Nearly Macro-free microKanren

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

Abstract

This paper describes changes to the microKanren implementation that make it more practical to use in a host language without macros. With the help of 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, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The authors 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 purely-function microKanren kernel, the authors hoped this separation would simultaneously aid implementers when studying the source code, and also that the functional core would make the language easier to port to other functional hosts. To support both those efforts, they also chose to program in a deliberately small and workaday set of Scheme primitives.

The sum of those implementation restrictions, however, necessitates some awkward compromises in places including binary logical operators, one at a time local variable introduction, and leaks in the stream abstractions. These made the surface syntax macros seem practically mandatory, and fell short enough that we compromised on a purely functional kernel in a pedagogical exposition [4]. It also divided host languages into the macro language "haves" and macro-less "have nots". Here, we bridge some of that divide by re-implementing parts of the kernel with some modest runtime features common to most languages.

In this paper we:

- show how to functionally implement more general logical operators, cleanly obviating some of the surface macros
- survey, the design space of purely functional implementation alternatives for the remaining macros in *The Reasoned Schemer, 2nd Ed*'s [4] core language implementation, and weigh the trade offs and real-world consequences
- suggest practical solutions for completely eliminating the macros in those places where the pure microKanren functional implementations had seemed impractical

This exercise 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. Our re-implementation's source code for and the source for our experimental results is available at https://github.com/jasonhemann/tfp-2023/.

In Section 2, we illustrate by example what made surface syntax macros feel practically mandatory. 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 some outstanding questions on performance impacts

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

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

of these implementation choices, and consider how Kanren language implementers outside of the Scheme family might benefit from these alternatives.

2 All aboard!

We assume the reader is familiar with the miniKanren implementation of *TRS2e*. Although based on microKanren, this 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].

The Carmelit in Haifa is the world's shortest subway system with only six stations on its line: an example sufficiently small that modeling it should be painless. But in microKanren, to model the order a rider travels past the stops riding that subway end to end requires 11 logical operator nodes, because microKanren only provides *binary* conjunctions and disjunctions. (Listing 6 contains this paper's alternative solution, requiring just 3.) For a logic programming language, solely binary logical operators is too low level. To our eyes, this makes the superficial syntax macros practically mandatory, and host languages without a macro system are out of luck.

Moreover, the microKanren language doesn't offer the programmer sufficient guidance in using that fine-grained control. For a series of n goals, the programmer can associate them 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.

The core *TRS2e* language implementation relies on macros `fresh`, `defrel`, and `run` to introduce new logic variables, globally define relations, and execute queries. Earlier implementations of those same behaviors via pure functional shallow embeddings, without macros, had some harsh consequences. We will revisit those earlier implementations and their trade-offs, survey the landscape of available choices, and suggest performant compromises for those truly without macros, thus increasing microKanren's *practical* portability.

3 `disj` and `conj` logical goal constructors

microKanren's binary `disj₂` and `conj₂` operators are goal combinators: they each take two goals, and produce a new goal. Disjunction and conjunction work slightly differently. A *goal* is an outcome the program attempts to achieve, a goal can fail or succeed (and it can succeed many 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 is a state that results from achieving that goal in the given state. The `$append` function used in `disj₂` is an internal kernel primitive that combines two streams into one, with an interleave mechanism to prevent starvation; the result is a stream of the ways to achieve the two goals' disjunction. The `$append-map` function used in `conj₂` is to `$append` what the standard `append-map` is to `append`. The ways to achieve the conjunction of two goals are all the ways to achieve the second goal in a state itself the result of achieving the first goal. `$append-map` runs the second goal over the stream of results from the first goal, and combines the results of mapping together into a single stream representing the conjunction of the two, again with special attention to interleaving and starvation.

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

(define ((conj₂ g₁ g₂) s)
  ($append-map g₂ (g₁ s)))
```

Listing 1. microKanren `disj₂` and `conj₂`

We want to implement disjunction and conjunction over arbitrary quantities of goals, as functions. These implementations should subsume the binary `disj₂` and `conj₂` and they also should not use `apply`. Further, they should not build any extraneous closures: unnecessarily building closures at runtime is always a bad idea. 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 hence can use this paper's contributions.

Listing 2 shows our new versions, implemented as shallow wrappers over simple folds. In each, the first steps are to dispense with the trivial case, and then to call a recursive help function that makes no use of variadic functions. That is, all of our focus will be on the recurring over the list `gs`. Unlike `D`, the function `C` does not take in the state `s`; the help procedure does not need the state for conjunction. In each recursive call, we accumulate by mapping (using that special delaying implementation of `append-map∞` for Kanren-language streams) the next goal in the list. This left-fold implementation of conjunctions therefore left-associates the conjuncts.

3.1 Deriving semantic equivalents

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`.

```
(define ((disj . gs) s)
  (cond
    ((null? gs) (list))
    (else (D ((car gs) s) (cdr gs) s))))

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

(define ((conj . gs) s)
  (cond
    ((null? gs) (list s))
    (else (C (cdr gs) ((car gs) s)))))

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

Listing 2. Final re-definitions of `disj` and `conj`

```
(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)))))

(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))))

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

Listing 3. Derivation of `disj` function definition

As this is not part of the microKanren language itself, 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 procedures, the third and fourth in definitions in Listing 3.

We can replace the call to `disj₂` in that version by its definition in terms of `append∞` and perform a trivial β-reduction. The explicit `s` argument suggests removing the call to `apply` and making `D` recursive. The result is the final version of `D` in Listing 3. The definition of `disj` remains unchanged from before. 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 2.

We can derive the definition of `conj` from Listing 2 via a similar process. Starting with the variadic function based on the macro in Listing 4, we first η-expand and split the definition. We next substitute for the definitions of `conj` and `conj₂`. Finally, since `C` only needs `s` to *build* the stream, we can assemble the stream on the way in—instead of passing in `g` and `s` separately, we pass in their combination as a stream. The function is tail recursive, we can change the signature in the one and only external call and the recursive call. The result, after adding the trivial base case to `conj`, is shown in Listing 2.

Both the functional and the macro based versions of Listing 4 use a left fold over the goals, whereas the versions of `disj` use a right fold. This is not an accident. Folklore suggests that left associating conjunctions tends to improve the performance of miniKanren's interleaving search. The authors know of no thorough algorithmic proof of such claims, but see for instance discussions and implementation in [10] for some of the related work so far. In **??**, we display the results of some micro benchmarks that suggest the same. We have generally, however, resorted to small step visualizations of the search tree to explain the performance impact. The authors believe it is worth considering if we can make an equally compelling argument for this preference through equational reasoning and comparing the implementations of functions.

The benefits of a left-fold over conjunctions becomes a little more obvious by comparison to a right-fold implementation after we η-expand, unfold to a recursive help function, substitute in the definition of `conj₂`, and β-reduce. From there, 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; we show the upshot of

```
(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))))))

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

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

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

Listing 4. Derivation of split `conj` function definition

this sequence in Listing 5. Since we know that any call to `append-map∞` we construct will always yield a result, the version in Listing 2 is tail recursive. The equivalent right-fold implementation needs to somehow construct a closure for every recursive call. Building these superfluous closures is expensive. The same closure stacking behavior appears in the right fold variant of `disj`. Basic programming horse sense suggests the more elegant variants from Listing 2.

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 6, we implement `carmelit-subway` as an example,

```
(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∞))))
```

Listing 5. A right-fold variant of `conj` after some derivations

and it reads much better than the 11 binary logical operator nodes the programmer would have needed to write in a microKanren language without macros.

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

4 Tidying up the Impure Operators

The miniKanren `conda` operator that provides nested "if-then-else" behavior relies on the microKanren `ifte` underlying it. The definition of `conda` requires one or more conjuncts per clause and one or more clauses. The last line of `conda` contains the only place in the implementation that relies structurally on permitting nullary conjunctions, or disjunctions, of goals. Everywhere else conjunctions are one-or-more, and this one structural dependency is off-putting. There is a temptation to rewrite the second pattern in miniKanren's `conda` to demand *two* or more goals in each if-then clause and removing the dependency.

Some microKanren programmers without macros would be perfectly satisfied just using `ifte` directly, especially so given the research community's focus on purely relational

```
(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∞)))))))

(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 7. A typical macro implementation of `conda` and functional reimplementation

programming. But just as the standard forked **`if`** led to McCarthy's **`if`** notation and **`cond`**, a programmer may eventually feel the need for a nested implementation. The lower portion of Listing 7 contains a functional implementation of that cascade behavior. It includes the delay-and-restart behavior of `ifte` together with `conda`'s logical cascade. The `s∞` can be either empty, non-empty, or a function of no arguments. In the last case, we invoke `s∞`. Rather than building a largely redundant implementation of `condu`, we expose the higher-order goal `once` to the user. The definition of `once` in Listing 8 is taken directly from [4]. The programmer can simulate `condu` by wrapping `once` around every test goal.

5 Remainders and Practicalities

The 2013 microKanren paper demonstrates how to implement a purely functional Kanren language in an eager host. In Listing 9 we display these alternative mechanisms for introducing fresh logic variables, executing queries, and introducing delay and interleave. The versions in Listing 9 are slightly adjusted to be consistent with this presentation.

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

Listing 8. The `once` function

```
(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 init-state)))))

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

Listing 9. Functional microKanren equivalents of *TRS2e* kernel macros

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 offer some thoughts on the choice.

logic variables. Many of the choices for these last option 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 of variables, you may also need additional functions to support your implementation of variables. In a shallow embedding, designating some set as logic variables means either using a **`struct`**-like mechanism to construct a bespoke datatype hidden from the end user, or to arrogate 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 implicitly removes numbers, as data, from the user's term language and uses the natural numbers directly as an indexed set of variables.

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` 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 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.

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.

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

Listing 10. Queries as expressed with global-state variables

See sokuza-kanren [9] for an example of this style. With this latter approach, however, we can expose `var` directly to the programmer and the programmer can use **`let`** bindings to introduce several logic variables simultaneously.

run. Listing 9 also shows how we have implemented a `run`-like behavior without using macros. Using the purely-functional implementation of logic variables, a function like `call/initial-state` can do the job of `run` and `run*`. 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* actual same logic variable into the query, and also to 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 Listing 10, or create a special variable introduction mechanism for the first variable, scoped over both the query and the answer projection.

define. The microKanren programmer can just use their host language's **`define`** feature to construct relations as host-language functions, and manually introduce the delays in relations using a help function like `Zzz` 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 would 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`. That can be a subtle mistake

to debug. Those implementing shallow streams-based implementation in an eager host language may have no other choice, however.

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 working without macros that may be more practical than those of Hemann and Friedman.

Forcing ourselves to define `disj` and `conj` functionally, and with the restrictions we placed on ourselves in this reimplementation, removed a degree of implementation freedom and led us to what seems like the right solution. The result is closer to the design of Prolog, where the user represents conjunction of goals in the body of a clause with a comma and disjunction, either implicitly in listing various clauses or explicitly with a semicolon. 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.

Existing techniques for implementing `defrel`, `fresh` and `run` (and `run*`) without macros have with serious drawbacks. These include exposing the implementation of streams and delays, and the inefficiency and clumsiness of introducing variables one at a time, or the need to reason with global state. With a few more runtime features from the host language, an implementer can overcome some of those drawbacks, and may find one the suggested solutions 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 helps narrow the gap between implementations in functional host languages with and without macro systems and helps implementers build more elegant, expressive and efficient microKanrens in their chosen host languages.

Acknowledgments

Thanks to Ken Shan and Jeremy Siek, for helpful discussions and debates during design decision deliberations. Thanks also to Greg Rosenblatt and Michael Ballantyne for their insights and suggestions. We would also like to thank our anonymous reviewers for their insightful contributions.

References

[1] Michael Ballantyne, Alexis King, and Matthias Felleisen. "Macros for domain-specific languages." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[2] Ryan Culpepper. "Fortifying macros." In: *Journal of functional programming* 22.4-5 (2012), pp. 439–476.

[3] Matthias Felleisen. *Re: Question about tail recursion.* 2014. URL: https://lists.racket-lang.org/users/archive/2014-August/063844.html.

[4] Daniel P. Friedman et al. *The Reasoned Schemer, Second Edition.* The MIT Press, Mar. 2018. ISBN: 0-262-53551-3. URL: mitpress . mit . edu / books / reasoned - schemer-0.

[5] 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.