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 to broaden the kinds of platforms it may be embedded in. We lift microKanren's macro-support requirement for host platforms, trading it for modest runtime features common to most languages. The resulting implementation is smaller, simpler, and 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**; **Automatic programming**.

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

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 reimplementation separates surface syntax macros from function definitions. In doing so, 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.

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

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

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. This note shows how an implementer in a host language with variadic (any number of arguments) functions can build a somewhat more powerful kernel language and obviate some macros, including those 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. Implementers in languages *with* macro systems may even choose to replace existing less powerful language primitives by our comparatively short but more powerful variants.

Beyond being a conceptually simpler foundation for a full Kanren language at the level of *The Reasoned Schemer, Second Edition*, this implementation may even be more efficient. miniKanren folklore conjectures that right-associative conjunction is an inefficient design choice for implementing the miniKanren search strategy. Thinking about the expressivity of these primitives and code improvement "horse sense" led us to left-associate conjuncts.

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 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.

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 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

```
(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`

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 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 `conj₂` (`disj₂`). 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.

To reduce the reliance on macros, we want to implement disjunction and conjunction over one-or-more goals as functions. These implementations should not require `apply` or rely on the binary `disj₂` and `conj₂`. 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. Variadic functions make these operators more ergonomic, and 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. 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

A developer might derive 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 η 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 η expanded and split definition of `disj`

We can replace the call to `disj₂` in Listing 6 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 7. The definition of `disj` remains unchanged from Listing 6.

```
(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

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-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. `conj₂`-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 η-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 `conj₂`.

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.

```
(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

Both the functional 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. 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 [8] for some of the related work so far. However, we have generally 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 11. A right-fold variant of `conj`

Listing 11 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 `conj₂`, 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 4 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∞))))
```

```
(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`

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 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 exposes 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.

```
(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*)
                (cddr 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

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.

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 14 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 15 is taken directly from [4]. The programmer can simulate `condu` 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

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

Listing 16. 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 16 relies on a help function `Zzz` to introduce delays, akin to some earlier implementations [5].

`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

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

Listing 17. Queries as expressed with global-state variables

the identity function. For another example, we could represent each variable using a unique memory location, sokuzakanren [7] 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.

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 17.

6 Future Work

This note shows how to provide a somewhat more powerful 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` [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 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] 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.

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

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