We focus on the pure subset because the current BFS is designed for a pure relational programming system. We prove in Coq that these two search strategies are semantically equivalent, since the result of (run n ? g) is the same either way. (See supplements for the formal proof.) To compare efficiency, we translate the current 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 the improved BFS. And it runs slower in all benchmarks. Details about differences in efficiency are in section 7.

benchmark	size	DFS_i	DFS_{bi}	DFS_f	improved BFS	current BFS
very-recursive^o	100000	166	103	412	451	1024
	200000	283	146	765	839	1875
	300000	429	346	2085	1809	4408
append^o	100	17	18	17	17	65
	200	145	137	137	142	121
	300	388	384	387	429	371
revers^o	10	3	4	3	18	36
	20	35	35	33	3070	3695
	30	329	333	315	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	96
	198	22	71	-	374	652
	297	24	292	-	1668	3759
'(I love you)-2	99	485	179	-	44	94
	198	808	638	-	414	630
	297	1265	916	-	1651	3691

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

7 QUANTITATIVE EVALUATION

In this section, we compare the efficiency of the search strategies. A concise description is in Table 1. A hyphen means "running out of 500 MB memory". The first two benchmarks are from TRS2. `revers`^{o} is from Rozplokhas and Boulytchev [8]. The next two benchmarks about generating quines are based on a similar test case in Byrd et al. [2]. We modify the relational interpreters because we don't have disequality constraints (e.g. `absent`^{o}). The sibling benchmarks differ in the `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 based on a similar test case in Byrd et al. [2]. Again, we modify the relational interpreters for the same reason. And 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 `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} is still more stable than DFS_i. Thus, we consider DFS_{bi} a competitive alternative to DFS_i. Among the two implementations of BFS, the improved BFS constantly performs as well or better. It is hard to conclude how `disj` fairness affects 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 [10] points out that a disjunct complex would be 'fair' if it were a full and balanced tree.

Seres et al. [9] 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.

Rozplokhas and Boulytchev [8] address the non-commutativity of conjunction, while our work about `disj` fairness addresses the non-commutativity of disjunction.

9 CONCLUSION

We analyze the definitions of fair `disj` and fair `conj`, then propose a new definition of fair `conj`. Our definition of fair `conj` 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} is a competitive alternative to DFS_{i}, the current miniKanren search strategy, and that the improved BFS is more practical than the current BFS.

We prove formally that the improved BFS is semantically equivalent to the current BFS. But, the improved BFS is shorter and performs better in comparison with a straightforward translation of their Haskell code.

We acknowledge that we are making large claims about efficiency given that we have just a few benchmarks, but our intuition is based on sound principles. ...
c1 c2

ACKNOWLEDGMENTS

REFERENCES

[1] Benjamin Strahan Boskin, Weixi Ma, David Thrane Christiansen, and Daniel P. Friedman. 2018. A Surprisingly Competetive Conditional Operator: for miniKanrenizing the Inference Rules of Pie *(Scheme '18)*. St Louis, MO, USA.

[2] 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).

[3] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.

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

[5] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016*. ACM Press. https://doi.org/10.1145/2989225.2989230

[6] 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.

[7] Dmitry Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).

[8] 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.

[9] Silvija Seres, J Michael Spivey, and C. A. R. Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.

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

[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 have sent an email to Weixi on the clarification.