# Nearly Macro-free microKanren

Jason Hemann
Daniel P. Friedman
jason.hemann@shu.edu
dfried@cs.indiana.edu

## Abstract

This paper describes changes to the microKanren implementation that make it more practical to use in a host language without macros. With the help of some modest runtime features common to most languages, we show how an implementer lacking macros can come closer to the expressive power that macros usually provide—with varying degrees of success. The result is a still functional microKanren that invites slightly shorter programs, and is relevant even to implementers that enjoy macro support. For those without it, we address some practical concerns that necessarily occur without macros so they can better weigh their options.

***CCS Concepts:*** • **Software and its engineering → Constraint and logic languages**.

***Keywords:*** logic programming, miniKanren, DSLs, embedding, macros

**ACM Reference Format:**
Jason Hemann and Daniel P. Friedman. 2023. Nearly Macro-free microKanren. In *Proceedings of Symposium on Trends in Functional Programming (TFP '23).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

The authors designed microKanren [5] as a compact relational programming language kernel to undergird a miniKanren implementation. Macros are used to implement the surrounding higher-level miniKanren operators and surface syntax. microKanren is often used as a tool for understanding the guts of a relational programming language through studying its implementation. By re-implementing miniKanren as separate surface syntax macros over a purely-function microKanren kernel, the authors hoped this separation would simultaneously aid 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.

The sum of those implementation restrictions, however, necessitates some awkward compromises in places including binary logical operators, one at a time local variable introduction, and leaks in the streams abstractions. These made the surface syntax macros seem practically mandatory, and fell short enough that we compromised on a purely functional kernel in a pedagogical exposition [4]. It also divided host languages into the macro language "haves" and macro-less "have nots". Here, we bridge some of that divide by re-implementing parts of the kernel with some modest runtime features common to most languages.

In this paper we:

- show how to functionally implement more general logical operators, cleanly obviating some of the surface macros
- provide guidance and utilitarian solutions for eliminating all other macros in *The Reasoned Schemer, 2nd Ed* [4] (*TRS2e*) core language implementation
- survey the purely functional design space, and weigh the trade offs and real-world concerns of completely eliminating those remaining macros.

This exercise resulted in some higher-level (variadic rather than just binary) operators, a more succinct kernel language, and made possible some performance improvement. Approximately half of the changes are applicable to any microKanren implementation, and the other half are necessarily awkward yet practical strategies for platforms lacking macro support. The source code for our re-implementation and experimental results is available at https://github.com/jasonhemann/tfp-2023/.

In Section 2, we illustrate by example what made surface syntax macros feel practically mandatory. In Section 3, we implement conjunction and disjunction, and in Section 4 we discuss the re-implementation of the impure operators. We discuss the remaining macros in Section 5. We close some outstanding questions on performance impacts of these implementation choices, and consider how Kanren language implementers outside of the Scheme family might benefit from these alternatives.


*TFP '23, January 13-15, 2023, Boston, Massachusetts*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

## 2 All Aboard!

We assume the reader is familiar with the miniKanren implementation of *The Reasoned Schemer, 2nd Ed* (*TRS2e*). Although based on microKanren, this implementation makes some concessions to efficiency and safety and uses a few macros in the language kernel itself. In addition to that implementation, in this paper we make occasional references to earlier iterations such as Hemann et al. [6], an expanded archival version of the 2013 paper [5].

Haifa's Carmelit is the world's shortest subway system with a line of just six stations; an example sufficiently small that modeling it should be painless. But in microKanren, to model the order a passenger visit the stops riding the subway end to end requires 11 logical operator nodes, because microKanren only provides *binary* conjunctions and disjunctions. (Listing 10 contains this paper's alternative solution, requiring just three.) For a logic programming language, solely binary logical operators is too low level. To our eyes, this makes the superficial syntax macros are practically mandatory, and host languages without a macro system are out of luck.

The microKanren language doesn't offer the programmer much guidance in using that fine-grained control. For a series of $n$ goals, the programmer can associate them to the left, to the right, or some mixtures of the two. The syntax does not obviously encourage any one choice, and subtle changes in program structure can have profound effects on performance.

The *TRS2e* microKanren's soft-cut operator, `ifte`, is similarly low level. It permits a single test, a single consequent, and a single alternative. To build an if-then-else cascade, a microKanren programmer without the `conda` surface macro would need to unroll that cascade by hand.

The core *TRS2e* language implementation relies on macros `fresh`, `defrel`, and `run` to introduce new logic variables, globally define relations, and execute queries. Earlier implementations of those same behaviors via pure functional shallow embeddings, without macros, had some harsh consequences. We will revisit those earlier implementations and their trade-offs, survey the landscape of available choices, and suggest performant compromises for those truly without macros, thus increasing microKanren's *practical* portability.

## 3 `disj` and `conj` logical goal constructors

We want to implement disjunction and conjunction over arbitrary quantities of goals, as functions. These implementations should subsume the binary `disj2` and `conj2` and should not use `apply`. Further, they should not build any extraneous closures: unnecessarily building closures at runtime is always a bad idea. This re-implementation requires a host that supports variable arity functions, a widely available feature included in such languages as JavaScript, Ruby, Java, and Python. These languages do not generally support macros and hence are beneficiaries of this paper's contributions.

```
(define ((disj . gs) s)
  (cond
    ((null? gs) (list))
    (else (D ((car gs) s) (cdr gs) s))))

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

**Listing 1.** Eventual redefinition of `disj`

```
(define ((conj . gs) s)
  (cond
    ((null? gs) (list s))
    (else (C (cdr gs) ((car gs) s)))))

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

**Listing 2.** Eventual redefinition of `conj`

Listings 1 and 2 show our new implementations. We re-implement these operators as shallow wrappers over simple folds. In each, the first steps are to dispense with the trivial case, and then to call a recursive help function that makes no use of variadic functions. That is, all of our focus will be on the recurring over 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 Kanren-language streams) the next goal in the list. This left-fold implementation of conjunction therefore left-associates the conjuncts.

### 3.1 Semantic equivalence

A developer might derive these definitions as follows. We start with the definition of a recursive `disj` macro like one would define as surface syntax over the microKanren `disj2`. 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 3.** Deriving `disj` function from macro

Since `disj` produces and consumes goals, we can η expand the definition in Listing 3 by a curried parameter `s`. We then split `disj` into two mutually-recursive procedures, to build the variant in Listing 4.

```
(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 4.** An η expanded and split definition of `disj`

We can replace the call to `disj₂` in Listing 4 by its definition in terms of `append∞` and perform a trivial β-reduction. The explicit `s` argument suggests removing the call to `apply` and making `D` recursive. The result is the version of `D` in Listing 5. The definition of `disj` remains unchanged from Listing 4.

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

**Listing 5.** Derivation of `disj` function definition

In both clauses of `D` we combine `g` and `s`, this suggests constructing that stream in `disj` and passing it along. Adding the trivial base case to that `disj` yields the definition in Listing 1.

We can derive the definition of `conj` from Listing 2 via a similar process. Starting with the variadic function based on the macro in Listing 6, we first η-expand and split the definition.
We next substitute for the definitions of `conj` and `conj₂`.

```
(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 6.** `conj₂`-based `conj` function and macro

```
(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 7.** Derivation of split `conj` function definition

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

**Listing 8.** 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 2.

Both the functional and the macro based versions of Listing 6 use a left fold over the goals, whereas the versions of `disj` use a right fold. This is not an accident. Folklore suggests left associating conjunctions tends to improve the performance of miniKanren's interleaving search. The authors know of no thorough algorithmic proof of such claims, but see for instance discussions and implementation in [9] for some of the related work so far. We have generally, however, resorted to small step visualizations of the search tree

to explain the performance impact. The authors believe it is worth considering if we can make an equally compelling argument for this preference through equational reasoning and comparing the implementations of functions.

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

**Listing 9.** A right-fold variant of `conj`

Listing 9 shows a right-fold variant of `conj`. The choice to fold left becomes a little more obvious after we η-expand, unfold to a recursive help function, substitute in the definition of $\texttt{conj}_2$, and β-reduce.

```
(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. Since we know that any call to `append-map∞` we construct will always yield a result, the version in Listing 2 is tail recursive. The equivalent right-fold implementation needs either to construct a closure for every recursive call, or otherwise demands a help function mutually recursive with `C`.

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

If we want to implement a variadic version that does not rely on a primitive `conj2` and does not resort to `apply`, we have the two aforementioned choices. Basic programming horse sense suggests the more elegant variant from Listing 2.

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 10. Moreover, removing the nullary case simplifies the wrapper function, and removes the very slight cost of a runtime `null?` check. It also suggests rewriting the second

```
(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 10.** A new Carmelit subway without `conde`

```
(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 11.** A typical implementation of `conda`

pattern in `conda` to $\texttt{(conda (}g_0\ \texttt{g g* ...) ln ...)}$, since the consequents of `conda` clauses rely structurally on permitting nullary conjunctions of goals.

This implementation of `carmelit-subway` uses our `disj` and `conj` functions, a far cry better than the 11 binary logical operator nodes the programmer would need to write in a language without macros.

## 4 Tidying up the Impure Operators

The `conda` operators provides nested "if-then-else" behavior. The definition of `conda` (see Listing 11) requires one or more conjuncts per clause and one or more clauses.

Some programmers would be perfectly satisfied just using `ifte` directly. But just as the standard forked `if` begat McCarthy's `if` notation and `cond`, a programmer may eventually feel the need for a nested implementation. Here are other (alternative) implementation choices one could consider.

```
(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*)
                (cddr q-and-a*)
                s))))
    ((pair? s∞) (append-map∞ a s∞))
    (else (lambda () (A (s∞) a q-and-a* s)))))
```

**Listing 12.** A functional `conda` implementation

```
(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 13.** The `once` function

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 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 12 includes the delay-and-restart behavior of `ifte` together with `conda`'s logical cascade. The `s∞` can be either empty, non-empty, or a function of no arguments. In the last case, we invoke `s∞`. Rather than building a largely redundant implementation of `condu`, we expose the higher-order goal `once` to the user. The definition of `once` in Listing 13 is taken directly from [4]. The programmer can simulate `condu` by wrapping `once` around every test goal.

```
(define (subtleo x)
  (Zzz
    (disj
      (subtleo x)
      (== x 'cat))))
```

**Listing 14.** Omitting the delay is a subtle bug

## 5 Remainders and Practicalities

We have not fully obviated the use of macros. In this section we collect together some workarounds to obviate macros in the rest of the implementation. Some of these come with significant drawbacks. With these, however, a programmer in even a pedestrian functional language should be able to directly translate the implementation and our test programs.

***`define`.*** The microKanren programmer can just use their host language's `define` feature to construct relations as host-language functions, and manually introduce the delays in relations. This may be a larger concession than it looks, since it exposes the delay and interleave mechanism to the user, and both correct interleaving and even the termination of relation *definitions* rely on a whole-program correctness property of relation definitions having a delay. Listing 14 relies on a help function `Zzz` to introduce delays, akin to some earlier implementations [5]. Another downside worth mentioning is the programmer must now take extra care not to provide multiple goals to define. The `define` form will treat all but the last expression as statements and silently drop them, rather than conjoin them as in `defrel`. That small *gotcha* can be subtle but significant drawback.

***`fresh`.*** In any implementation there must be some mechanism to produce the next fresh variable. For example, we treat the natural numbers as an indexed set of variables, and we thread the current index through the computation. We use `add1` to get the next index; to go from index to variable is the identity function. For another example, we could represent each variable using a unique memory location, sokuza-kanren [8] style, and the operation to produce a new variable requires introducing an unused memory location. Depending on the implementation of variables, you may also need additional functions to support your implementation of variables. If variables are not from an indexed set, you may also need an operation to (re)construct specifically the first element of the set, or otherwise store that value for later re-use.

Of course, one of these approaches requires memory allocation and external global state, while the other does not. Furthermore, the latter approach models logic variables as coming from a single global pool rather than reusing them separately across each disjunct, and so requires some global store and strictly more logic variables overall.

```
(call/initial-state 1
  (let ((q (var 'q)))
    (conj
      (let ((x (var 'x)))
        (== q x))
      (reify q))))
```

**Listing 15.** Queries as expressed with global-state variables

With this latter approach, however, we can expose `var` directly to the programmer and the programmer can use `let` bindings to introduce several logic variables simultaneously.

***run.*** We note that we can also implement `run` without using macros. Using the purely-functional implementation of logic variables, the definitions of `run` and `run*` easily translate to functions like `call/initial-state` [5]. The query is itself expressed as a goal that introduces the first logic variable `q`. The pointer-based logic variable approach forces the programmer to explicitly invoke `reify` as though it were a goal as the last step of executing the query, as in Listing 15.

## 6 Future Work

This note shows how to provide a somewhat more concise core language that significantly reduces the need for macros. The result almost rivals the expressivity of the full "microKanren + macros" approach. Variadic functions make this implementation much more convenient for the end programmer, and Scheme's polyvariadic function syntax ensures at a host-language level that the microKanren programmer provides at least one parameter to `conj` and `disj`.

The old desugaring macros do not seem to suggest how to associate the calls to the binary primitives—both left and right look equally nice. Forcing ourselves to program the solution functionally, and the restrictions we placed on ourselves in this reimplementation, removed a degree of implementation freedom and 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.

Techniques for implementing `defrel`, `fresh` and `run` (and `run*`) without macros come with serious drawbacks. These include exposing the implementation of streams and delays, and the inefficiency and clumsiness of introducing variables one at a time, or the need to reason with global state.

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` [7]. 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 if our desiderata here are *causally* related to good miniKanren performance. Can we reason at the implementation level and peer through to the implications for performance? If left associating `conj` is indeed uniformly a dramatic improvement, 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]. 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 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. We would also like to thank our anonymous reviewers for their insightful contributions.

## References

[1] Michael Ballantyne, Alexis King, and Matthias Felleisen. "Macros for domain-specific languages." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[2] Ryan Culpepper. "Fortifying macros." In: *Journal of functional programming* 22.4-5 (2012), pp. 439–476.

[3] Matthias Felleisen. *Re: Question about tail recursion.* 2014. URL: https://lists.racket-lang.org/users/archive/2014-August/063844.html.

[4] Daniel P. Friedman et al. *The Reasoned Schemer, Second Edition.* The MIT Press, Mar. 2018. ISBN: 0-262-53551-3. URL: mitpress.mit.edu/books/reasoned-schemer-0.

[5] Jason Hemann and Daniel P. Friedman. "μkanren: A Minimal Functional Core for Relational Programming." In: *Scheme 13.* 2013. URL: http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.

[6] Jason Hemann et al. "A Small Embedding of Logic Programming with a Simple Complete Search." In: *Proceedings of DLS '16*. ACM, 2016. URL: http://dx.doi.org/10.1145/2989225.2989230.

[7] Andrew W Keep et al. "A pattern matcher for miniKanren or How to get into trouble with CPS macros." In: *Technical Report CPSLO-CSC-09-03* (2009), p. 37.

[8] Oleg Kiselyov. *The taste of logic programming*. 2006. URL: http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[9] Gregory Rosenblatt et al. "First-order miniKanren representation: Great for tooling and search." In: *Proceedings of the miniKanren Workshop* (2019), p. 16.