Towards criteria for implementations of variable arity conjunction and disjunction in microKanren

Anonymous Author(s)

Abstract

We describe here how, at the cost of an additional language feature—namely variadic and "rest argument" functions—we can implement a slightly higher-level kernel language. This implementation includes conjunction and disjunction for microKanren that do not create superfluous closures. The result is a shorter and simpler overall implementation that is more elegant.

ACM Reference Format:
Anonymous Author(s). 2018. Towards criteria for implementations of variable arity conjunction and disjunction in microKanren. In *Proceedings of miniKanren Workshop (mKW '22).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

microKanren [2] 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. A design goal for this reimplementation of miniKanren has been to drop some of the complexity associated with mixing macros and function definitions, especially for those implementers trying to avoid macros. This goal has not been met, but we show how the macros for conjunction and disjunction can be avoided, leaving an underlying pure functional core. In doing so, the authors hope this separation will 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 host languages. To support both those efforts, the authors have chosen to program these foundational parts of (relational) logic programming using a new kind of microKanren in a deliberately small and workaday set of Scheme primitives.

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

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

Until now microKanren implementations have divided the programming languages of the world more or less into the expressive macro system "haves" and the macro system-less "have nots". This note shows how an implementer using a functional language with variadic (any number of arguments) functions but without `apply` or a macro system can still achieve an intermediate point.

Even implementers using and programmers studying implementations written in languages *with* sufficiently expressive macro systems can benefit from these improvements. There are benefits to using a limited set of features in a language. Based on an evaluation of minikanren.org, earlier microKanren implementers measured their results by the number of lines in the core implementation, the number of additional lines for the surface syntax, and the limited features they required from the host. So replacing less powerful language primitives by comparatively short but more powerful variants is a win. 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*. The source of the original inefficiency—right-associative conjunction—is already known miniKanren implementation folklore. We were pleasantly surprised that thinking about the expressivity of primitives and code improvement "horse sense" led us to this same idea. That very horse sense might lead one to suggest these improved variants should actually be the external programming constructs. This, too, is very speculative.

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 alternative implementations.

```
(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 implementation of the Carmelit subway.

2 All Aboard!

We assume the reader is familiar with the miniKanren language and in particular with microKanren implementations. We refer the reader to some elementary resources [1, 2, 3] for further explanation, or to minikanren.org for a slew of implementations across many 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 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.

We defer further discussion on this point to Section 3. Here we intend just to illustrate that those programming in a microKanren in a macro-less host language, for even fairly trivial tasks, could get bothered by low-level details such as binary conjunction and disjunction, with all the difficulties of a shallow embedding. E.g., a small programming mistake, such as incorrectly nesting a conjunct somewhere in a long conjunction chain, can cause relatively obscure errors and reading those error messages forces the programmer out of the microKanren-DSL thinking, and back into the host programming language's debug model.

3 `conj` and `disj` logical goal constructors

Programmers rightly expect better than having to chain their logic together like building circuitry from logic gates. Such higher-level logical operations may or may not be present in the core language, but if not, the programmer will reasonably expect some syntax sugar to provide them. Witness Scheme's `and` and `or` macros as examples. [4] Like in a typical miniKanren implementation, the `conde` macro is a shallow syntactic wrapper for disjuctive normal form. Listing 2 shows our new implementation.

```
(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∞)))))

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

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

(define-syntax conde
  (syntax-rules ()
    ((conde (g g₁ ...) ...)
     (disj (conj g g₁ ...) ...))))
```

Listing 2. Re-implementations of `conj` and `disj`.

Mandating one-or-more arguments lets us re-implement these operators as shallow wrappers over 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`. These implementations rely on variadic functions, but do not require `apply`. They no longer rely on binary `conj₂` and `disj₂`, and they do not build any extraneous closures. Unnecessarily building closures at runtime is always a bad idea.

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.

Compare these to typical macro-based microKanren implementations of the underlying conjunction and disjunction operations in Listing 3. Here, `#s` and `#f` represent primitive goals that unconditionally succeed and fail, respectively.

```
(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)))

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

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

Listing 3. Macro based implementations of `conj` and `disj`.

Those `conj` and `disj` macros are not *quite* defined as simple recursions over their binary functional primitives. The definition of 0-way conjunction (disjunction) is independent of the unfolding of `conj₂` (`disj₂`). In a sense the `conj` and `disj` 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 his `conde` expressions. The programmer knows statically how these 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).

3.1 Implementation Correctness

A developer equipped with variadic functions might discover these definitions as follows. We start with the definition of `disj` from Listing 2. At the cost of an `apply`, we can define this as a single explicitly recursive function.

```
(define ((disj g . gs) s)
  (cond
    ((null? gs) (g s))
    (else
     (append∞ (g s)
       ((apply disj gs) s)))))
```

We inverse β-substitute in the recursive case, to draw out a similarity between the two clauses.

We can replace the call to `append∞` by the definition of `disj₂`.

```
(define ((disj g . gs) s)
  (cond
    ((null? gs) (g s))
    (else ((λ (s)
             (append∞ (g s)
               ((apply disj gs) s)))
           s))))

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

(define-syntax disj
  (syntax-rules ()
    ((disj g) g)
    ((disj g gs ...) (disj₂ g (disj gs ...)))))
```

Listing 4. Derived `disj₂`-based function and macro.

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

Now `s` is a curried parameter we can η away. We provide the completed functional definition and a comparable to the macro-based implementation in Listing 4.

We can demonstrate the correctness of our `conj` from Listing 2 in the same manner. We first deconstruct the stream accumulator back into its two constituent pieces.

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

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

This we can collapse into a single recursive function, bringing the goal to the front.

```
(define ((conj g . gs) s)
  (cond
    ((null? gs) (g s))
    (else
     ((apply conj
        (cons
          (λ (s) (append-map∞ (car gs) (g s)))
          (cdr gs))
      s)))))
```

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

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

Listing 5. Derived `conj₂`-based function and macro.

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

Listing 6. A right-fold variant of `conj`.

We can replace the abstraction over `append-map∞`, which is to say the body of `conj₂`, by its definition.

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

By η reducing over the definition, we produce a simple variadic function. We show the result, and a comparable macro-based definition in Listing 5.

Both the variadic function based and the macro based versions of Listing 5 use a left fold over the goals, whereas the versions of `disj` use a right fold. This is not an accident, and we believe this is important. It is miniKanren folklore that left associating conjunctions tends to improve miniKanren performance, but we have generally resorted to small step visualizations of the search tree to demonstrate why that might be. We can make an equally compelling argument for this preference through equational reasoning. Listing 6 shows a right-fold variant of `conj`. Looking at either the functional or macro definitions built from `conj₂`, it's difficult to assert any preference for one over the other. The `apply` primitive and the macros both get in the way.

The choice 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 g gs s))

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

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

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

Since we know that any call to `append-map∞` we construct will always yield a result, the other one is tail recursive. This version is not, and it builds closures.

We can still reconstruct the stream argument, and come up with an approximately analogous right-fold version.

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

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

So 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 `conj2` and does not resort to `apply`, we have the two aforementioned choices. Basic programming horse sense suggests the variant from Listing 2.

Left associating conjuncts is so important that it dramatically improves miniKanren's search. A full, thorough, explanation of the behavior is outside the scope of our work here, but see discussions by Rosenblatt and Ballantyne [**here**, **nowhere**].

DONE UP UNTIL HERE

```
(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 7. A reimplemented 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 8. A typical implementation of `conda`.

miniKanren's biased search heuristic causes some operational differences between this reimplementation and the older one. With this implementation, instead of the *first* goal being the most heavily weighted in the search, here the *last* goal is most heavily weighted. Because *n*-ary disjunction is now built in to the functional definition, this behavior is now transparently visible in the implementation of `D`. Imagine for a moment that in the given state `s` each goal in `(g₁ . gs)` produces infinitely many answers. The `append` is doing a fair interleaving between its two stream arguments: giving one a little time to work and then handing off to the other. At each recursion, approximately half the answers in that stream come from the most recently evaluated goal, and so the resources dedicated to all the preceding goals is halved, in order to accommodate this new one.

A declarative programmer oughtn't need to concern himself with the operational search behavior, and since miniKanren provides a complete search, a query should still return the same bag of answers, so we do not see this as a problem; we just consider it an interesting distinction. If however operationally-aware programmers found the "reversed weighting" unintuitive, some language front end could rearrange the programmer's arguments. For example, reversing a list of arguments is an exemplary use of an APS macro.

IS IT ACTUALLY SO? HOW WOULD I IMPLEMENT FOLD THE OTHER WAY.

Though this note mainly concerns the choice to implement surface language functionality as macros, 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 `conj` and `disj` to the user.

4 Cleaning up the Impure Operators

The zero-ary conjunction and disjunction base cases both forced primitive goals `S` and `F` into the language and also hid the opportunity of functional left-fold implementations as helpers to variadic surface functions. The zero-ary conjunction case was itself implemented as an answer to an earlier problem in implementing an `if-then-else` soft-cut operator. The zero-ary disjunction base case was not *really* required, but given that we needed one the other came along for the ride, so to speak.

The operators `conda` and `conde` look superficially similar, syntactically. Semantically though, the disjunction and nested "if-then-else" behaviors are quite different, and implementing the desired behavior for this `conda` from existing combinators raises some oddities. The definition of `conda` already rules out zero-way conjunctions of goals in the clause body as well as zero-way disjunctions of such clauses. Another oddity is that, for a `conda` syntax that permits one-or-more-goals in each clause, the final clause of the `conda` definition is the one and only place in the implementation that may require a nullary conjunction of goals.

There are work-arounds. One could: 1. special case one-goal clauses in the last clause of the `conda` implementation with a special `ifte` variant. 2. special case that clause by introducing `S` as an additional goal 2. mandate that all clauses in `conda` contain at least two goals. 3. mandate that all clauses *except the final default clause* contain at least two goals. 4. build a special-case conjunction solely for this purpose. In all but the first of these choices, either the implementation implicitly or the syntax explicitly can force additional unneeded executions of unwanted goals. The first choices seems even worse for introducing another complex function definition that is nearly identical to one used in the below case.

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

(define ((ife g₁ g₃) s)
  (let loop ((s∞ (g₁ s)))
    (cond
      ((null? s∞) (g₃ s))
      ((pair? s∞) s∞)
      (else (lambda ()
              (loop (s∞)))))))

(define ((ifte g₁ g₂ g₃) s)
  ...)
```

Listing 9. An inauspicious re-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 10. A functional `conda` implementation.

Here too, removing macros and relying on variable arity function definition provides a more elegant solution. Some programmers would be perfectly satisfied to have lived forever with `ifte` itself. 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.

The implementation in Listing 9 includes the delay-and-restart behavior of `ifte` together with `conda`'s logical cascade.

4.1 condu

We have not yet addressed `condu`. We could implement this for functional programmers with a variant of Listing 9. However, we instead choose to make `once` a primitive higher-order goal.

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

Listing 11. A functional `once` implementation.

5 Conclusion

narrows the gap between implementations in functional host languages with and without macro systems.

More languages than ever before support variable arity functions/methods (aka `varargs` aka `slurpy` methods), including Raku, Java, and Ruby, to name just a few. Many of these languages also support polyvariadic functions. The authors hope that this document helps implementers build more elegant, expressive and efficient Kanrens in their chosen host languages.

Languages with variadic but without polyvariadic (that is to say "at least k arguments") functions—how do they fare here?

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

Macro systemless languages would still have to resort to exposing implementations of streams and *some* lower level operations like introducing variables one at a time. However....

We take it as granted that conjunction and disjunction themselves are sufficiently high-level operators for a surface language and that the 0-element base cases are at best unnecessary and likely undesireable.

Given the opportunity to define a surface language and its desugaring, we really shouldn't tempt the programmer by making undesireable programs representable when we can avoid doing so.

Of course, no implementer *needs* a macro system to implement a shallow embedding of an LP language. In our implementations, we still use macros to implement `defrel` and `fresh`, the former to prevent exposing the implementation of streams and delays, and the latter both for the added efficiency and to avoid the awkwardness of introducing each variable 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. Expert miniKanren programmers use macros to extend the language yet again, as with `matche`, and also with wholly more expressive and highly optimized

language forms, as in the original `Kanren`. It is nice to find we can spread some of that extra expressiveness to the broader Kanren language community.

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.

References

[1] 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.

[2] Jason Hemann and Daniel P. Friedman. 2013. Mkanren: a minimal functional core for relational programming. In *Scheme 13*. http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.

[3] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of DLS '16*. ACM. http://dx.doi.org/10.1145/2989225.2989230.

[4] Alex Shinn, John Cowan, and Arthur A. Gleckler. 2013. Revised^7 Report on the Algorithmic Language Scheme. Tech. rep. http://www.scheme-reports.org/.