[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 ($\mathrm{DFS_i}$), the balanced interleaving depth-first search ($\mathrm{DFS_{bi}}$), the fair depth-first search ($\mathrm{DFS_f}$), and the standard breadth-first search ($\mathrm{BFS_{ser}}$). $\mathrm{DFS_{bi}}$ and $\mathrm{DFS_f}$ are new. And we present a new, more efficient and simpler implementation of $\mathrm{BFS_{ser}}$. Using quantitative evaluation, we argue that $\mathrm{DFS_{bi}}$ and $\mathrm{DFS_f}$ are competitive alternatives to $\mathrm{DFS_i}$, and that our $\mathrm{BFS_{ser}}$ 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 Reason Schemer* and *The Reasoned Schemer, 2nd Ed* (TRS2). Byrd et al. [1] have proved that miniKanren programs are useful in solving many problems as in miniKanren.org.

[c2] [c3]

[c4] [c5]

A subtlety arises when a `cond`^{e} contains many clauses: not every clause has an equal chance to contribute to the result. As an example, consider the following relation `repeato` and its invocation.

[c1] *LKC*: submission deadline: Mon 27 May 2019
[c2] *DPF*: I think that the abbreviations for the search strategies should not come up until the Introduction.
[c3] *LKC*: Do you mean they should not come up in Abstract?
[c4] *DPF*: When we talk about complete and incomplete, I think we should make it explicit that we are only talking about (run n (x y z) g ...) and not (run* (x y z) g ...), but an alternative is that a stream is *productive* or *unproductive*.
[c5] *LKC*: I have added a sentence at the place where we define completeness.

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 (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 repeato 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 TR2, 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 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. A little miniKanrener, however, may beg to differ–understanding implementation details and fiddling with clause order is not the first priority of a beginner.

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 contribution, we

- propose and implement balanced interleaving depth-first search (DFS_{bi}), a new search strategy with almost-fair `disj`.
- propose and implement fair depth-first search (DFS_{f}), a new search strategy with fair `disj`.
- implement in a new way breath-first search (BFS_{ser}), a search strategy with fair `disj` and fair `conj`. And we prove formally that our BFS_{ser} implementation is semantically equivalent to BFS_{ser}. Our code runs faster in all benchmarks and we believe it is simpler.

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*, *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 `repeato`.

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 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.

```
;; DFSbi (almost-fair disj)
> (run 16 q
    (conde
      ((repeato 'a q))
      ((repeato 'b q))
      ((repeato 'c q))
      ((repeato 'd q))
      ((repeato '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.*

c1

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

[c1] *LKC*: I might be wrong, but '*so* evenly ... that the maximal ...' looks more natural to me

2.2 fair conj

[c2]

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 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).

```
;; DFS_i (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))
```

```
;; DFS_f (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))
```

```
;; DFS_bi (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 repeato 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

[c2] *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 positition in large enough answer lists.
c1 c2

```
;; naively fair conj
> (run 6 q
    (fresh (xs)
      (conde
        ((repeato 'a xs))
        ((repeato 'b xs)))
      (repeato 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_{ser} 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 are the same. Here the cost of each answer is equal to the length of its inner list plus the length of its outer list.

```
;;BFSser (fair conj)
> (run 12 q
    (fresh (xs)
      (conde
        ((repeato 'a xs))
        ((repeato 'b xs)))
      (repeato 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.

[c1] *DPF*: Does this assume that we never use run*, since in fact, the paper does not include and uses of run*. Something should be mentioned around here.

[c2] *LKC*: We never use run* in this paper because fairness is only interested when we have an unbounded number of answers. However, it is perfectly fine to use run* with any search strategies.

```
(define-syntax cond^e
  (syntax-rules ()
    ((cond^e (g ...) ...)
     (disj (conj g ...) ...))))
```

Fig. 1. implementation of cond^e

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 ($\equiv$).

The definition of cond^e (Fig. 1) is shared by all search strategies. It relates clauses disjunctively, and goals in the same clause conjunctively.

c1 c2

Fig. 2 and Fig. 3 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. 2 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. 3 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.

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. 4. 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.

c1 *DPF*: If we use the name subscript, the subscripts should use an underscore and we should do it for gunderscorel and gundescorer. If we can use an l that looks less like a 1, that would even be better. Look at the l in TLT.

c2 *LKC*: I have turned those 'subscripts' to real subscripts.

```
#| 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. 2. implementation of $\mathrm{DFS_i}$ (Part I)

5 FAIR DEPTH-FIRST SEARCH

c3 c4

Fair DFS ($\mathrm{DFS_f}$) has fair `disj` and unfair `conj`. The implementation of $\mathrm{DFS_f}$ differs from $\mathrm{DFS_i}$'s in `disj2` (Fig. 5). The new `disj2` calls a new and fair version of append^{∞}. $\mathsf{append}^{\infty}_{fair}$ immediately calls its helper, `loop`, with the first argument, `s?`, set to `#t`, which indicates that we haven't swapped $\mathtt{s}^{\infty}$ and $\mathtt{t}^{\infty}$. 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 $\mathtt{s}^{\infty}$ and $\mathtt{t}^{\infty}$ are ended with a thunk and we have swapped them. In this case, we construct a new thunk. The new thunk swaps two spaces back in the recursive call. This is unnecessary for fairness. We do it to produce answers in a more natural order. c1 c2

[c3] *DPF*: Why does the pair? case precede the null? case? Is it simply more efficient? It shouldn't be

[c4] *LKC*: I did for efficiency. I have re-ordered two cases back.

[c1] *DPF*: When or where do you swap them back?

[c2] *LKC*: In the recursion

```
#| 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. 3. implementation of DFS$_i$ (Part II)

6 BREADTH-FIRST SEARCH

BFS$_{ser}$ has both fair `disj` and fair `conj`. Our implementation is based on DFS$_f$ (not DFS$_i$). To implement BFS$_{ser}$ based on DFS$_f$, we need append-map$^{\infty}_{fair}$ in addition to append$^{\infty}_{fair}$. The only difference between append-map$^{\infty}$ and append-map$^{\infty}_{fair}$ is that the latter calls append$^{\infty}_{fair}$ instead of append$^{\infty}$.

The implementation is improvable in two ways. First, as mentioned in subsection 2.2, BFS$_{ser}$ puts answers in bags and answers of the same cost are in the same bag. In this implementation, however, it is unclear where this information is recorded. 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 search spaces. We address these issues in the first subsection.

We prove formally our implementation is equivalent to the original one by Seres et al. [6]. The details are in the second subsection.

```
(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. 4. 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. 5. implementation of DFS_{f}

```
#| → 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 optimized BFS that implements pure features

6.1 optimized BFS

c1 c2

As mentioned in subsection 2.2, BFS puts answers in bags and answers of the same cost are in the same bag. The cost information is recorded 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 $\texttt{append}^{\infty}_{fair}$ and the $\texttt{append-map}^{\infty}_{fair}$ respects the cost information. We make these facts more obvious 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 others in Fig. 7). We compare these functions with Seres et al.'s implementation in our proof. The first three functions in Fig. 6 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 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 bag of $\texttt{t}^{\infty}$.

Kiselyov et al. [4] has shown that a *MonadPlus* hides in implementations of logic programming system. Our BFS_{ser} implementation 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.

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

6.2 compare our implementation with the original one

In this section, we compare the pure subset of our implementation with the original one. We focus on the pure subset because the original system is pure. Seres et al. represent search spaces with streams of lists of answers, where each list is a bag.

To compare efficiency, we translate BFS_{ser}'s Haskell code into Racket (See supplements for the translated code). The translation is direct due to the similarity in both logic programming systems and search space representations. The translated code is longer and slower than our implementation. Details about differences in efficiency are in the table below.

We use Coq to show that the two implementations are semantically equivalent, (i.e., `(run n ? g)` produces the same result (See supplements for the formal proof).

7 QUANTITATIVE EVALUATION

Here we compare the efficiency of search strategies. A concise description is in Table 1. A hyphen means "running out of memory." The first two benchmarks are from TRS2. `reverso` is from 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. ≠, $\texttt{absent}^o$). Our version generates de Bruijnized expressions and prevents closures from being inside a list. The two 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`

[c1] *DPF*: Never refer to a literal section like 2.2. Instead stick a label where it starts right near the section number and stick another label where the subsection starts. Then ref the two labels and place a period between them.
[c2] *LKC*: Got it! Fixed

```
#| Space × (State × Space → Space) × (→ Space) → Space |#
(define (elim s∞ ks kf)
  (let ((ss (car s∞)) (f (cdr s∞)))
    (cond
      ((and (null? ss) f)
       (step (lambda () (elim (f) ks kf))))
      ((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 (s0 s∞)
        (append-map∞_fair g2
          (append∞_fair (unit s0) s∞)))
      (lambda () (g3 s)))))

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

Fig. 7. new and changed functions in optimized BFS that implements impure features

`love you`). This benchmark is from Byrd et al. [1]. Again, the sibling benchmarks differ in the `cond`^{e} clause order of their relational interpreters. The first one has elimination rules (i.e. application, `car`, and `cdr`) at the end, while the other has them at the beginning. We conjecture that DFS_i would perform badly in the second case because elimination rules complicate the problem when synthesizing (i.e., our evaluation supports our conjecture.) c1 c2

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

[c1] *DPF*: Below, veryrecursiveo (if possible, should have its o raised.) One way to do this is to have a different macro. In the table there are other occurrences that need their names with a different macro.
[c2] *LKC*: Fixed

benchmark	size	DFS_i	DFS_{bi}	DFS_f	our BFS_{ser}	original 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

8 RELATED WORKS

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

Seres et al. [6] describe a breadth-first search strategy. We present another implementation. Our implementation is semantically equivalent to theirs. But, ours is shorter and performs better in comparison with a straightforward translation of their Haskell code.

9 CONCLUSION

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

We devise two new search strategies (i.e. balanced interleaving DFS (DFS_{bi}) and fair DFS (DFS_f)) and devise a new implementation of BFS. These strategies have different features in fairness: $_{bi}$ has an almost-fair `disj` and unfair `conj`. DFS_f has fair `disj` and unfair `conj`. BFS_{ser} 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_{ser} is less practical than other strategies. [c1] [c2]

[c1] *DPF*: what?

[c2] *LKC*: other search strategies

We present another implementation of BFS_{ser}. Our implementation is semantically equivalent to the original one. But, ours is shorter and performs better in comparison with a straightforward translation of their Haskell code.
[c3] [c4]

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.

[c3] *DPF*: Finish this paragraph with a clarification and you may place this anywhere in the conclusion. This may require some consultation with Weixi.

[c4] *LKC*: I will do it later.