

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 [3] 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 reimplementing 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.

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.

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

© 2018 Association for Computing Machinery.

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.

```

111 (defrel (carmelit-subway a b c d e f)
112   (conde
113     ((= a 'carmel-center)
114      (= b 'golomb)
115      (= c 'masada)
116      (= d 'haneviim)
117      (= e 'hadar-city-hall)
118      (= f 'downtown))
119     ((= a 'downtown)
120      (= b 'hadar-city-hall)
121      (= c 'haneviim)
122      (= d 'masada)
123      (= e 'golomb)
124      (= 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 as described in *The Reasoned Schemer, Second Edition* and in particular with microKanren implementations. See minikanren.org for more 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 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

```

166 (define ((conj g . gs) s)
167   (C gs (g s)))
168
169 (define (C gs s)
170   (cond
171     ((null? gs) s)
172     (else
173      (C (cdr gs)
174         (append-map (car gs) s))))))
175
176 (define ((disj g . gs) s)
177   (D g gs s))
178
179 (define (D g gs s)
180   (cond
181     ((null? gs) (g s))
182     (else
183      (append (g s)
184              (D (car gs) (cdr gs) s))))))
185
186 (define-syntax conde
187   (syntax-rules ()
188     ((conde (g g1 ...) ...)
189      (disj (conj g g1 ...) ...))))

```

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

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 disjunctive normal form. Listing 2 shows our new implementation.

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.

```

221 (define-syntax conj
222   (syntax-rules ()
223     ((conj) #s)
224     ((conj g) g)
225     ((conj g0 g ...) (conj2 g0 (conj g ...))))
226
227 (define ((conj2 g1 g2) s)
228   (append-map g2 (g1 s)))
229
230 (define-syntax disj
231   (syntax-rules ()
232     ((disj) #f)
233     ((disj g) g)
234     ((disj g0 g ...) (disj2 g0 (disj g ...))))
235
236 (define ((disj2 g1 g2) s)
237   (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 code 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.

```

266 (define ((disj g . gs) s)
267   (cond
268     ((null? gs) (g s))
269     (else
270      (append∞ (g s)
271                ((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.

```

276 (define ((disj g . gs) s)
277   (cond
278     ((null? gs) (g s))
279     (else ((λ (s)
280              (append∞ (g s)
281                        ((apply disj gs) s)))
282            s))))
283
284 (define (disj g . gs)
285   (cond
286     ((null? gs) g)
287     (else (disj2 g (apply disj gs)))))
288
289 (define-syntax disj
290   (syntax-rules ()
291     ((disj g) g)
292     ((disj g gs ...) (disj2 g (disj gs ...))))

```

Listing 4. Derived disj₂-based function and macro.

```

297 (define ((disj g . gs) s)
298   (cond
299     ((null? gs) (g s))
300     (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.

```

308 (define ((conj g . gs) s)
309   (C g gs s))
310
311 (define (C g gs s)
312   (cond
313     ((null? gs) (g s))
314     (else
315      (C (λ (s) (append-map∞ (car gs) (g s)))
316         (cdr gs)
317         s))))

```

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

```

321 (define ((conj g . gs) s)
322   (cond
323     ((null? gs) (g s))
324     (else
325      (apply conj
326               (cons
327                (λ (s) (append-map∞ (car gs) (g s)))
328                (cdr gs))
329               s))))

```

```

331 (define (conj g . gs)
332   (cond
333     ((null? gs) g)
334     (else
335      (apply conj
336       (cons (conj2 g (car gs)) (cdr gs))))))
337
338 (define-syntax conj
339   (syntax-rules ()
340     ((conj g) g)
341     ((conj g g1 gs ...)
342      (conj (conj2 g g1) gs ...))))

```

Listing 5. Derived conj₂-based function and macro.

```

346 (define (conj g . gs)
347   (cond
348     ((null? gs) g)
349     (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.

```

356 (define ((conj g . gs) s)
357   (cond
358     ((null? gs) (g s))
359     (else
360      ((apply conj
361       (cons (conj2 g (car gs)) (cdr gs)))
362       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 *worth further consideration*. 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.

```

383 (define ((conj g . gs) s)
384   (C g gs s))

```

```

386 (define (C g gs s)
387   (cond
388     ((null? gs) (g s))
389     (else (append-map-∞ (apply C gs) (g s)))))
390
391 (define ((conj g . gs) s)
392   (C gs (g s)))
393
394 (define (C g gs s)
395   (cond
396     ((null? gs) (g s))
397     (else (append-map-∞ (apply C gs) (g s)))))
398

```

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.

```

402 (define ((conj g . gs) s)
403   (C gs (g s)))
404
405 (define (C gs s∞)
406   (cond
407     ((null? gs) s∞)
408     (else
409      (append-map∞
410       (λ (s) (let ((t∞ ((car gs) s)))
411                (C (cdr gs) t∞)))
412       s∞))))
413

```

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.

```

418 (define ((conj g . gs) s)
419   (C gs (g s)))
420
421
422 (define (C gs s∞)
423   (cond
424     ((null? gs) s∞)
425     (else
426      (append-map-∞
427       (λ (s) (C (cdr gs) ((car gs) s)))
428       s∞))))
429

```

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

```

441 (defrel (carmelit-subway a b c d e f)
442   (disj
443     (conj (== a 'carmel-center)
444           (== b 'golomb)
445           (== c 'masada)
446           (== d 'haneviim)
447           (== e 'hadar-city-hall)
448           (== f 'downtown))
449     (conj (== a 'downtown)
450           (== b 'hadar-city-hall)
451           (== c 'haneviim)
452           (== d 'masada)
453           (== e 'golomb)
454           (== f 'carmel-center))))

```

Listing 7. A reimplemented Carmelit subway without conde.

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 (*g*₁ . . . *g*_s) 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.

Declarative programmers oughtn’t need concern themselves 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.

```

496 (define-syntax conda
497   (syntax-rules ()
498     ((conda (g0 g ...) (conj g0 g ...))
499       ((conda (g0 g ...) ln ...)
500         (ifte g0 (conj g ...) (conda ln ...))))))
501
502 (define ((ifte g1 g2 g3) s)
503   (let loop ((s∞ (g1 s)))
504     (cond
505       ((null? s∞) (g3 s))
506       ((pair? s∞)
507        (append-map g2 s∞))
508       (else (lambda ()
509                (loop (s∞)))))))
510

```

Listing 8. A typical implementation of conda.

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 last clause by introducing #s as an additional goal.
3. Mandate that all clauses in conda contain at least two goals.
4. Mandate that all clauses *except the final default clause* contain at least two goals.
5. 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


```

551 (define-syntax conda
552   (syntax-rules ()
553     ((conda (g0 g ...) (conj g0 g ...))
554      ((conda (g0) ln ...)
555       (if g0 (conda ln ...)))
556      ((conda (g0 g ...) ln ...)
557       (ifte g0 (conj g ...) (conda ln ...)))))
558
559 (define ((if g1 g3) s)
560   (let loop ((s∞ (g1 s)))
561     (cond
562       ((null? s∞) (g3 s))
563       ((pair? s∞) s∞)
564       (else (lambda ()
565                (loop (s∞)))))))
566
567 (define ((ifte g1 g2 g3) s)
568   ...)

```

Listing 9. An inauspicious re-implementation of conda.

```

573 (define ((conda q a . q-and-a*) s)
574   (A (q s) a q-and-a* s))
575
576 (define (A s∞ a q-and-a* s)
577   (cond
578     ((null? s∞)
579      (cond
580        ((null? (cdr q-and-a*)) ((car q-and-a*) s))
581        (else (A ((car q-and-a*) s)
582                  (cadr q-and-a*)
583                  (cddr q-and-a*) s))))))
584     ((pair? s∞) (append-map∞ a s∞))
585     (else (lambda () (A (s∞) a q-and-a* s)))))
586

```

Listing 10. A functional conda implementation.

seems even worse for introducing another complex function definition that is nearly identical to one used in the below case.

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.

```

606 (define (once g)
607   (lambda (s)
608     (let loop ((s∞ (g s)))
609       (cond
610         ((null? s∞) '())
611         ((pair? s∞)
612          (cons (car s∞) '()))
613         (else (lambda ()
614                  (loop (s∞)))))))

```

Listing 11. A functional once implementation.

4.1 condu

We have not yet addressed condu. We could implement this for functional programmers with a variant of Listing 9. Instead, we choose to make once a primitive higher-order goal. This relies on a local let-bound named recursive procedure.

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 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.

Languages without expressive macro systems would still have to resort to exposing implementations of streams and *some* lower level operations like introducing variables one at a time. But!

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 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 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 wider Kanren language community.

Early microKanren implementations restrict themselves to syntax-rules macros. Several more powerful syntax constructions over miniKanren may rely on more expressive macro systems, including Ballantyne’s `minikanren-ee` [1].

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] 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] 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.
- [3] 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>.
- [4] Alex Shinn, John Cowan, and Arthur A. Gleckler. 2013. Revised? Report on the Algorithmic Language Scheme. Tech. rep. <http://www.scheme-reports.org/>.