μKanren: A Minimal Functional Core for Relational Programming

Jason Hemann   Daniel P. Friedman

Indiana University

{jhemann,dfried}@cs.indiana.edu

Abstract

This paper presents μKanren, a minimalist language in the miniKanren family of relational (logic) programming languages. Its implementation comprises fewer than 40 lines of Scheme. We motivate the need for a minimalist miniKanren language, and iteratively develop a complete search strategy. Finally, we demonstrate that through sufficient user-level features one regains much of the expressiveness of other miniKanren languages. In our opinion its brevity and simple semantics make μKanren uniquely elegant.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages, Constraint and logic languages

Keywords miniKanren, relational programming, logic programming, Scheme

1. Introduction

miniKanren is the principal member of an eponymous family of relational (logic) programming languages. Many of its critical design decisions are a reaction to those of Prolog and other well-known 5th-generation languages. One of the differences is that, while a typical Prolog implementation might be thousands of lines of C code, a miniKanren language is usually implemented in somewhere under 1000 lines. Though there are miniKanren languages of varied sizes and feature sets (see `http://miniKanren.org`), the original published implementation was 265 lines of Scheme code. In those few lines, it provides an expressiveness comparable to that of an implementation of a pure subset of Prolog.

We argue, though, that deeply buried within that 265-line miniKanren implementation is a small, beautiful, relational programming language seeking to get out. We believe μKanren is that language. By minimizing the operators to those strictly necessary to do relational programming and placing much of the interface directly under the user's control, we have further simplified the implementation and illuminated the role and interrelationships of the remaining components. By making the implementation entirely functional and devoid of macros, heretofore opaque sections of the system's internals are made manifest. What is more, by re-adjudicating what functions are properly the purview of the end user, we develop an implementation that weighs in at 39 lines of Scheme. The presentation of the language and its features follows. The complete implementation of μKanren is found in the appendix.

2. The μKanren Language

Herein, we briefly describe the syntax of μKanren programs, with a focus on those areas in which μKanren differs from earlier miniKanren languages. Readers intimately familiar with miniKanren programming may lightly peruse this section; readers seeking a thorough introduction to miniKanren programming are directed to Byrd [3] and Friedman et. al [4], from which the present discussion is adapted.

A μKanren program proceeds through the application of a *goal* to a *state*. Goals are often understood by analogy to predicates. Whereas the application of a predicate to an element of its domain can be either true or false, a goal pursued in a given state can either *succeed* or *fail*. A goal's success may result in a sequence of (enlarged) states, which we term a *stream* [12]. We use functions to simulate relations. An arbitrary n-ary relation is viewed as an $(n-1)$-ary partial function, mapping tuples of domain elements into a linearized submultiset of elements of the codomain over which the initial relation holds. A given collection of goals may be satisfied by zero or more states. The result of a μKanren program is a stream of satisfying states. The stream may be finite or infinite, as there may be finite or infinitely many satisfying states.

A program's resulting stream thus depends on the goals that comprise that program. In μKanren there are four primitive goal constructors: $\equiv$, `call/fresh`, `disj`, and `conj`. The $\equiv$ goal constructor is the primary workhorse of the system; goals constructed from $\equiv$ succeed when the two arguments *unify* [1]. The success of goals built from $\equiv$ may cause the state to grow. Unlike the implementation of $\equiv$ detailed in Friedman et. al [4], that presented here does not prohibit circularities in the substitution.

The `call/fresh` goal constructor creates a fresh (new) logic variable. `call/fresh`'s sole argument is a unary function whose binding variable is a fresh logic variable and whose body is an expression over which the fresh variable's binding is scoped and which evaluates to a goal.

The `disj` and `conj` goal constructors express a sort of boolean logic over goals—they represent the binary disjunction and conjunction of goals, respectively. That is, both take two goals as arguments. A goal constructed from `disj` returns a non-empty stream if either of its two arguments are successful, and a goal constructed from `conj` returns a non-empty stream if the second argument can be achieved in the stream generated by the first. Both the names and implementations of `disj` and `conj` are inspired by those forms in Sokuza Kanren [6].

A state is a pair of a substitution (represented as an association list) and a non-negative integer representing a fresh-variable counter. In μKanren, as in most miniKanren languages, we use triangular substitutions [1]. In a μKanren program, the first variable will always be `#(0)` (a vector whose only element is 0), although its binding will not always be the initial entry in the substitution.

The user of the system begins a program by applying a goal to a state. That goal may be arbitrarily complicated. Most frequently the initial goal is created from a `call/fresh`. The value of applying that goal to a state is a stream. A stream is conceptually a possibly-unbounded list of states. While in principle the user of the system may begin with any state, in practice the user almost always begins with `empty-state`. `empty-state` is a user-level alias for a state virtually devoid of information: the substitution is empty, and the first variable will be indexed at 0. Consider the below example μKanren query.

```
> (define empty-state '(() . 0))
> ((call/fresh (λ (q) (≡ q 5))) empty-state)
((((#(0) . 5)) . 1))
```

In the invocation presented above, the result is the value of the goal `(call/fresh (λ (q) (≡ q 5)))` in the `empty-state`. This is a stream. Here this result is a stream of exactly one resulting state. In this state, the substitution binds a single variable `#(0)` to 5. The `1` denotes the other part of the state, the variable counter: here 1 is the next available variable index for the next `call/fresh`. The cdr is `()`, the empty stream, which means there are no further results. μKanren programs, in general, are not guaranteed to terminate, and so may not return with a result. A second example demonstrates the use of the remaining primitive goal constructors.

```
> (define a-and-b
    (conj
      (call/fresh (λ (a) (≡ a 7)))
      (call/fresh (λ (b) (disj (≡ b 5) (≡ b 6))))))
> (a-and-b empty-state)
((((#(1) . 5) (#(0) . 7)) . 2)
 (((#(1) . 6) (#(0) . 7)) . 2))
```

In this example, the outermost goal of the query is not built from `call/fresh`. Instead it is a conjunction of queries. The resulting stream contains two results, which represent ways to jointly satisfy both queries. The first result produced is the state whose substitution represents the unification of the first variable with the number 7, and the second with 5; the next state's substitution is that representing still the unification of the first variable with 7, but the second variable this time with 6. The user may wish to view these results with respect to the first variable, the second variable, both, or neither—μKanren leaves that decision to the user.

This example also demonstrates a μKanren relation globally defined as one might an ordinary Scheme function. As μKanren is embedded in Scheme, users may define goals and specify relations in Scheme and use them in the execution of a μKanren program.

3. Design Philosophy

Though the μKanren design draws a great deal of inspiration from that of Friedman et. al [4], certain philosophical concerns mandate a number of significant changes to the design and interface of the system. We first describe those motivations and with them in mind step through the implementation.

Though the μKanren implementation presented herein is short, and we consider its size a virtue, it isn't short for brevity's sake. Designing a pure and simple functional program will bring essentials to the fore. This entails stripping away interface components that, while desirable, properly ought to be the purview of the user. Doing so helps to crystallize the architecture of a miniKanren-like language's kernel.

When attempting to understand the implementation of a typical miniKanren language, the scope and complexity of the system can be overwhelming, for students and seasoned developers alike. Even if μKanren isn't the language in which one will program, understanding the internals of this implementation will clarify the inner workings of miniKanren languages generally. Too, we argue by example that μKanren may provide a more straightforward platform on which to add features than a monolithic architecture.

The adjective monolithic and the name μKanren are chosen advisedly. One of the design metaphors of μKanren is that of a microkernel operating system. A microkernel consists of the minimum or near-minimum amount of software required to implement an operating system [2]. In addition to security assurances, microkernels, by virtue of their small size and limited scope, tend to be easier to maintain, easier to keep free of bugs, and easier to understand. To borrow another metaphor, this time from software engineering, minimizing of the responsibilities of the kernel and pushing all that can be into user space loosens coupling and increases the cohesion of the system, and helps eliminate inappropriate intimacy among submodules. The resulting system is by design more clearly correct, with fewer pieces to consider simultaneously and a clear delineation of responsibilities.

4. µKanren Implementation

The entire system is comprised of a handful of functions for the implementation of variables, streams themselves, the interface for creating and combining streams, and four primitive goal constructors.

Variables themselves are represented as vectors that hold their variable index. Variable equality is determined by coincidence of indices in vectors.

```
(define (var c) (vector c))
(define (var? x) (vector? x))
(define (var=? x1 x2) (= (vector-ref x1 0) (vector-ref x2 0)))
```

The `walk` operator searches for a variable's value in the substitution; the `ext-s` operator extends the substitution with a new binding. When a non-variable term is walked, the term itself is returned. When extending the substitution, the first argument is always a variable, and the second is an arbitrary term. In Friedman et. al [4], `ext-s` performs a check for circularities in the substitution; here there is no such prohibition.

```
(define (walk u s)
  (let ((pr (and (var? u) (assp (λ (v) (var=? u v)) s))))
    (if pr (walk (cdr pr) s) u)))

(define (ext-s x v s) `((,x . ,v) . ,s))
```

The first of the four basic goal constructors, ≡, takes two terms as arguments and returns a goal[1] that succeeds if those two terms *unify* in the received state. If they unify, a substitution, possibly extended, is returned. In this case, ≡ passes this new substitution, paired with the variable counter to comprise a state, to `unit`. `unit` lifts the state into a stream whose only element is that state. If those two terms fail to unify in that state, the empty stream, `mzero`, is instead returned.

```
(define (≡ u v)
  (λg (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))

(define (unit s/c) (cons s/c mzero))
(define mzero '())
```

Terms of the language are defined by the `unify` operator. Here, terms of the language consist of variables, objects deemed identical under `eqv?`, and pairs of the foregoing. To unify two terms in a substitution, both are walked in that substitution. If the two terms walk to the same variable, the original substitution is returned unchanged. When one of the two terms walks to a variable, the substitution is extended, binding the variable to which that term walks with the value to which the other term walks. If both terms walk to pairs, the cars and then cdrs are unified recursively, succeeding if unification succeeds in the one and then the other. Finally, non-variable, non-pair terms unify if they are identical under `eqv?`, and unification fails otherwise. The definition of `unify` and the terms of the language are orthogonal to the presentation of µKanren: both could be changed with limited consequence for the rest of the system.

```
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      (else (and (eqv? u v) s)))))
```

The second basic goal constructor is `call/fresh`. The `call/fresh` goal constructor takes a unary function `f` whose body is a goal, and itself returns a goal. This returned goal, when provided a state `s/c`, binds the formal parameter of `f` to a new logic variable (built with the variable constructor operator `var`), and passes a state, with the substitution it originally received and a newly incremented fresh-variable counter, `c`, to the goal that is the body of `f`.

```
(define (call/fresh f)
  (λg (s/c)
    (let ((c (cdr s/c)))
      ((f (var c)) `(,(car s/c) . ,(+ c 1))))))
```

The final two basic goal constructors are `disj` and `conj`. The `disj` goal constructor takes two goals as arguments and returns a goal that succeeds if either of the two subgoals succeed. The `conj` goal constructor similarly takes two goals as arguments and returns a goal that succeeds if *both* goals succeed for that state.

```
(define (disj g1 g2) (λg (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (λg (s/c) (bind (g1 s/c) g2)))
```

These are implemented in terms of the `mplus` and `bind` operators, respectively[2]. It is through these two operators that the search strategy of µKanren is encoded. We iteratively develop versions of these operators, which in their final form will provide a complete search.

We begin with an implementation that is correct for finite relations. The addition of two lines yields the capacity to return streams of infinitely-many results. A slight change to this implementation results in a breadth-first search strategy typical of miniKanren implementations.

4.1 Finite Depth-First Search

The `mplus` operator is responsible for merging streams. In a goal constructed from `disj`, the resulting stream contains the states that result from success of either of the two goals. `mplus` simply appends the list returned as the result of the first call to that returned as the result of the second. In this form it is simply an implementation of `append`.

[1] We use the names λ_g and $\lambda_\$$ (formerly λ_f) to convey information to the reader and for consistency with other miniKanren implementations. The *\$* in $\lambda_\$$ is for "stream" and the *g* in λ_g is for "goal". In the code both are written `lambda`.

[2] For those who perfer to think in terms of monads, these definitions of bind, and mplus, together with the implementation of unit and mzero, comprise something akin to the list monad.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    (else (cons (car $1) (mplus (cdr $1) $2)))))
```

As a consequence, if the invocation of either of the two goals on the state results in an infinite stream, the invocation of `disj` will not complete (because Scheme is call-by-value).

`bind` is required in the implementation of `conj`.

```
(define (bind $ g)
  (cond
    ((null? $) mzero)
    (else (mplus (g (car $)) (bind (cdr $) g)))))
```

When a goal constructed from `conj` is invoked, the goal g$_1$ is invoked on the current state; the result of this invocation is a finite stream (represented as a proper finite list). The resulting stream is by definition finite, as those which spawn an infinite number of results fail to return. `bind` receives this resulting stream and the goal g$_2$. In `bind` that goal (g) is invoked on each element of the stream. If the stream of results of g$_1$ is empty or becomes exhausted `mzero`, the empty stream, is returned. If instead the stream contains a state and potentially more, then g is invoked on the first state. The stream which is the result of that invocation is merged to a stream containing the invocation of the rest of the $ passed in the second goal g. The results of these invocations in bind form the result of the `conj`. The `bind` operator is essentially an implemementation of `append-map` [10], though with its arguments reversed.

The search strategy these operators implement is a depth-first search strategy limited to a finite search space. The implementation here is similar in many respects to that of Sokuza Kanren [6]. Limiting queries to a finite search space, however, is an unfortunate and severe restriction. Consider the definition of the goal constructor `fives` below and the query that follows it.

```
> (define (fives x) (disj (≡ x 5) (fives x)))
> ((call/fresh fives) empty-state)
```

This query will fail to terminate, as the call to `disj` will invoke `mplus` to collect all results and returns them as a list. For an infinite relation, such as `fives` above, collecting all the results before returning any of them ensures no results are returned. A slightly more sophisticated approach to combining streams yields an ability to return streams of infinitely many results.

4.2 Infinite Streams

To remedy this limitation, `mplus` and `bind` must be augmented with an ability to return a stream without computing the answers it contains. These we term *immature streams*. When returning a finite number of infinitely many results, the result is a partially computed stream, with an immature stream in the cdr of the last pair.

A μKanren stream is then defined to be either the empty stream, an immature stream, or a *mature stream*, a pair of a state and a stream, which together comprise the stream's first and remaining states.

The redefinition of `mplus` below and the introduction of $\lambda_\$$ both warrant explanation. The changes from the previous definition are highlighted.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (λ$() (mplus ($1) $2)))
    (else (cons (car $1) (mplus (cdr $1) $2)))))
```

Immature streams are implemented as $\lambda_\$$s. The added `procedure?` line matches immature streams. The invocation of a goal on a state still yields a stream of results, now potentially infinite. In the case that the invocation of a goal g returns a non-empty stream, it must be either an immature stream, or a pair consisting of some state and a stream (either mature or immature). When the cdr of $\$_1$ (in the third `cond` clause of `mplus`) is an immature stream, the recursive invocation of `mplus` will match the second `cond` clause. This then returns a $\lambda_\$$ which, when invoked, will continue the search for more results in that stream. The $\lambda_\$$ is required to ensure termination at each invocation.

The `bind` operator is modified similarly, to correspond with the change to `mplus`. Since streams may also now be procedures, the `procedure?` line is added to account for that. Like in `mplus`, the $\lambda_\$$ is wrapped around the right-hand side to ensure termination.

```
(define (bind $ g)
  (cond
    ((null? $) mzero)
    ((procedure? $) (λ$() (bind ($) g)))
    (else (mplus (g (car $)) (bind (cdr $) g)))))
```

Making use of this capacity requires intervention on the part of the user, as in the following example. Consider again the definition of `fives` from Section 4.1. Though we now have the ability to return streams of infinitely many results, with this definition the query will still fail to terminate because of Scheme's call-by-value semantics. If `fives` is instead defined as below, the query will instead return.

```
(define (fives x)
  (disj (≡ x 5) (λg (s/c) (λ$() ((fives x) s/c)))))
```

The act of performing an inverse-η on a goal and then wrapping its body in a $\lambda_\$$ we refer to as inverse-η-delay. Invers-η-delay is an operation that takes a goal and returns a goal, as the result of doing so on any goal g is a function from a state to a stream.

$$g \quad \Rightarrow_{\eta^{-1}-delay} \quad \begin{array}{l}(\lambda_g\ (s/c)\\ \quad(\lambda_\$\ ()\\ \quad\quad(g\ s/c)))\end{array}$$

In order to ensure termination, when designing potentially non-terminating goal-constructors, such as (mutually) recursively-defined goal constructors, serious calls in the body of such goal-constructors must be inverse-η-delayed in order for the μKanren search to properly terminate. Inducing a delay may also be of use in relations which are finite but multitudinous. Unlike in miniKanren, properly inverse-η-delaying serious goals is the responsibility of the user.

As a consequence of this change, executions of µKanren programs can now also return an immature stream as part of a result. The user may manually advance the computation by invoking the immature stream, or more likely, invoke a user-level operator to advance a stream until it matures, as described in Section 5.

Since the basic goals and every goal constructed by one of the basic goal constructors are guaranteed to terminate, and since we require that every serious call be inverse-η-delayed, this seems to be a wholly satisfactory way to search for results.

4.3 Interleaving Streams

Though µKanren will return with a stream, the stream may be such that it will contain results which in principle can never be returned. Depth-first search over infinite relations is an incomplete search strategy. Consider the relation `sixes` and the associated query below.

```
> (define (sixes x)
    (disj (≡ x 6) (λg(s/c) (λ$() ((sixes x) s/c)))))
> (define fives-and-sixes
    (call/fresh (λ (x) (disj (fives x) (sixes x)))))
> (fives-and-sixes empty-state)
```

In this call, all answers returned will be states with the first variable bound to 5. This is despite the fact that another branch also containing satisfying results is waiting in the wings. In order to achieve a wider spread of answers, it is important that a mature stream be able to yield control.

One method to accomplish the hand-off of control between two or more procedures is a *binary trampoline*. A trampoline [5] provides the ability to share control between procedures. In a binary trampoline, two procedures each take a "step"—a bounded amount of computation—and then yield control to the other. In the case of the binary trampoline this "step–handoff–step–handoff" sequence continues until one of the two produces a value. Implementing a trampoline through `mplus` will cause every path in the search to terminate after a bounded amount of work—a single "bounce" on the trampoline—and then yield control to the next waiting possible direction of the computation. The addition of a binary trampoline is accomplished by a small change to the definition of `mplus`.

Consider again the definition of `mplus` above. When the search starts upon a path, it has no way to deviate until it exhausts a stream. If instead, the search along a particular path is mandated to hand-off control after a bounded amount of work, then no one branch will monopolize control, and the search will then be complete.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (λ$() (mplus $2 ($1))))
    (else (cons (car $1) (mplus (cdr $1) $2)))))
```

With the change highlighted here, we achieve a complete search strategy. Where before we would have returned an immature stream which, when invoked, would continue the search as it was to be invoked, we now instead hand-off control to the waiting stream, while also taking a step with the one which had previously been in control. As every finite stream will eventually be exhausted, and the user will inverse-η-delay invocations of recursive goals, this change means that no branch can continue to monopolize the search, and thus this version of `mplus` implements a complete search strategy.

The complete µKanren search strategy amounts to something on the order of a tree of trampolines. Results emerge ordered roughly by the number of trampoline bounces required to find them in a search tree, and approximates performing the search in parallel. Another analogy is that of a juggler juggling jugglers, where each juggler is an `mplus`, and the act of juggling is represented by the handoff of control in the second `cond` line of `mplus`. It is the search strategy of almost all miniKanren implementations, and until recently [11] it was at least the default choice of all published miniKanren implementations.

5. User-level Functionality

Although, as demonstrated above, one *can* program in the core µKanren language, a user may rightly desire a more sophisticated set of tools with which to program and through which to view the results. A sample set of such tools is provided. These can certainly be further augmented to suit a particular user's needs. Indeed, we believe it is a virtue of the µKanren approach that the user can readily and easily augment this system to regain much of the functionality of a standard miniKanren implementation.

5.1 Recovering miniKanren's control operators

To begin with, manually performing the inverse-η-delay can quickly become tedious. The user can instead employ the below macro `Zzz` (pronounced "snooze") to relieve some of the tedium.

```
(define-syntax Zzz
  (syntax-rules ()
    ((_ g) (λg(s/c) (λ$() (g s/c))))))
```

Too, manually nesting calls to `conj` and `disj` can quickly grow tiresome. The macros `conj+` and `disj+` introduced below provide the `conj` and `disj` of one or more goals.

```
(define-syntax conj+
  (syntax-rules ()
    ((_ g) (Zzz g))
    ((_ g0 g ...) (conj (Zzz g0) (conj+ g ...)))))


(define-syntax disj+
  (syntax-rules ()
    ((_ g) (Zzz g))
    ((_ g0 g ...) (disj (Zzz g0) (disj+ g ...)))))
```

Although not strictly necessary, we choose to Zzz each of the goals. A user writing only in terms of conj+ and disj+ will not need to manually Zzz the goals. With the definition of Zzz provided this entails a delay around every goal, whereas strictly speaking it is only required around the invocation of serious goals. A more sophisticated implementation of Zzz that forgoes delaying goals constructed from ≡ and simply recurs on subgoals of goals constructed from disj, conj, and call/fresh would come closer to the ideal.

With conj+, disj+, and the basic goal constructors of μKanren, the conde and fresh forms familiar to miniKanren programmers can be introduced as straightforward macros as well.

```
(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) ...) (disj+ (conj+ g0 g ...) ...))))

(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g ...) (conj+ g0 g ...))
    ((_ (x0 x ...) g0 g ...)
     (call/fresh (λ (x0) (fresh (x ...) g0 g ...))))))
```

conde is simply the disj+ of a (non-empty) sequence of conj+s. The fresh of miniKanren, which introduces any number of fresh variables into scope, is built as a recursive macro using call/fresh and conj+.

The user, having augmented their system with these higher-level features, can feel free to program with them, and escape to μKanren for more fine-grained control when desired.

5.2 From Streams to Lists

Invoking an immature stream to return results needn't be performed manually. With an operator like pull below, this could instead be done automatically.

```
(define (pull $) (if (procedure? $) (pull ($)) $))
```

This will simply advance the stream until it matures. pull also returns () for the empty stream (i.e. mzero). This model, in which the user gets one result at a time from the system, is reminiscent of Prolog.

pull itself can be abstracted to operations that pull all results or pull the first n results.

```
(define (take-all $)
  (let (($ (pull $)))
    (if (null? $) '() (cons (car $) (take-all (cdr $))))))

(define (take n $)
  (if (zero? n) '()
    (let (($ (pull $)))
      (cond
        ((null? $) '())
        (else (cons (car $) (take (- n 1) (cdr $))))))))
```

take-all pulls all results from the stream, and take pulls the first n or as many results as the stream contains, whichever is least. The user is of course free to develop other abstractions of their own. In the implementation correct only for finite relations, the call to pull is unnecessary but benign, and in the later versions it ensures that the work of take and take-all always occurs over a mature stream.

5.3 Recovering Reification

Relational programming languages typically also introduce a means by which the user can *reify* the results. *Reification* is the process by which the user sloughs off information from the resulting stream in order to clarify the presentation of that which is desired. Reification, now in the user's domain, is as a consequence more flexible than in many miniKanren languages. By placing the reifier under the user's control, they are now no longer mandated to reify against the first variable in the system, or limited to reifying the result once. Now that the substitution can be bound and manipulated, the user can reify along any number of dimensions for different views of the data. They now also have the ability to choose how the substitution should be presented, including a choice between idempotent [7] or triangular views of the substitution. The Scheme operators null? and length can both be seen as trivial reifiers, for the user seeking to know simply if any results were found or how many. The sample reifier presented here, mK-reify, is typical of other miniKanren languages.

```
(define (mK-reify s/c*)
  (map reify-state/1st-var s/c*))

(define (reify-state/1st-var s/c)
  (let ((v (walk* (var 0) (car s/c))))
    (walk* v (reify-s v '()))))
```

The reifier here, mK-reify, reifies a list of states s/c* by reifying each state's substitution with respect to the first variable. The reify-s, reify-name, and walk* helpers are required for reify-state/1st-var.

```
(define (reify-s v s)
  (let ((v (walk v s)))
    (cond
      ((var? v)
       (let ((n (reify-name (length s))))
         (cons `(,v . ,n) s)))
      ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
      (else s))))

(define (reify-name n)
  (string→symbol
    (string-append "_" "." (number→string n))))

(define (walk* v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) v)
      ((pair? v) (cons (walk* (car v) s)
                       (walk* (cdr v) s)))
      (else v))))
```

5.4 Recovering the Interface to Scheme

The conventions for calling programs and viewing the results can be similarly refined. We introduced empty-state in Section 2 as an alias for a state devoid of information.

The definition is restated below. We introduce here as well `call/empty-state`, an operator that takes a goal as an argument and attempts that goal in the `empty-state`. This reduces slightly some of the boilerplate in invoking μKanren program.

```
(define empty-state '(() . 0))
(define (call/empty-state g) (g empty-state))
```

By making use of the above, the missing `run` and `run*` interfaces of miniKanren are also recoverable[3]. `run` and `run*` execute the provided goals, and call `take` or `take-all` as appropriate. Here, `mK-reify` is encoded into the `run` and `run*` macros; this could be replaced by a different reifier, or the definitions of `run` and `run*` could instead be paramaterized over the reifier and the choice left up to the ultimate user.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x ...) g0 g ...)
     (mK-reify (take n (call/empty-state
                         (fresh (x ...) g0 g ...)))))))

(define-syntax run*
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (mK-reify (take-all (call/empty-state
                          (fresh (x ...) g0 g ...)))))))
```

Via the above, the user regains much of the expressiveness of other miniKanren languages. These are but a few possible user-level additions to the language. The user might, for instance, wish for a `take-all/p`, that pulls all results that match a certain predicate.

6. The Road not Taken

Many enhancements to the functionality of the core language are also possible. In a language with a pointer equality test (e.g. `eq?`), such as Scheme or Lisp, it could be used instead of `var-eq?` to test for the identity of variables. This would dramatically decrease the cost of `walk`, though arguably at the cost of a "side-effect" in the implementation.

We might instead implement the substitution using a persistent hash-table [9] instead of a linked list. This would provide $O(1)$ lookup of variables in the substitution and eliminate the use of `assp` (or `assq`) in `walk`. In the core.logic implementation of miniKanren, Nolen [8] achieves significant performance gains by implementing the substitution as a persistent hash table.

Our implementation of `unify`, as mentioned in Section 4, allows circularities in the substitution. miniKanren languages typically prohibit this. Using `occurs✓` in the redefinition of `ext-s` below would make μKanren's behavior in this regard correspond with miniKanren's.

[3] The definitions of run and run∗ are not precisely those of miniKanren, as they allow 0 or more variables in the list of query variables. These definitions can be further constrainted to provide exactly miniKanren's behavior.

```
(define (occurs✓ x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (var=? v x))
      (else (and (pair? v) (or (occurs✓ x (car v) s)
                               (occurs✓ x (cdr v) s)))))))

(define (ext-s x v s)
  (if (occurs✓ x v s) #f `((,x . ,v) . ,s)))
```

In the previous definition of `mplus` in Section 4.3, swapping streams in the recursive call of the third `cond` clause in addition to the second would lead to a fairer distribution of answers over infinite searches, and would obviate much of the use for inverse-η-delay over finite searches.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (λ$() (mplus $2 ($1))))
    (else (cons (car $1) (mplus $2 (cdr $1))))))
```

The `conj` and `disj` goal constructors might also be defined differently, or equivalently, other core language primitives for the conjunction and disjunction of goals might be chosen in their stead. `conj+` and `disj+` (or `conj*` and `disj*`, variants that represent the conjunction (disjunction) of 0 or more goals) could replace `conj` and `disj` as primitives. A number of different implementations of each are possible; their precise definitions are left as exercises for the interested reader.

7. Conclusions and Future Work

In this paper, we present μKanren, a "featherweight" implementation of a pure relational (logic) programming language. μKanren is the smallest in the miniKanren family of languages. Its kernel is entirely functional, contains no macros, and comprises but 14 definitions and 39 lines of code. As such, we believe the implementation is both simpler to understand and more directly portable than that of other miniKanren languages.

As it is intended to be a bare-bones implementation, inevitably certain features don't make the cut. Absent are the impure operators of the original miniKanren, constraints beyond the substitution, and cutting-edge operators of the more expansive implementations. It does not come equipped with a reifier, and the user must take responsibility for Scheme infinite loops, and for inverse-η-delaying recursive goals. It is nevertheless sufficient to perform real relational programming tasks, and general enough for end-users to abstract its interface and build more powerful search tools.

μKanren provides what we believe is a reasonable first introduction to the internals of a Kanren-like language. Students, developers, and aspiring implementers alike may find μKanren a worthwhile object of study before diving into the internals of another implementation.

Too, the smaller and wholly functional core may in and of itself encourage further adoption of miniKanren. A strong hygienic macro system is absent from many major languages. A reference implementation that does not take advantage of those features may ease the burden of porting to such languages. In so doing μKanren may help make miniKanren-like languages the default choice for developers looking to roll their own relational programming languages.

In future research, we hope to gain more experience extending μKanren's feature set without compromising the elegance of its model. We hope to implement other search strategies in a similarly straightforward manner, and perhaps develop a mechanism to change out search strategies during the course of a search.

We believe that the language presented here expresses the notion of a pure relational programming language in a more concise manner than have earlier miniKanren implementations. The μKanren model—small, and conservative in its feature set but easily extensible—may also suggest another path forward for the miniKanren family of languages.

Acknowledgments

We extend our thanks to Will Byrd and Will Ness for their inspiration. We also thank Adam Foltzer and Andre Kulenschmidt for their comments and criticisms of earlier drafts. We are especially grateful to Chung-chieh Shan, Jeremy Siek, Cameron Swords, and Sam Tobin-Hochstadt for their assistance clarifying the presentation of the present work.

References

[1] F. Baader and W. Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[2] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, et al. Microkernel operating system architecture and Mach. *Journal of information processing*, 14(4):442–453, 1992.

[3] W. E. Byrd. *Relational programming in miniKanren: techniques, applications, and implementations.* PhD thesis, Indiana University, 2009.

[4] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer.* The MIT Press, July 2005. ISBN 0262562146.

[5] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proc. 4th ACM SIGPLAN international conference on Functional programming*, pages 18–27. ACM Press, 1999.

[6] O. Kiselyov. The taste of logic programming, 2006. URL `http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren`.

[7] J. W. Lloyd. *Foundations of Logic Programming, 1st Edition.* Springer, 1984. ISBN 3-540-13299-6.

[8] D. Nolen. core.logic, 2013. URL `https://github.com/clojure/core.logic`.

[9] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[10] O. Shivers. List Library. Scheme Request for Implementation. SRFI-1, 1999. URL `http://srfi.schemers.org/srfi-1/srfi-1.html`.

[11] C. Swords and D. Friedman. rKanren: Guided search in miniKanren. *Scheme and Functional Programming*, 2013.

[12] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

Appendix: μKanren Implementation

```
(define (var c) (vector c))
(define (var? x) (vector? x))
(define (var=? x1 x2) (= (vector-ref x1 0) (vector-ref x2 0)))

(define (walk u s)
  (let ((pr (and (var? u) (assp (λ (v) (var=? u v)) s))))
    (if pr (walk (cdr pr) s) u)))

(define (ext-s x v s) `((,x . ,v) . ,s))

(define (≡ u v)
  (λg (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))

(define (unit s/c) (cons s/c mzero))
(define mzero '())

(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      (else (and (eqv? u v) s)))))

(define (call/fresh f)
  (λg (s/c)
    (let ((c (cdr s/c)))
      ((f (var c)) `(,(car s/c) . ,(+ c 1))))))

(define (disj g1 g2) (λg (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (λg (s/c) (bind (g1 s/c) g2)))

(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (λ$ () (mplus $2 ($1))))
    (else (cons (car $1) (mplus (cdr $1) $2)))))

(define (bind $ g)
  (cond
    ((null? $) mzero)
    ((procedure? $) (λ$ () (bind ($) g)))
    (else (mplus (g (car $)) (bind (cdr $) g)))))
```