

# Some criteria for implementations of disjunction and conjunction in microKanren

Jason Hemann

Daniel P. Friedman

jhemann@northeastern.edu

dfried@cs.indiana.edu

## Abstract

We describe how an additional language feature—namely variadic, “rest argument” functions—can lead to a more powerful kernel language. We implement disjunction and conjunction functions that preserve search order and that do not create superfluous closures. The result is a shorter and simpler overall implementation that is more elegant.

## ACM Reference Format:

Jason Hemann and Daniel P. Friedman. 2022. Some criteria for implementations of disjunction and conjunction in microKanren. In *Proceedings of miniKanren Workshop (mKW '22)*. ACM, New York, NY, USA, 6 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 Introduction

More languages than ever before support variable arity functions/methods (aka `varargs` aka `slurpy` methods), including JavaScript, Raku, Java, Python, and Ruby, to name just a few. microKanren [5] is a compact approach to implementing a relational programming language. The microKanren approach has worked out well as a tool for understanding the guts of a relational programming language through studying its implementation. The microKanren reimplementations separates surface syntax macros from function definitions. In doing so, the authors hoped this separation would simultaneously aid future would-be implementers when studying the source code, and also that the functional core would make the language easier to port to other functional hosts. To support both those efforts, they also chose to program in a deliberately small and workaday set of Scheme primitives.

This note shows how an implementer using a host language with variadic (any number of arguments) functions can build a somewhat more powerful kernel language and obviate macros for disjunction and conjunction. Until now

there was a large gap between those microKanren implementations in languages with expressive macro systems and those without; variadic functions permit an intermediate point in the language design space. Even implementers in languages *with* macro systems might choose to replace existing less powerful language primitives by our comparatively short but more powerful variants. Beyond that, this more powerful kernel language obviates several macros in the desugaring layer, and eventually leads us to see merit in modifying the core language’s syntax. This is, of course, subjective.

Beyond being a conceptually simpler foundation for a full Kanren language at the level of *The Reasoned Schemer, Second Edition*, this implementation appears to be more *efficient*. Right-associative conjunction is already conjectured in miniKanren implementation folklore to be an inefficient design choice. We were pleasantly surprised that thinking about the expressivity of primitives and code improvement “horse sense” led us to the same improved design.

In Section 2, we briefly revisit microKanren implementations and illustrate why surface syntax macros had seemed practically mandatory. In Section 3, we implement conjunction and disjunction, and in Section 4 we discuss the reimplementation of the impure operators. We close with a discussion of some performance impacts of these implementation choices, and consider how Kanren language implementers outside of the Scheme family might benefit from these alternatives.

## 2 All Aboard!

We assume the reader is familiar with the miniKanren language as described in *The Reasoned Schemer, Second Edition* and in particular with microKanren implementations. See [minikanren.org](http://minikanren.org) for many implementations across multiple host languages.

The world’s shortest subway system is Haifa’s Carmelit, with only six stations. The system is a line, so its trains travel back and forth. If we wanted to describe the order in which we rode the train from the beginning to the end, we could use the `carmelit-subway` relation in Listing 1. We express this relation using the more compact miniKanren syntax because unfolding this relation into binary conjunctions and disjunctions would be painful. We certainly *could* write it out by hand—in fact, we could write it out many

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

mKW '22, September 15, 2022, Ljubljana, Slovenia

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

```
(defrel (carmelit-subway a b c d e f)
  (conde
    ((= a 'carmel-center)
     (= b 'golomb)
     (= c 'masada)
     (= d 'haneviim)
     (= e 'hadar-city-hall)
     (= f 'downtown))
    ((= a 'downtown)
     (= b 'hadar-city-hall)
     (= c 'haneviim)
     (= d 'masada)
     (= e 'golomb)
     (= f 'carmel-center))))
```

**Listing 1.** A miniKanren version of the Carmelit subway

```
(define-syntax conde
  (syntax-rules ()
    ((conde (g ...) ...)
     (disj (conj g ...) ...))))

(define-syntax disj
  (syntax-rules ()
    ((disj) #u)
    ((disj g) g)
    ((disj g0 g ...) (disj2 g0 (disj g ...))))

(define ((disj2 g1 g2) s)
  (append∞ (g1 s) (g2 s)))

(define-syntax conj
  (syntax-rules ()
    ((conj) #s)
    ((conj g) g)
    ((conj g0 g ...) (conj2 g0 (conj g ...))))

(define ((conj2 g1 g2) s)
  (append-map∞ g2 (g1 s)))
```

**Listing 2.** Macro based implementations of disj and conj

ways. We could nest those conjunctions to the left, or to the right, or try and reduce the indentation by trying to keep them somewhat balanced: the program itself does not seem to obviously encourage one particular choice.

Listing 2 shows typical macro-based implementations of conde and the underlying conjunction and disjunction operations. Here, #s and #u represent primitive goals that unconditionally succeed and fail, respectively.

```
(define ((disj g . gs) s)
  (D (g s) gs s))

(define (D s∞ gs s)
  (cond
    ((null? gs) s∞)
    (else
     (append∞ s∞
      (D ((car gs) s) (cdr gs) s)))))
```

**Listing 3.** Eventual redefinition of disj

```
(define ((conj g . gs) s)
  (C gs (g s)))

(define (C gs s∞)
  (cond
    ((null? gs) s∞)
    (else
     (C (cdr gs)
      (append-map∞ (car gs) s∞)))))
```

**Listing 4.** Eventual redefinition of conj

### 3 disj and conj logical goal constructors

The disj and conj macros of Listing 2 are not *quite* defined as simple recursions over their binary functional primitives. The definition of zero-way conjunction (disjunction) is independent of the unfolding of conj<sub>2</sub> (disj<sub>2</sub>). In a sense the disj and conj macros confuse and entangle primitive success and failure goals with those recursive unfoldings in terms of the binary operators.

Those zero-way logical operation base cases don't add much. The programmer who tries to write an elegant, efficient solution to a pure relational programming task would not encounter these additional base cases in conde expressions. The programmer knows statically how such goals should behave, so there is no benefit to executing them. A conjunction of no goals would simply succeed, and a disjunction of no goals would simply fail. Truthfully, only the impure conda operator seems to require these additional base cases (discussed further in Section 4). They are for all intents and purposes superfluous.

Here, we want to implement disjunction and conjunction over one-or-more goals as functions. These implementations will rely on variadic function syntax, but should not require apply or rely on the binary disj<sub>2</sub> and conj<sub>2</sub>. Further, they should not build any extraneous closures: unnecessarily building closures at runtime is always a bad idea.

Listings 3 and 4 show our new implementations. Mandating one-or-more arguments lets us re-implement these operators as shallow wrappers over simple folds. The first step in each is merely to remove the rest argument gs and act as

if there were no need for a rest argument. That is, all of our focus will be on the *list*, *gs*. Unlike *D*, the function *C* does not take in the state *s*; the help procedure does not need the state for conjunction. In each recursive call, we accumulate by mapping (using the special delaying implementation of *append-map* for miniKanren streams) the next goal in the list. This left-fold implementation of conjunction therefore left-associates the conjuncts.

### 3.1 Semantic equivalence

Equipped with variadic functions, a developer might discover these definitions as follows. We start from the definition of the *disj* macro in Listing 5. At the cost of an *apply*, we can build the corresponding explicitly recursive *disj* function.

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

(define (disj g . gs)
  (cond
    ((null? gs) g)
    (else (disj2 g (apply disj gs)))))
```

**Listing 5.** Deriving *disj* function from macro

Since *disj* produces and consumes goals, we can  $\eta$  expand the definition in Listing 5 by a curried parameter *s*. We then split *disj* into two mutually-recursive procedures, to build the variant in Listing 6.

```
(define ((disj g . gs) s)
  (D g gs s))

(define (D g gs s)
  (cond
    ((null? gs) (g s))
    (else ((disj2 g (apply disj gs)) s))))
```

**Listing 6.** An  $\eta$  expanded and split definition of *disj*

We can replace the call to *disj<sub>2</sub>* in Listing 6 by its definition in terms of *append<sub>∞</sub>* and perform a trivial  $\beta$ -reduction. The explicit *s* argument suggests removing the call to *apply* and making *D* recursive. The result is the version of *D* in Listing 7. The definition of *disj* remains unchanged from Listing 6. We combine *g* and *s* in each clause; this suggests constructing that stream in *disj* and passing it along. Making this final change results in the definition in Listing 3.

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

**Listing 7.** Derivation of *disj* function definition

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

(define (conj g . gs)
  (cond
    ((null? gs) g)
    (else
     (apply conj
              (cons (conj2 g (car gs)) (cdr gs))))))
```

**Listing 8.** *conj<sub>2</sub>*-based *conj* function and macro

We can derive the definition of *conj* from Listing 4 via a similar process. Starting with the variadic function based on the macro in Listing 8, we first  $\eta$ -expand and split the definition.

```
(define ((conj g . gs) s)
  (C g gs s))

(define (C g gs s)
  (cond
    ((null? gs) (g s))
    (else
     ((apply conj
              (cons (conj2 g (car gs)) (cdr gs))
                   s))))))
```

**Listing 9.** Derivation of split *conj* function definition

We next substitute for the definitions of *conj* and *conj<sub>2</sub>*.

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

**Listing 10.** Replacing *apply* in *C* function definition

Finally, since *C* only needs *s* to *build* the stream, we can assemble the stream on the way in—instead of passing in *g* and *s* separately, we pass in their combination as a stream. The function is tail recursive, we can change the signature in the one and only external call and the recursive call. We show the result in Listing 4.

Both the variadic function based and the macro based versions of Listing 8 use a left fold over the goals, whereas the versions of *disj* use a right fold. This is not an accident. It is miniKanren folklore that left associating conjunctions tends to improve miniKanren performance. A full, thorough, explanation of everything known about this behavior is outside the scope of our work here, but see discussions and implementation of [7]. The matter seems so significant that the community might consider reclassifying left-associative conjunction as a matter of correctness rather than an optimization, as in “tail call optimization” vs. “Properly Implemented Tail Call Handling” [3]. However, we have generally resorted to small step visualizations of the search tree to demonstrate why that might be. The authors believe it is worth considering if we can make an equally compelling argument for this preference through equational reasoning. Listing 11 shows a right-fold variant of *conj*. The choice to fold left becomes a little more obvious after we  $\eta$ -expand, unfold to a recursive help function, substitute in the definition of *conj<sub>2</sub>*, and  $\beta$ -reduce.

```
(define (conj g . gs)
  (cond
    ((null? gs) g)
    (else (conj2 g (apply conj gs)))))
```

**Listing 11.** A right-fold variant of *conj*

```
(define ((conj g . gs) s)
  (C gs (g s)))

(define (C g gs s)
  (cond
    ((null? gs) (g s))
    (else (append-map-∞ (apply C gs) (g s)))))
```

Here, we cannot (easily) replace the *apply* call by a recursive call to *C*, because we are still waiting for an *s*. We can only abstract over *s* and wait.

```
(define (C gs s∞)
  (cond
    ((null? gs) s∞)
    (else
     (append-map-∞
      (λ (s) (C (cdr gs) ((car gs) s)))
      s∞)))))
```

```
(defrel (carmelit-subway a b c d e f)
  (disj
    (conj (== a 'carmel-center)
          (== b 'golomb)
          (== c 'masada)
          (== d 'haneviim)
          (== e 'hadar-city-hall)
          (== f 'downtown))
    (conj (== a 'downtown)
          (== b 'hadar-city-hall)
          (== c 'haneviim)
          (== d 'masada)
          (== e 'golomb)
          (== f 'carmel-center)))))
```

**Listing 12.** A new Carmelit subway without *conde*

Since we know that any call to *append-map<sub>∞</sub>* we construct will always yield a result, the version in Listing 4 is tail recursive. The equivalent right-fold implementation needs to construct a closure for every recursive call. If we want to implement a variadic version that does not rely on a primitive *conj<sub>2</sub>* and does not resort to *apply*, we have the two aforementioned choices. Basic programming horse sense suggests the variant from Listing 4.

Though this note mainly concerns the choice to implement surface language behavior as functions, it may also point to these as more natural user-level primitives than *conde*. An implementation could choose to forego *conde* and provide just those underlying logical primitives *disj* and *conj* to the user, as in the new definition of Carmelit in Listing 12. Moreover, reintroducing the nullary case only requires a simple wrapper macro, or otherwise, a run-time *null?* check for a very slight cost.

## 4 Tidying up the Impure Operators

The operators *conda* and *conde* look superficially similar, syntactically. Semantically though, *conda*’s nested “if-then-else” behavior is quite different, and implementing the desired behavior for *conda* from existing pieces points to some strangeness. The definition of *conda* (see Listing 13) requires one or more conjuncts per clause and one or more clauses. The consequent of each *conda* clause is the one and only place in the whole language implementation that permits nullary conjunctions of goals. This soft-cut operator seems to force both nullary conjunction and those primitive goals *#s* and *#u* into the language.

Some programmers would be perfectly satisfied just using *ifte* directly. But just as the standard forked *if* begat McCarthy’s *if* notation and *cond*, we can suspect that a programmer would eventually feel the need for a nested implementation. There are other alternative implementation choices one could consider.

```

(define-syntax conda
  (syntax-rules ()
    ((conda (g0 g ...) (conj g0 g ...))
     ((conda (g0 g ...) ln ...)
      (ifte g0 (conj g ...) (conda ln ...)))))

(define ((ifte g1 g2 g3) s)
  (let loop ((s∞ (g1 s)))
    (cond
      ((null? s∞) (g3 s))
      ((pair? s∞)
       (append-map g2 s∞))
      (else (lambda ()
               (loop (s∞)))))))

```

**Listing 13.** A typical implementation of conda

```

(define ((conda q a . q-and-a*) s)
  (A (q s) a q-and-a* s))

(define (A s∞ a q-and-a* s)
  (cond
    ((null? s∞)
     (cond
       ((null? (cdr q-and-a*)) ((car q-and-a*) s))
       (else (A ((car q-and-a*) s)
                 (cadr q-and-a*)
                 (caddr q-and-a*)
                 s))))
    ((pair? s∞) (append-map∞ a s∞))
    (else (lambda () (A (s∞) a q-and-a* s)))))

```

**Listing 14.** A functional conda implementation

1. Syntactically mandate that all clauses *except the final default clause* contain at least two goals.
2. Introduce a special clause of the conda macro specifically for “if then” clauses with a single goal.
3. Unconditionally add an a *#s* goal to each clause during macro expansion.

Each of these choices can, implicitly or explicitly, force additional unneeded executions of unwanted goals. Working with variable arity function syntax also suggests a more elegant solution for conda.

The implementation in Listing 14 includes the delay-and-restart behavior of ifte together with conda’s logical cascade. The  $s_{\infty}$  can be either empty, non-empty, or a function of no arguments. In the last case, we invoke  $s_{\infty}$ . Rather than building a largely redundant implementation of conda, we expose the higher-order goal once to the user. The definition of once in Listing 15 is taken directly from [4]. The programmer can simulate conda by wrapping once around every test goal.

```

(define (once g)
  (lambda (s)
    (let loop ((s-inf (g s)))
      (cond
        ((null? s-inf) '())
        ((pair? s-inf)
         (cons (car s-inf) '()))
        (else (lambda ()
                 (loop (s-inf))))))))

```

**Listing 15.** The once function

## 5 A functional run

When you do a `call/fresh` there must be some mechanism to find the next fresh variable. For example we use the natural numbers and `add1` for the set of variables and the operation to produce a new variable we merely need to add 1 to the natural number representing the most recent variable. For another example, we could represent each variable using a unique memory location and the operation to produce a new variable requires introducing an unused memory location. One of them of course requires memory allocation and the other does not.

Depending on the implementation of variables, you may also need additional functions to support your category of variables. If variables are not from an indexed set, you may also need an operation to specifically (re)construct the first element of the set, or otherwise store that value for later re-use.

For the reader interested in removing macros, we note that we can also implement `run` without using macros [5]. It is relatively straightforward to rewrite the definitions of `run` and `run*` so that the argument to `call/initial-state` is itself a goal that introduces the first logic variable `q`.

## 6 Conclusion

This note shows that at the cost of one additional feature—namely, variadic functions—implementers provide a somewhat more powerful core language and significantly reduce the need for macros in implementing a language as expressive as the full “microKanren + macros” approach provides. Our desugaring macros would have worked whether we left or right associated. Removing a degree of freedom in the implementation, and forcing ourselves to program the solution functionally, led us to what seems like the right solution.

The result is closer to the design of Prolog, where the user represents conjunction of goals in the body of a clause with a comma and disjunction, either implicitly in listing various clauses or explicitly with a semicolon.

We assume it is agreed that our definitions of `disj` and `conj` themselves are sufficiently high-level operators for a surface language and that the zero-element base cases are

at best unnecessary and likely undesirable; given the opportunity to define a surface language and its desugaring, we really shouldn't tempt the programmer by making undesirable programs representable when we can avoid it.

Scheme's polyvariadic function syntax (that is to say "at least  $k$  arguments", for some positive integer  $k$ ) ensures at a host-language level that the microKanren programmer provides at least one parameter to `conj` and `disj`. Embeddings in languages with variadic but without polyvariadic function syntax would have to implement that check.

We have not fully obviated the use of macros. While neither `run` nor `run*` have to be macros, for now we still want to implement `defrel` and `fresh` with macros. The former to prevent exposing the implementation of streams and delays, and the latter to avoid the inefficiency and clumsiness of introducing variables one at a time.

From time to time we find that the usual miniKanren implementation is itself lower-level than we would like to program with relations. Early microKanren implementations restrict themselves to syntax-rules macros. Some programmers use macros to extend the language further as with `matche` [6]. Some constructions over miniKanren, such as `minikanren-ee` [1], may rely on more expressive macro systems like `syntax-parse` [2].

We would still like to know: are our design goals here also causally related to good miniKanren performance? Regardless, we hope this document helps narrow the gap between implementations in functional host languages with and without macro systems and helps implementers build more elegant, expressive and efficient Kanrens in their chosen host languages.

## Acknowledgments

Thanks also to Ken Shan and Jeremy Siek, for helpful discussions and debates during design decision deliberations. Thanks also to Greg Rosenblatt and Michael Ballantyne for their insights and suggestions.

## References

- [1] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proceedings of the ACM on Programming Languages*, 4, OOPSLA, 1–29.
- [2] Ryan Culpepper. 2012. Fortifying macros. *Journal of functional programming*, 22, 4-5, 439–476.
- [3] Matthias Felleisen. 2014. Re: question about tail recursion. (2014). <https://lists.racket-lang.org/users/archive/2014-August/063844.html>.
- [4] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*. The MIT Press, (Mar. 2018). ISBN: 0-262-53551-3. [mitpress.mit.edu/books/reasoned-schemer-0](http://mitpress.mit.edu/books/reasoned-schemer-0).
- [5] Jason Hemann and Daniel P. Friedman. 2013. Mkanren: a minimal functional core for relational programming. In *Scheme 13*. <http://scheme-workshop.org/2013/papers/HemannMuKanren2013.pdf>.
- [6] Andrew W Keep, Michael D Adams, Lindsey Kuper, William E Byrd, and Daniel P Friedman. 2009. A pattern matcher for minikanren or how to get into trouble with cps macros. *Technical Report CPSLO-CSC-09-03*, 37.
- [7] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-order minikanren representation: great for tooling and search. *Proceedings of the miniKanren Workshop*, 16.