Some criteria for implementations of conjunction and disjunction in microKanren

Anonymous Author(s)

Abstract

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

ACM Reference Format:
Anonymous Author(s). 2022. Some criteria for implementations of conjunction and disjunction in microKanren. In *Proceedings of miniKanren Workshop (mKW '22).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

microKanren [4] 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 future would-be implementers when studying the source code, and also that the functional core would make the language easier to port to other functional hosts. To support both those efforts, they also chose to program in a deliberately small and workaday set of Scheme primitives.

This note shows how an implementer using a host language with variadic (any number of arguments) functions can build a somewhat more powerful kernel language and obviate macros for conjunction and disjunction. Until now there was a large gap between those microKanren implementations in languages with expressive macro systems and those without; variadic functions permit an intermediate point in the language design space. Even implementers in languages *with* macro systems might choose to replace existing less powerful language primitives by our comparatively short but more powerful variants. Beyond that, this more powerful kernel language obviates several macros in the desugaring layer, and eventually leads us to see merit in modifying the core language's syntax. This is, of course, subjective.

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

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 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.

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

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

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

Listing 1. A miniKanren implementation of the Carmelit subway.

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

(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) #u)
    ((disj g) g)
    ((disj g₀ g ...) (disj₂ g₀ (disj g ...)))))

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

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

In a microKanren in a macro-less host language, a programmer could get bothered by low-level details such as stacking binary conjunction and disjunction. 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 away from DSL-level thinking, and back into the host programming language's debug model.

```
(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)))))
```

Listing 3. Re-implementation of `disj`.

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.

3 `conj` and `disj` logical goal constructors

The `conj` and `disj` macros of Listing 2 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 such goals should behave, so there is no benefit to executing them. A conjunction of no goals would simply succeed, and a disjunction of no goals would simply fail. Truthfully, only the impure `conda` operator seems to require these additional base cases (discussed further in Section 4) they are for all intents and purposes superfluous.

Here, we want to implement conjunction and disjunction over one-or-more goals as functions. These implementations will rely on variadic function syntax, but should not require `apply` or rely on the binary `conj₂` and `disj₂`. Further, they should not build any extraneous closures: unnecessarily building closures at runtime is always a bad idea.

Listings 3 and 4 show our new implementations. Mandating one-or-more arguments lets us re-implement these operators as shallow wrappers over simple folds. The first step in each is merely to remove the rest argument `gs` and act as if there were no need for a rest argument. That is, all of our focus will be on the *list*, `gs`. 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

```
(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. Re-implementation of `conj`.

append-map for miniKanren streams) the next goal in the list. This left-fold implementation of conjunction therefore left-associates the conjuncts.

3.1 Implementation correctness

Equipped with variadic functions, a developer might discover these definitions as follows. We start with the definition of `disj` from Listing 5. At the cost of an `apply`, we can define `disj` as a single explicitly recursive 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. `disj₂`-based macro and function definition.

Since `disj` produces and consumes goals, we can η expand the definition by a curried parameter `s`. We also split `disj` into two mutually-recursive procedures.

```
(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))))
```

We can replace the call to `disj₂` by its definition in terms of `append∞` and perform a trivial β-reduction. The explicit `s` argument suggests making `D` recur into itself. The result is the definition in Listing 3.

We can derive the definition of `conj` from Listing 4 via a similar process. Starting with the variadic function definition of Listing 6, we first η-expand and split the definition.

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

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

We will substitute the definitions of `conj` and `conj₂`.

```
(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))))
```

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

Both the variadic function based and the macro based versions of Listing 6 use a left fold over the goals, whereas the versions of `disj` use a right fold. This is not an accident. It is miniKanren folklore that left associating conjunctions tends to improve miniKanren performance. A full, thorough, explanation of what we know about this behavior is outside the scope of our work here, but see discussions by Rosenblatt [**nowhere**] and Ballantyne [**nowhere**]. The matter seems so significant that the community might consider reclassifying left-associative conjunction as a matter of correctness rather than an optimization, as in "tail call optimization" vs. "Properly Implemented Tail Call Handling" [2].

However, we have generally resorted to small step visualizations of the search tree to demonstrate why that might be. The authors believe it is *worth further consideration* if we can make an equally compelling argument for this preference through equational reasoning. Listing 7 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)
  (cond
    ((null? gs) g)
    (else (conj₂ g (apply conj gs)))))
```

Listing 7. A right-fold variant of `conj`.

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

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

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

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

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 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 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 `conj` and `disj` to the user.

```
(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 8. 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 9. A typical implementation of `conda`.

4 Cleaning up the Impure Operators

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 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 pieces raises some oddities. The definition of `conda` (see Listing 9) rules out zero-way conjunctions of goals in the clause body as well as zero-way disjunctions of such clauses. A `conda` that permits one-or-more-goals in each clause is the one and only place in the whole langauge implementation that may require a nullary conjunction of goals. So this soft-cut operator seems to force both nullary conjunction and those primitive goals `#s` and `#u` into the language.

```
(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.

```
(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.

There are alternative choices one could consider.

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

All these choices, however, implicitly or explicitly can force additional unneeded executions of unwanted goals. Here too, removing macros and relying on variable arity function definition provides a more elegant solution.

The implementation in **??** 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 **??**. 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 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.

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 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 wider Kanren language community.

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

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] Matthias Felleisen. [n. d.] Re: question about tail recursion. (). https://lists.racket-lang.org/users/archive/2014-August/063844.html.

[3] 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.

[4] 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.