

# Run, Run as Fast as You Can, You Can't Catch Me, I'm The microKanren.\*

## Removing More Macros From microKanren

Yafei Yang<sup>[0009-0006-1402-8014]</sup>, Darshal Shetty<sup>[0000-0001-8365-5763]</sup>, and  
Daniel P. Friedman<sup>[0000-0001-9992-1675]</sup>

Indiana University, Bloomington, USA  
{yafyang,dcshetty}@iu.edu, dfried@indiana.edu

**Abstract.** We present a functional implementation for microKanren that disentangles `run` and `run*`. Prior to our implementation, one can get either partial results using `run` or all results using `run*`. If we use `run` with a number of expected results and want to get more results from the same goal later, we must change the argument passed to `run` and execute the modified program, causing a repetitive computation. Our `run` makes it possible to return results in an incremental way without having to change a program repeatedly.

**Keywords:** logic programming · miniKanren · delayed recursion

## 1 Introduction

In the second edition of *The Reasoned Schemer* (*TRS2e*), two higher-level operators are provided: `run` and `run*`. [1] `run` takes an expression that should evaluate to a non-negative number  $n$ , one or more logic variables (We abbreviate “logic variable” to “variable” as was done in *TRS2e*), and one or more goals. Then it tries to find at most  $n$  values of the given variables that satisfy those goals. `run*` does a similar job, except that it doesn't use  $n$  and tries to find all possible values of the given variables. Because we don't know in advance if running a goal will go into an infinite loop, we usually want to start by using `run` with a small  $n$ . If `run` does give us some results, we can try increasing  $n$  or using `run*` to get more.

For simple relations, the process described above works perfectly. We don't need to wait for too long before realizing that running a goal might go into an infinite loop. If these relations become more complex, however, problems emerge with the process. First, to get the expected number of results, we must modify our programs over and over again, which can be tedious and error-prone. Second, we don't know what a proper starting number is. If we start with a very small number, it's possible that we need to modify our programs many times. More

---

\* Our title was inspired by the *The Gingerbread Man*, which first appeared as *The Gingerbread Boy*, St. Nicholas Magazine, May 1875, retold by an anonymous author.

importantly, every time we modify  $n$ , we need to rerun our program, causing several repetitive computations.

We attempt to solve these problems by a different implementation of `run`. Instead of returning the expected number of results from a stream and discarding the rest, our `run` gives users access to the rest of the stream without exposing the implementation details. If we want to get more results, we can resume from where we stop, rather than starting over again. In *TRS2e*, `run` and `run*` are implemented with macros. Recent work on microKanren [2] makes it possible to replace these macros with functions, so we choose the same technique. In section 2, we give a familiar example. In section 3, we give the new implementation of `run` and `run*`, together with the required changes to some internal functions. For clarity, we disentangle `run*` from `run` by using different helper functions. We conclude with an example that uses the new `run`.

## 2 Problem

We start with the “Hello, World!” program using miniKanren, `appendo`, written with the functional `disj` and `conj` [2]

```
(defrel (appendo l t out)
  (disj
    (conj (== '() l) (== t out))
    (fresh (a d res)
      (== `(,a . ,d) l)
      (== `(,a . ,res) out)
      (appendo d t res))))
```

We want to know how to get `'(a b c d e)` by appending two lists, so we ask for one answer

```
(run 1 (x y)
  (appendo x y '(a b c d e)))
```

Fortunately, we get a result

```
'((( (a b c d e)))
```

that says `x` is the empty list, and `y` is `'(a b c d e)`. We are curious if there are more results, so we increase the number of queries,

```
(run 4 (x y)
  (appendo x y '(a b c d e)))
```

and we get more results.

```
'((( (a b c d e))
  ((a) (b c d e))
  ((a b) (c d e))
  ((a b c) (d e)))
```

This time we think it might be safe to ask for all possible results, so we use `run*`

```
(run* (x y)
      (appendo x y '(a b c d e)))
```

to get all results

```
'((( (a b c d e))
  ((a) (b c d e))
  ((a b) (c d e))
  ((a b c) (d e))
  ((a b c d) (e))
  ((a b c d e) ()))
```

Each use of `run` needs to recompute the result returned from the previous one. This is not a problem for such a simple relation, but what if our relation contains thousands of subgoals? It would be a price too high to pay. Can we avoid repetitive computations? That's where our `run` comes into play.

### 3 Adapting run

#### 3.1 Functional Implementation of run

We review the functional implementation [2] of four goal constructors `disj`, `conj`, `conda`, and `once` (See Fig. 1 and Fig. 2.) The definitions above the dashed line are user-level operators, and the definitions below the dashed line are internal functions. In addition, we also include the functional implementation of `run`

```
(define (call/initial-state n fq)
  (let ((q (var 'q)))
    (map (reify q)
         (take∞ n ((fq q) initial-state)))))
```

Here `call/initial-state` takes `#f` or a natural number as `n` and a function  $f_q$ .  $f_q$  should take a variable and return a goal. `call/initial-state` invokes `take∞` on `n` along with a stream returned by applying  $(f_q \text{ } q)$  to the initial state. The result is a stream that can be further processed by `reify`, which will return the reified form of the variable `q`.

We choose to use a `vector` of length 1 to represent a variable as mentioned in the framernote on page 145 of *TRS2e*.

```
(define (var q)
  (vector q))
```

Each time `var` is invoked, a new location is allocated. To compare two variables, we compare their locations.

The function `call/fresh` in *TRS2e* takes a symbol `y` and a function `f`. It invokes `var` with `y` to generate a new variable, then it passes that variable to  $f_q$ , which should return a goal.

```

(define ((disj . g*) s)
  (cond
    ((null? g*) '())
    (else (D ((car g*) s) (cdr g*) s))))

(define ((conj . g*) s)
  (cond
    ((null? g*) (cons s '()))
    (else (C (cdr g*) ((car g*) s)))))

```

---

```

(define (D s∞ g* s)
  (cond
    ((null? g*) s∞)
    (else
     (append∞ s∞
      (D ((car g*) s) (cdr g*) s)))))

(define (C g* s∞)
  (cond
    ((null? g*) s∞)
    (else
     (C (cdr g*)
      (append-map∞ (car g*) s∞)))))

```

Fig. 1. A functional disj and conj.

```

(define (call/fresh y fq)
  (fq (var y)))

```

Finally, we disentangle `run` and `run*` by using different helper functions. We redefine `run` and `run-goal` as

```

(define (run n fq)
  (call/fresh 'q
   (lambda (q)
     (run-goal n (fq q) q))))

(define (run-goal n g q)
  (reify∞ n (g initial-state) (reify q)))

```

`run` no longer takes `#f`. The first argument of `run-goal` is restricted to a natural number. The second argument is a goal. The third argument is a variable.

```

(define ((conda . g*) s)
  (cond
    ((null? g*) '())
    (else (A (cdr g*) ((car g*) s) s))))

(define ((once g) s)
  (0 (g s)))

(define (A g* s∞ s)
  (cond
    ((null? g*) s∞)
    ((null? (cdr g*)) (append-map∞ (car g*) s∞))
    (else (ifs∞te s∞ (car g*) (cdr g*) s))))

-----

(define (ifs∞te s∞ g g+ s)
  (cond
    ((null? s∞) (A (cdr g+) ((car g+) s) s))
    ((pair? s∞) (append-map∞ g s∞))
    (else (lambda () (ifs∞te (s∞) g g+ s)))))

(define (0 s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞) (cons (car s∞) '()))
    (else (lambda () (0 (s∞))))))

```

Fig. 2. A functional conda and once.

`reify∞` can be defined as

```

(define (reify∞ n S∞ T)
  (cond
    ((or (zero? n) (null? S∞)) '())
    ((pair? S∞) (cons (T (car S∞))
                      (reify∞ (sub1 n) (cdr S∞) T)))
    (else (reify∞ n (S∞) T))))

```

The first argument of `reify∞` must also evaluate to a natural number. The second argument is a stream. The third argument `T` is a function that transforms a substitution into its reified form. The function `reify∞` absorbs the responsibility of `map` and `take∞`.

Next we define `run*` with another helper function `reify*∞`

```

(define (run* fq)
  (call/fresh 'q
    (lambda (q)
      (reify*∞ ((fq q) initial-state) (reify q))))))

(define (reify*∞ S∞ T)
  (cond
    ((null? S∞) '())
    ((pair? S∞) (cons (T (car S∞))
                      (reify*∞ (cdr S∞) T)))
    (else (reify*∞ (S∞) T))))

```

Since `run*` always returns all results, we don't need to pass a number. `reify*∞` either returns all results or goes into an infinite loop.

### 3.2 Improving reify<sup>∞</sup>

The functions `run` and `run*` do nothing more than their macro variants. We still need to modify and rerun our programs repeatedly to get more results. Why? Observe `reify∞`, we can see that in the base case, if `n` is 0, `reify∞` will always return `'()`, no matter what `S∞` is, leaving us no option to fetch more results from `S∞`.

We solve this problem by returning a finite list of reified results together with `S∞`. To do that, we first convert `reify∞` to accumulator-passing style by adding an extra argument `acc`

```

(define (reify∞ n S∞ T acc)
  (cond
    ((or (zero? n) (null? S∞)) acc)
    ((pair? S∞) (reify∞ (sub1 n) (cdr S∞) T
                        (cons (T (car S∞)) acc)))
    (else (reify∞ n (S∞) T acc))))

```

Next we change the return value of `reify∞` to a pair, whose `car` is the reified result list, and `cdr` is the remaining `S∞`.

```

(define (reify∞ n S∞ T acc)
  (cond
    ((or (zero? n) (null? S∞)) (cons acc S∞))
    ((pair? S∞) (reify∞ (sub1 n) (cdr S∞) T
                        (cons (T (car S∞)) acc)))
    (else (reify∞ n (S∞) T acc))))

```

### 3.3 Improving run

With `reify∞`, users have access to the remaining stream. But there arises another problem: if we just return a stream, users must look into it and deal with

each of its variants. To keep this information private from users, we require that `run` returns a one-argument function expecting a natural number `k`. When this function is invoked, it passes at most `k` reified results from the remaining stream to the function `f`. Thus, `run` can be redefined as

```
(define (run n fq f)
  (call/fresh 'q
    (lambda (q)
      (run-goal n (fq q) q f))))

(define (run-goal n g q f)
  (reify∞+ n (g empty-S) (reify q) f))

(define (reify∞+ n S∞ T f)
  (let ((result (reify∞ n S∞ T)))
    (let ((_ (f (car result) n)))
      (lambda (k)
        (reify∞+ k (cdr result) T f)))))
```

Now `run` takes an extra argument `f`, which is a function that takes two arguments. The first argument should be a list of reified results, and the second argument is the requested result number. By allowing users to pass a function to `run`, the results can be processed in a more flexible way. In `reify∞+`, we first invoke `reify∞` to get at most `n` reified results and the rest of the stream. We invoke the user-provided function `f` to process the reified results. Then a delayed recursive function is returned. Given a natural number `k`, this function returns at most `k` results from the rest of the stream.

Since we disentangled `run*` from `run`, we don't need to modify the former anymore. Fig. 3 gives the complete definition of `run` and `run*`.

## 4 An Experiment

Now let's use the latest `run` to define a pretty `printer` for miniKanren. It prints each result in a separate line iteratively using the standard `for`.

```
(define (printer res query-n)
  (newline)
  (let ((res-length (length res)))
    (printf "~a/~a Result(s) returned\n"
      res-length query-n)
    (for ((r res))
      (printf "=====\n")
      (printf "Result: ~a\n" r)
      (newline)))
```

Then we can use it to display the results of `appendo` in a user-friendly way.

```

(define (run n  $f_q$  f)
  (call/fresh 'q
    (lambda (q)
      (run-goal n ( $f_q$  q) q f))))

(define (run*  $f_q$ )
  (call/fresh 'q
    (lambda (q)
      (reify* $\infty$  (( $f_q$  q) initial-state) (reify q)))))

```

---

```

(define (call/fresh y  $f_q$ )
  ( $f_q$  (var y)))

(define (run-goal n g q f)
  (reify $\infty$ + n (g empty-S) (reify q) f))

(define (reify $\infty$ + n  $S_\infty$  T f)
  (let ((result (reify $\infty$  n  $S_\infty$  T)))
    (let ((_ (f (car result) n)))
      (lambda (k)
        (reify $\infty$ + k (cdr result) T f)))))

(define (reify $\infty$  n  $S_\infty$  T)
  (reify $\infty$ -acc n  $S_\infty$  T '()))

(define (reify $\infty$ -acc n  $S_\infty$  T acc)
  (cond
    ((or (zero? n) (null?  $S_\infty$ )) (cons acc  $S_\infty$ ))
    ((pair?  $S_\infty$ ) (reify $\infty$ -acc (sub1 n) (cdr  $S_\infty$ ) T
      (cons (T (car  $S_\infty$ )) acc)))
    (else (reify $\infty$ -acc n ( $S_\infty$ ) T acc))))

(define (reify* $\infty$   $S_\infty$  T)
  (cond
    ((null?  $S_\infty$ ) '())
    ((pair?  $S_\infty$ ) (cons (T (car  $S_\infty$ ))
      (reify* $\infty$  (cdr  $S_\infty$ ) T)))
    (else (reify* $\infty$  ( $S_\infty$ ) T))))

(define (var q)
  (vector q))

```

Fig. 3. A functional run, run\*, and call/fresh.



```

> (define next
  (run 4
    (lambda (q)
      (call/fresh 'r
        (lambda (r)
          (call/fresh s
            (lambda (s)
              (conj
                (== `(:,r ,s) q)
                (appendo r s '(a b c d e))))))))
    printer))

4/4 Result(s) returned:
=====
Result : ((a b c) (d e))
=====
Result : ((a b) (c d e))
=====
Result : ((a) (b c d e))
=====
Result : (() (a b c d e))

> (define next (next 3))

2/3 Result(s) returned:
=====
Result : ((a b c d e) ())
=====
Result : ((a b c d) (e))

> (define next (next 10))

0/10 Result(s) returned

```

As can be seen in this example, every time we invoke `next`, we get more results from the rest of the stream, until all results are returned. There is no need to recompute the same result repeatedly.

## 5 Conclusion

We present a functional implementation of `run` and `run*` that gives users access to streams after `run` is returned. The functional implementation makes it easier to port the same technique to different languages. Giving users access to streams avoids potential repetitive computations. Using a function rather than handing out a stream to users directly helps hide the implementation details. Users can

use effects or monads in the function passed to `run`. Disentangling `run*` from `run` indeed simplifies the shared internal functions. The ideas here are based almost exclusively on *TRS2e* [1] and “Nearly Macro-free microKanren” [2].

## References

1. Friedman, D.P., Byrd, W.E., Kiselyov, O., Hemann, J.: The Reasoned Schemer. The MIT Press, Cambridge, Massachusetts, 2nd edn. (2018)
2. Hemann, J., Friedman, D.P.: Nearly Macro-free microKanren. In: Chang, S. (ed.) Trends in Functional Programming, vol. 13868, pp. 72–91. Springer Nature Switzerland, Cham (2023). [https://doi.org/10.1007/978-3-031-38938-2\\_5](https://doi.org/10.1007/978-3-031-38938-2_5)