Nearly Macro-free microKanren

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

Abstract

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

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

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

ACM Reference Format:
Jason Hemann and Daniel P. Friedman. 2023. Nearly Macro-free microKanren. In *Proceedings of Symposium on Trends in Functional Programming (TFP '23).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Initially we designed microKanren [6] as a compact relational programming language kernel to undergird a miniKanren implementation. Macros are used to implement the surrounding higher-level miniKanren operators and surface syntax. microKanren is often used as a tool for understanding the guts of a relational programming language through studying its implementation. By re-implementing miniKanren as separate surface syntax macros over a macro-less and side-effect free microKanren kernel, we hoped to simultaneously aid implementers when studying the source code, and also to make the language easier to port to other hosts that support programming with functions. To support both of those efforts, we also chose to program in a deliberately small and workaday set of Scheme primitives.

The sum of those implementation restrictions, however, seemed to force some awkward choices including: binary logical operators, one at a time local variable introduction, and leaks in the stream abstractions. These made the surface syntax macros seem almost required, and were far enough from our goals that *The Reasoned Schemer, 2nd Ed* [4] did not use a macro-less functional kernel. It also divided host languages into those that used macros and those that did not. We bridge some of that split by re-implementing parts of the kernel using some modest runtime features common to many languages.

Here we:

- show how to functionally implement more general logical operators, cleanly obviating some of the surface macros,
- survey the design space of macro-less, mostly side-effect free implementation alternatives for the remaining macros in the *TRS2e* core language implementation, and weigh the trade-offs and real-world consequences, and
- suggest practical solutions for completely eliminating the macros in those places where the purest microKanren implementations seemed impractical.

This resulted in some higher-level (variadic rather than just binary) operators, a more succinct kernel language, and enabled some performance improvement. Around half of the changes are applicable to any microKanren implementation, and the more concise goal combinators of Section 3 may also be of interest to implementers who embed goal-oriented languages like Icon [5]. The other half are necessarily awkward yet practical strategies for those platforms lacking macro support. The source code for both our re-implementation and our experimental results is available at https://github.com/jasonhemann/tfp-2023/.

In Section 2, we illustrate by example what seemed to force surface syntax macros. In Section 3, we implement conjunction and disjunction, and in Section 4 we discuss the re-implementation of the impure operators. We discuss the remaining macros in Section 5. We close with a discussion of the performance impacts of these implementation choices, and consider how implementers of Kanren languages in other hosts might benefit from these alternatives.

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

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

2 The problem

We briefly reprise here some basics of programming in the miniKanren implementation of *TRS2e*. Although based on microKanren, the *TRS2e* implementation makes some concessions to efficiency and safety and uses a few macros in the language kernel itself. In addition to that implementation, in this paper we make occasional references to earlier iterations such as Hemann et al. [7], an expanded archival version of the 2013 paper [6].

2.1 Background

Programs consist of a database of relation definitions and queries executed in the context of those definitions. A relation definition consists of a name for the relation and a number of parameters equal to the relations arity, and then a body. This body states the condition under which the relationship holds between the parameters. Parameters range over a domain of terms, and the programmer can introduce local auxiliary variables as needed. The core *TRS2e* language implementation relies on macros `fresh`, `defrel`, and `run` to introduce new logic variables, globally define relations, and execute queries. Relations can refer to themselves or one another in their definitions; the whole database of relations is mutually-recursively defined. The programmer states the relation body as a logical expression, built up with conjunctions and disjunctions, over a class of equations and primitive constraints on terms. Take the `carmelit-subway` relation of Listing 1 as a concrete example. The Carmelit in Haifa is the world's shortest subway system with only six stations on its line; `carmelit-subway` is a six-place relation modeling a passenger riding that subway end to end. `carmel-center`, `golomb`, `masada`, etc. are all term constants in the program; each is the name of a stop on the line. The sub-expression (`== a 'carmel-center`) is an equation between the parameter `a` and the term `carmel-center`. The `conde` operator (the `e` is for "every line") takes any number of parenthesized sequences of expressions. Each parenthesized sequence represents the conjunction of those expressions, and the meaning of the `conde` expression represents the disjunction of those conjunctions. Altogether, this program states that there are two ways to ride the subway end to end: a passenger can either start at `carmel-center` and travel five stops to `downtown`, or start at `downtown` and travel five stops to `carmel-center`.

Each query against the database contains some expression the programmer wants to satisfy and introduces some number of variables against which the language should express the answer. Each query asks either for all, or some bounded number of answers. The `run` expression of Listing 1 is a query—it asks for at most three ways a passenger could have ridden the Carmelit, printed as a list of six values

```
(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))))

> (run 3 (s₀ s₁ s₂ s₃ s₄ s₅)
    (carmelit-subway s₀ s₁ s₂ s₃ s₄ s₅))
'((carmel-center golomb masada
   haneviim hadar-city-hall downtown)
  (downtown hadar-city-hall haneviim
    masada golomb carmel-center))
```

Listing 1. The Carmelit subway relation and a query. Results reformatted for clarity and space.

for variables $s_0 \ldots s_5$. miniKanren returns a list of two answers, the only two ways a passenger could ride the whole subway end to end.

2.2 Limitations of microKanren

A six station subway line is an example sufficiently small that modeling it should be painless. Without the higher-level operators that the miniKanren macros implement, however, that same relation requires 11 logical operator nodes, because microKanren only provides *binary* conjunctions and disjunctions. In our view, this makes the superficial syntax macros practically mandatory, and impedes host languages without a macro system. For a logic programming language, solely binary logical operators is too low level.

Moreover, the microKanren language doesn't offer the programmer sufficient guidance in using that fine-grained control. With n goals, the programmer can associate to the left, to the right, or some mixtures of the two. The syntax does not obviously encourage any one choice. Subtle changes in program structure can have profound effects on performance, and mistakes are easy to make.

Similarly, the soft-cut operator `ifte` in the *TRS2e* language kernel is also low level. It permits a single test, a single consequent, and a single alternative. To construct an if-then-else cascade, a microKanren programmer without the `conda` surface macro would need to code that unrolled conditional expression by hand.

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

(define ((conj₂ g₁ g₂) s)
  (append-map∞ g₂ (g₁ s)))

(define-syntax disj
  (syntax-rules ()
    ((disj g) g)
    ((disj g₁ g₂ g* ...)
     (disj₂ g₁ (disj g₂ g* ...)))))

(define-syntax conj
  (syntax-rules ()
    ((conj g) g)
    ((conj g₁ g₂ g* ...)
     (conj (conj₂ g₁ g₂) g* ...))))
```

Listing 2. microKanren `disj₂`, `conj₂`, and macros that use them

In earlier renditions of related work, the strictures of pure functional shallow embedding—without macros—seemed to force implementations of variable introduction, relation definition, and the query interface that suffered from harsh downsides. We will revisit those earlier implementations and their trade-offs, survey the available options, and suggest compromises for those truly without macros, thus increasing microKanren's *practical* portability.

3 The `disj`unction and `conj`unction goal constructors

microKanren's binary `disj₂` and `conj₂` operators, shown in Listing 2, are goal combinators: they each take two goals, and produce a new goal. A *goal* is what the program attempts to achieve: it can fail or succeed (and it can succeed multiple times). A goal executes with respect to a *state*, here the curried parameter `s`, and the result is a *stream* of states, usually denoted `s∞` as each entry in the stream is a state that results from achieving that goal in the given state. Disjunction and conjunction work slightly differently. The `append∞` function used in `disj₂` is a kernel primitive that combines two streams into one, with an interleave mechanism to prevent starvation; the result is a stream of the ways to achieve the two goals' disjunction. The `append-map∞` function used in `conj₂` is to `append∞` what `append-map` is to `append`. The ways to achieve the conjunction of two goals are all the ways to achieve the second goal in a state that results from achieving the first goal. `append-map∞` runs the second goal over the stream of results from the first goal, and combines together the results of mapping into a single stream. That stream represents the conjunction of the two goals, again with special attention to interleaving and starvation.

We want to implement disjunction and conjunction as functions taking arbitrary quantities of goals. These implementations should subsume the binary `disj₂` and `conj₂` and they also should not use `apply`. Nor should our implementations induce a tortured program and extraneous closures to *avoid* using `apply`; we address this somewhat-cryptic requirement in Section 3.2. This re-implementation requires a host that supports variable arity functions, a widely available feature included in such languages as JavaScript, Ruby, Java, and Python. These languages do not generally support macros, and so we advise implementers working in such languages to use the present approach.

3.1 Calculating the solutions

A developer might derive these definitions as follows. We start with the definition of a recursive `disj` macro like one might define as surface syntax over the microKanren `disj₂`. As this is not part of the microKanren language, we would like to dispense with the macro and implement this behavior functionally. At the cost of an `apply`, we can build the corresponding explicitly recursive `disj` function. Since `disj` produces and consumes goals, we can η-expand that first functional definition by a curried parameter `s`. We then split `disj` into two mutually-recursive functions. We use the symbol ⊂ here to indicate that the newer `disj` produces the same values as the old one, although it now does so by calling a new globally-defined function `D`. This new help function will only ever be called from `disj` so does not *need* to be global; we do so for space and clarity. In this paper, every such help function we introduce with ⊂ could have instead been local and the relationship actually an equality.

```
(define (disj g . g*)
  (cond
    ((null? g*) g)
    (else (disj₂ g (apply disj g*)))))
= { η-expansion }
(define ((disj g . g*) s)
  (cond
    ((null? g*) (g s))
    (else ((disj₂ g (apply disj g*)) s))))
⊂
(define ((disj g . g*) s)
  (D g g* s))

(define (D g g* s)
  (cond
    ((null? g*) (g s))
    (else ((disj₂ g (apply disj g*)) s))))
```

In that version of `D`, 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 removing the call to `apply` and making `D` self-recursive. The definition of `disj` remains unchanged from before.

```
= { by definition of disj₂ }
(define (D g g* s)
  (cond
    ((null? g*) (g s))
    (else
     ((λ (s)
        (append∞ (g s) ((apply disj g*) s)))
      s))))
= { β-reduction }
(define (D g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (append∞ (g s) ((apply disj g*) s)))))
= { by definition of disj }
(define (D g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (append∞ (g s) (D (car g*) (cdr g*) s)))))
```

In both clauses of `D` we combine `g` and `s`, this suggests constructing that stream in `disj` and passing it along. Adding the trivial base case to that `disj` yields the definition in Listing 3.

We can derive the definition of `conj` from Listing 3 via a similar process. Starting with the variadic function based on the macro in Listing 2, we first η-expand and split the definition.

```
(define (conj g . g*)
  (cond
    ((null? g*) g)
    (else
     (apply conj
       (cons (conj₂ g (car g*)) (cdr g*))))))
= { η-expansion }
(define ((conj g . g*) s)
  (cond
    ((null? g*) (g s))
    (else
     ((apply conj
        (cons (conj₂ g (car g*)) (cdr g*))) s))))
⊂
(define ((conj g . g*) s)
  (C g g* s))

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

We next substitute for the definitions of `conj` and `conj₂`.

```
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else
     ((apply conj
        (cons (conj₂ g (car g*)) (cdr g*)))
      s))))
= { by the definition of conj }
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (C (conj₂ g (car g*)) (cdr g*) s))))
= { by the definition of conj₂ }
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (C (λ (s)
          (append-map∞ (car g*) (g s)))
        (cdr g*)
        s)))))
```

This definition of `C` sequences the invocations of goal and state in the order they appear. `append-map∞` acts like a nondeterministic `compose` operator. In each recursive call, we accumulate by mapping, using `append-map∞`'s special delaying implementation of Kanren-language streams, the next goal in the list.

```
(C g (list g₁ g₂) s)
=
(C (λ (s)
     (append-map∞ g₁
       (g s)))
   (list g₂)
   s)
=
(C (λ (s)
     (append-map∞ g₂
       ((λ (s)
          (append-map∞ g₁
            (g s)))
        s)))
   '()
   s)
=
((λ (s)
   (append-map∞ g₂
     ((λ (s)
        (append-map∞ g₁
          (g s)))
      s)))
 s)
```

```
(define ((disj . g*) s)
  (cond
    ((null? g*) '())
    (else (D ((car g*) s) (cdr g*) s))))

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

(define ((conj . g*) s)
  (cond
    ((null? g*) (cons s '()))
    (else (C (cdr g*) ((car g*) s)))))

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

Listing 3. Final re-definitions of `disj` and `conj`

The state does not change in the recursion: `C` only needs `s` to *build* the stream. Therefore 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. Adding the trivial base case to `conj`, yields the version shown in Listing 3.

Both of these new versions are shallow wrappers over simple folds. The first steps are to dispense with the trivial case, and then to call a recursive help function that makes no use of variadic arguments. The focus is on recurring over the list `g*`. Unlike `D`, the function `C` does not take in the state `s`; the help function does not need the state for conjunction.

3.2 What's left?

More importantly, while both the functional and the macro based versions of `disj` use a right fold, the implementation of conjunctions in Listing 3 uses a left fold over the goals. This left-fold implementation of conjunctions therefore left-associates the conjuncts. This is not an accident.

Folklore suggests left associating conjunctions tends to improve 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 Rosenblatt et al. [10] for some of the related work so far. We have generally, however, resorted to small step visualizations of the search tree to explain the performance impact. It is worth considering if we can make an equally compelling argument for this preference through equational reasoning and comparing the implementations of functions.

Compare the preceding derivation from a left-fold over conjunctions with the following attempted derivation from a right-fold implementation. We η-expand and unfold to a recursive help function like before.

```
(define (conj g . g*)
  (cond
    ((null? g*) g)
    (else (conj₂ g (apply conj g*)))))
= { η-expansion }
(define ((conj g . g*) s)
  (cond
    ((null? g*) (g s))
    (else ((conj₂ g (apply conj g*)) s))))
⊂
(define ((conj g . g*) s)
  (C g g* s))

(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else ((conj₂ g (apply conj g*)) s))))
```

In `C`, we can substitute in the definition of `conj₂` and β-reduce. We once again η-expand the call to `(apply conj g*)` to substitute with the definition of `conj` and eliminate the `apply`. There, however, we get stuck.

```
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else ((conj₂ g (apply conj g*)) s))))
=  { by definition of conj₂ and β-reduction }
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else
      (append-map∞
        (apply conj g*)
        (g s)))))
= { η-expand to use the definition of conj }
(define (C g g* s)
  (cond
    ((null? g*) (g s))
    (else
      (append-map∞
        (λ (s)
          (C (car g*) (cdr g*) s))
        (g s)))))
```

At this point in the left-fold derivation, these calls are accumulating in a stack-like discipline, so we can simplify and pass `g` and `s` together as a stream. The equivalent right-fold implementation, however, is in a kind of continuation-passing style for non-deterministic computations. Each goal in the list seems to require a closure for every recursive call. Building these closures is expensive. Similar behavior shows up in the right-fold variant of `disj`. Basic programming horse sense suggests the more elegant variants from Listing 3, and could partly explain the performance gap.

```
(C g (list g₁ g₂) s)
=
(append-map∞
  (λ (s)
    (C g₁ (list g₂) s))
  (g s))
=
(append-map∞
  (λ (s)
    (append-map∞
      (λ (s)
        (C g₂ '() s))
      (g₁ s)))
  (g s))
=
(append-map∞
  (λ (s)
    (append-map∞
      (λ (s)
        (g₂ s))
      (g₁ s)))
  (g s))
```

The new `disj` and `conj` functions are, we believe, sufficiently high-level for programmers in implementations without macros. Though this note mainly concerns working towards an internal macro-less kernel language, it may also have something to say about the miniKanren-level surface syntax, namely that even the miniKanren language could do without its `conde` syntax (a disjunction of conjunctions that looks superficially like Scheme's **`cond`**) and have the programmer use these new underlying logical primitives. In Listing 4, we implement `carmelit-subway` as an example, and it reads much better than the 11 binary logical operators the programmer would have needed in the earlier version.

```
(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 4. A new Carmelit subway without `conde`

```
(define-syntax conda
  (syntax-rules ()
    ((conda g) g)
    ((conda g₁ g₂) (conj g₁ g₂))
    ((conda g₁ g₂ g₃ g* ...)
     (ifte g₁ g₂ (conda g₃ g* ...)))))

(define (conda g . g*)
  (cond
    ((null? g*) g)
    ((null? (cdr g*)) (conj g (car g*)))
    (else
     (ifte g (car g*) (apply conda (cdr g*))))))
```

Listing 5. A re-implemented `conda` macro and its functional equivalent

4 Tidying up the impure operators

The `conda` of *TRS2e* provides nested "if-then-else" behavior (the `a` is because at most one line succeeds). It relies on microKanren's underlying `ifte`. That `conda` requires one or more conjuncts per clause and one or more clauses. Once again, we would like to have an equivalently expressive feature without resorting to macros. Unlike the *TRS2e* implementation, the versions of `conda` in Listing 5 consume a sequence of goals. They consume those goals in "if-then" pairs, perhaps followed by a final "else"; we have no choice if we want to proceed without using a macro.

We defer the derivation of our new `conda` solution to Appendix A. Rather than building a largely redundant implementation of `condu`, we expose the higher-order goal `once` to the user. The programmer can simulate `condu` by wrapping `once` around every test goal. We do however take this opportunity to also lift the inner function `loop` to a top-level definition.

5 Removing more macros

The 2013 microKanren paper demonstrates how to implement a side-effect free macro-less Kanren language in an eager host. In Listing 7 we display these alternative mechanisms for introducing fresh logic variables, executing queries,

```
(define ((conda . g*) s)
  (cond
    ((null? g*) '())
    (else (A (cdr g*) ((car g*) s) s))))

(define (A g* s∞ s)
  (cond
    ((null? g*) s∞)
    ((null? (cdr g*)) (append-map∞ (car g*) s∞))
    (else (ifs∞te s∞ (car g*) (cdr g*) s))))

(define (ifs∞te s∞ g g+ s)
  (cond
    ((null? s∞) (A (cdr g+) ((car g+) s) s))
    ((pair? s∞) (append-map∞ g s∞))
    (else (λ () (ifs∞te (s∞) g g+ s)))))

(define ((once g) s)
  (0 (g s)))

(define (0 s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞) (cons (car s∞) '()))
    (else (λ () (0 (s∞))))))
```

Listing 6. A functional `conda`, `ifte`, and `once`

```
(define ((call/fresh f) s)
  (let ((v (state->newvar s)))
    ((f v) (incr-var-ct s))))

(define (call/initial-state n g)
  (reify/1st
    (take∞ n (pull (g initial-state)))))

(define (((Zzz g) s))
  (g s))

> (call/initial-state 3
    (call/fresh
      (λ (x)
        (== x 'cat))))
```

Listing 7. Definition and use of functional microKanren equivalents of *TRS2e* kernel macros

and introducing delay and interleave. The versions in Listing 7 are slightly adjusted to be consistent with this presentation.

Each of these has drawbacks that compelled the *TRS2e* authors to instead use macro-based alternatives in the kernel layer. In this section, we explicitly address those drawbacks and point out some other non-macro alternatives that may demand more from a host language than the original microKanren choices, and make some recommendations.

5.1 Logic variables

Many of the choices for these last options hinge on a representation of logic variables. Every implementation must have a mechanism to produce the next fresh logic variable. The choice of variable representation will affect the implementation of unification and constraint solving, the actual introduction of fresh variables, as well as answer projection, the formatting and presentation of a query's results. Depending on the implementation, the variables may also need additional functions to support them. In a shallow embedding, designing a set of logic variables means either using a **`struct`**-like mechanism to custom-build a datatype hidden from the microKanren programmer, or designating some subset of the host language's values for use as logic variables. Using **`struct`**s and limiting the visibility of the constructors and accessors is a nice option for languages that support it.

The choice of which host language values to take for logic variables divides roughly into the structurally equal and the referentially equal. For an example of the latter, consider representing each variable using a vector, and identifying vectors by their unique memory location. This latter approach models logic variables as a single global pool rather than reused separately across each disjunct, and so requires more logic variables overall. The microKanren approach uses natural numbers as an indexed set of variables, which necessitates removing numbers from the user's term language.

5.2 `fresh`

There are numerous ways to represent variables, and so too are there many ways to introduce fresh variables. In the microKanren approach, the current variable index is one of the fields of the state threaded through the computation; to go from index to variable is the identity function, and the `state->newvar` function we use can be just an accessor. The function `incr-var-ct` can reconstruct that state with the variable count incremented. The `call/fresh` function, shown in Listing 7, takes as its first argument a goal parameterized by the name of a fresh variable. `call/fresh` then applies that function with the newly created logic variable, thereby associating that host-language lexical variable with the DSL's logic variable. This lets the logic language "piggyback" on the host's lexical scoping and variable lookup, as shown in Listing 7.

This approach also means, however, that absent some additional machinery the user must introduce those new logic variables one at a time, once each per `call/fresh` expression, as though manually currying all functions in a functional language. This made programs larger than the relational `append` difficult to write and to read, and that amount of threading and re-threading state for each variable is costly. We could easily support, say instead, three variables at a time—force the user to provide a three-argument function and always supply three fresh variables at a time. Though practically workable the choice of some arbitrary quantity k of variables at a time, or choices k_1 and k_2 for that matter, seems unsatisfactory. It could make sense to inspect a procedure for its arity at runtime and introduce exactly that many variables, in languages that support that ability. In many languages, a procedure's arity is more complex than a single number. Variadic functions and keyword arguments all complicate the story of a procedure's arity. A form like **`case-lambda`** means that a single procedure may have several disjoint arities. The arity inspection approach could be a partial solution where the implementer restricts the programmer to using functions with fixed arity.

One last approach is to directly expose to the user a mechanism to create a new variable, and allow the programmer to use something like a **`let`** binding to do their own variable introduction and name binding. Under any referentially transparent representation of variables, this would mean that the programmer would be responsible for tracking the next lexical variable. This last approach pairs best with referentially opaque variables where the operation to produce a new variable allocates some formerly unused memory location so the programmer does not need to track the next logic variable. See sokuza-kanren [9] for an example of this style. With this latter approach, however, we can expose `var` directly to the programmer who can use **`let`** bindings to introduce several logic variables simultaneously.

5.3 `run`

Listing 7 also shows how we have implemented a `run`-like behavior without using macros. Using a referentially transparent implementation of logic variables, we can accomplish the job of `run` and `run*` by a `call/initial-state`-like function. The query is itself expressed as a goal that introduces the first logic variable `q`. A `run`-like operator displays the result with respect to the first variable introduced. This means pruning superfluous variables from the answer, producing a single value from the accumulated equations, and numbering the fresh variables. When logic variables are only identified by reference equality, the language implementation must pass the same *pointer-identical* logic variable into both the query and into the answer projection, called `reify`. 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 the first example in Listing 8, or create a special variable introduction mechanism for the first variable, scoped over both the query and the answer projection, as in the second example.

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

(define (call/initial-state n f)
  (let ((q (var 'q)))
    (map (reify q)
         (take∞ n ((f q) initial-state)))))
```

Listing 8. Several approaches to reifying variables in `call/initial-state`. Here `initial-state` is a representation of an initially empty set of equations

5.4 `define`

The microKanren programmer can use their host language's **`define`** feature to construct relations as host-language functions, and manually introduce the delays in relations using a help function like `Zzz` (Listing 7) to introduce delays, as in the original implementation. [6] 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, in an eager host language, even the termination of relation *definitions* rely on a whole-program correctness property of relation definitions having a delay. `Zzz` *if always used correctly* would be sufficient to address that problem, but forgetting it just once could cause the entire program to loop. Turning the delaying and interleaving into a user-level operation means giving the programmer some explicit control over the search, and that in turn could transform a logic language into an imperative one. Another downside of relying on a host-language **`define`** is that the programmer must now take extra care not to provide multiple goals in the body. The **`define`** form will treat all but the last expression as statements and silently drop them, rather than conjoin them as in `defrel`. For those implementing a shallowly embedded stream-based implementation in an eager host language, that can be a subtle mistake to debug.

6 Future work

This note shows how to provide a somewhat more concise core language that significantly reduces the need for macros, and provides some alternatives for those working without macros that may be more practical than those of Hemann and Friedman [6].

Forcing ourselves to define `disj` and `conj` functionally, and with the restrictions we placed on ourselves in this re-implementation, 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. The prior desugaring macros do not seem to suggest how to associate the calls to the binary primitives—both left and right look equally nice—where these transformations suggest a reason for the performance difference. The functional `conda` re-implementation is now also variadic, and exposing `once` to the programmer makes using committed choice almost as easy as with the earlier `condu`.

Existing techniques for implementing `defrel`, `fresh`, and `run` (and `run*`) without macros have serious drawbacks. They 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 about global state. With a few more runtime features from the host language, an implementer can overcome some of those drawbacks, and may find one of the suggested proposals an acceptable trade-off.

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` [8]. 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 narrows the gap between macro-free microKanrens and those using macro systems, and leads to more elegant, expressive and efficient implementations regardless of functional host language.

Acknowledgments

Thanks to Michael Ballantyne, Greg Rosenblatt, Ken Shan, and Jeremy Siek for their helpful discussions and ideas. Our thanks to Yafei Yang and Darshal Shetty for their implementation 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] Ralph E Griswold and Madge T Griswold. *The Icon programming language.* Vol. 55. Prentice-Hall Englewood Cliffs, NJ, 1983.

[6] 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.

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

[8] 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.

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

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

A `conda` derivation

```
(define (conda g . g*)
  (cond
    ((null? g*) g)
    ((null? (cdr g*)) (conj g (car g*)))
    (else
     (ifte g (car g*) (apply conda (cdr g*))))))
= { η-expansion }
(define ((conda g . g*) s)
  (cond
    ((null? g*) (g s))
    ((null? (cdr g*)) ((conj g (car g*)) s))
    (else
     ((ifte g (car g*) (apply conda (cdr g*)))
      s))))
⊂
(define ((conda g . g*) s)
  (A g g* s))
```

```
(define (A g g* s)
  (cond
    ((null? g*) (g s))
    ((null? (cdr g*)) ((conj g (car g*)) s))
    (else
     ((ifte g (car g*) (apply conda (cdr g*)))
      s))))
= { let bindings and η-expansion }
(define (A g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (let ((g1 (car g*)) (g* (cdr g*)))
       (cond
         ((null? g*) ((conj g g1) s))
         (else
          ((ifte g g1
             (λ (s)
               ((apply conda g*) s)))
           s)))))))
= { by definition of conda, conj, and C }
(define (A g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (let ((g1 (car g*)) (g* (cdr g*)))
       (cond
         ((null? g*) (append-map∞ g1 (g s)))
         (else
          ((ifte g g1
             (λ (s)
               (A (car g*) (cdr g*) s)))
           s)))))))
= { by definition of ifte and a β reduction }
(define (A g g* s)
  (cond
    ((null? g*) (g s))
    (else
     (let ((g1 (car g*)) (g* (cdr g*)))
       (cond
         ((null? g*) (append-map∞ g1 (g s)))
         (else
          (let loop ((s∞ (g s)))
            (cond
              ((null? s∞) (A (car g*) (cdr g*) s))
              ((pair? s∞) (append-map∞ g1 s∞))
              (else (λ () (loop (s∞)))))))))))))
```

At this point `g` and `s` in `conda` are *begging* to be passed as a stream `s∞`; we oblige them. We lift that local function `loop` to a global definition, passing all the parameters it needs. Since the only call to `ifs∞te` is in `A`, we know that `ifs∞te`'s third parameter will always be a non-empty list.

```
(define (A g* s∞)
  (cond
    ((null? g*) s∞)
    (else
     (let ((g1 (car g*)) (g* (cdr g*)))
       (cond
         ((null? g*) (append-map∞ g1 s∞))
         (else
          (let loop ((s∞ s∞))
            (cond
              ((null? s∞) (A (cdr g*) ((car g*) s)))
              ((pair? s∞) (append-map∞ g1 s∞))
              (else (λ () (loop (s∞))))))))))))
⊂
(define (A g* s∞)
  (cond
    ((null? g*) s∞)
    (else
     (let ((g1 (car g*)) (g* (cdr g*)))
       (cond
         ((null? g*) (append-map∞ g1 s∞))
         (else (ifs∞te s∞ g1 g* s)))))))

(define (ifs∞te s∞ g g+ s)
  (cond
    ((null? s∞) (A (cdr g+) ((car g+) s)))
    ((pair? s∞) (append-map∞ g s∞))
    (else (λ () (ifs∞te (s∞) g g+ s)))))
```

If we also add a line in `conda` to dispatch with the trivial case, we arrive at the definition in Listing 6. Most of Listing 6 is a functional implementation of that cascade behavior. `A` knows it has at least one goal; it's job is to determine if there is precisely one goal, precisely two goals, or more than two goals.