A Small Embedding of Logic Programming with a Simple Complete Search

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@indiana.edu

William E. Byrd Matthew Might
University of Utah
{Will.Byrd,might}@cs.utah.edu

Abstract

We present a straightforward, call-by-value embedding of a small logic programming language with a simple complete search. We construct the entire language in 54 lines of Racket—half of which implement unification. We then layer over it, in 43 lines, a reconstruction of an existing logic programming language, miniKanren, and attest to our implementation's pedagogical value. Evidence suggests our combination of expressiveness, concision, and elegance is compelling: since microKanren's release, it has spawned over 50 embeddings in over two dozen host languages, including Go, Haskell, Prolog and Smalltalk.

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

Keywords miniKanren, microKanren, logic programming, relational programming, streams, search, Racket

1. Introduction

Logic programming is a highly declarative approach to problem solving that has proven itself applicable to a wide variety of tasks [21]. Consider the below stylized Prolog definition of the append relation:

```
append(L,S,O) :- [] = L, S = O
              ; [A|D] = L, [A|R] = O, append(D,S,R).
```

We can use this one definition to solve a variety of problems.

```
(1) ?- append([t,u,v],[w,x],Q).
    Q = [t,u,v,w,x] ?

(2) ?- append([t,u,v],Q,[t,u,v,w,x]).
    Q = [w,x] ?
```

DLS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4445-6/16/11...$15.00
http://dx.doi.org/10.1145/2989225.2989230

```
(3) ?- Q = [L,S], append(L,S,[t,u,v,w,x]).
    L = [],
    Q = [[],[t,u,v,w,x]],
    S = [t,u,v,w,x] ?
    ...
    L = [t,u,v,w,x],
    Q = [[t,u,v,w,x],[]],
    S = [] ?
```

In (1), we ask "What are the possible results of appending `[t u v]` to `[w x]`?" In (2), we ask for a `Q` result which we can prepend to `[t u v]` to construct the full list `[t u v w x]`. In (3), we ask for a `Q` composed of terms `L` and `S` such that concatenating `L` and `S` yields `[t u v w x]`; we can find all such possibilities. Our definition of `append` also has many more uses.

The definition of `append` illustrates the sorts of things we can write in a logic programming language. Prolog is the traditional choice but we can easily transliterate this definition to miniKanren [11], a logic programming DSL shallowly embedded in many hosts, including Scheme and Racket [8].

```
(define-relation (append l s o)
  (conde
    ((== '() l) (== s o))
    ((fresh (a d)
       (== `(,a . ,d) l)
       (fresh (r)
         (== `(,a . ,r) o)
         (append d s r))))))
```

(We will explain the syntax in Section 5. The point is we can ask the same sorts of questions and get the same sorts of answers.) In miniKanren, queries are always executed via the `run` operator. Answers are always printed with respect to the *query variable*, here `q`, and are always returned as a list. We also provide a max limit to the number of answers, or `#f` for no limit.

```
(1) > (run #f (q) (append '(t u v) '(w x) q))
    '((t u v w x))

(2) > (run #f (q) (append '(t u v) q '(t u v w x)))
    '((w x))
```

```
(3) > (run #f (q) (fresh (l s)
                    (== `(,l ,s) q)
                    (append l s '(t u v w x))))
    '((() (t u v w x))
      ...
      ((t u v w x) ()))
```

This small example demonstrates the versatility of logic programs. The `append` relation is the `factorial` of logic programming: it gives a sense of the possibilities this style of programming allows. miniKanren researchers have used logic programming to create a Scheme interpreter that doubles as a quine generator, a theorem prover that doubles as a proof assistant, and a type checker that doubles as a type inhabiter, to name just a few [4, 5, 24]. In our experience, when people are first exposed to logic programming, things like `append` and especially the examples mentioned above look a little like magic.

It is difficult for the uninitiated to understand how logic programming works, or to ferret out the essential details from a language's implementation. An abundance of powerful, useful features combined with years—or even decades—worth of optimizations and improvements can obscure the more fundamental aspects of an implementation's inner workings. Even the small language implementations are rarely designed to be easily understood.

Previous miniKanren implementations, for instance, enmesh the macros that provide miniKanren's syntax with the execution of the logic program itself. This demands the reader have a detailed understanding of macros, and such tight coupling makes it difficult for aspiring implementers in languages without macro support. It is better to separate these concerns and allow functional programmers in a call-by-value language to implement the core logic programming features without the syntactic sugar.

Thus we present microKanren, a dead simple, side-effect free, embedding intended to clarify and explain the behavior of logic programming languages. Our work here comprises several meaningful contributions.

Improved interleaving search. miniKanren's interleaving depth-first search is based on Kiselyov et al.'s LogicT transformer [19]. These operators provide a complete search without the performance penalties associated with, for example, breadth-first search. miniKanren interleaves between successfully finding answers, with additional interleaving added to avoid starvation and to avoid problems associated with the eagerness of its host language. We combine the hand-off of control with relation definition, and in doing so decrease the amount of interleaving while maintaining a complete search. We achieve a minimal placement of interleaving points for arbitrary relation definitions.

Reconstructing an existing language. To the best of our knowledge, no other authors have reconstructed an existing language, used in practice, upon a similar logic-programming kernel. We reconstruct a full miniKanren over our pure, functional embedding in 43 lines of code. This brief and clear reconstruction explains how to move from a language sufficient in principle to one useful in practice. More practically, doing so led to a clearer, more easily ported implementation.

In addition to these technical offerings, we also provide a third less traditional but still meaningful contribution:

Pedagogical value. We find the process of building microKanren both enlightening and fun. These are both subjective judgments, but it seems like others agree. A preliminary (non-archival) draft of this paper [13] spawned more than 50 implementations[1] in more than two dozen languages. These host languages are a rather diverse set, including Go, Haskell, Prolog, Smalltalk, and miniKanren itself just to name a few. microKanren has been studied in academic and social paper-reading groups and been presented at several software development meet-ups and conferences, all without any connection to its authors. Both the presentation and implementation have been significantly improved from this earlier draft.

```
Goal     :: State → Stream
State    :: Subst × Nat
Stream   :: Mature | Immature
Mature   :: () | State × Stream
Immature :: Unit → Stream
```

Figure 1. microKanren Datatypes

2. Terminology

We begin by explaining a few fundamental terms, summarized in Figure 1 for later reference. A warning to the reader: we deliberately restrict ourselves to a handful of primitives to clarify the implementation. As a result the primitive constructor `cons` will have multiple uses; we highlight them when they appear. Also, characters like `-`, `$`, and `/` are used as, or in, valid identifiers.

Program. A microKanren program consists of zero or more *relations* (which will look similar to `append`) and an initial *goal* (which will look similar to the body of the `run` statements in the first set of examples). Invoking the first goal may require a call to some relation, which may itself require a call to another relation or relations, and so on.

Goals. Goals are implemented as functions that take a *state* and return a *stream* of states. They consist of primitive constraints like `(== x y)`, relation invocations like `(append 'x q '(x b c))`, and their closure under operators that perform conjunction, disjunction, and variable introduction.

[1] These implementations, including the present one, are collected at `http://miniKanren.org`.

State. We execute a program p by attempting an initial goal in the context of zero or more relations. The program proceeds by executing a goal in a *state*, which holds all the information accumulated in the execution of p. Most importantly, the state contains a *substitution*, the data structure that encodes the information necessary to satisfy the primitive constraints we've accumulated. The state also contains a *counter* for assigning unique identifiers to fresh variables. Every program's execution begins with an *initial state* devoid of any constraint information and a new variable count.

Streams. Executing a goal in a state s/c (connoting a pair of a substitution and a counter) yields a stream. A stream may take one of three shapes:

- *empty*: The stream may be empty, indicating that the goal cannot be achieved in s/c.
- *answer-bearing*: A stream may contain one or more resultant states. In this case, each element of the stream is a *different* way to achieve that goal from s/c. Here, we mean "different" in terms of control flow (i.e., disjunctions); the same state may occur many times in a single stream. Our streams are not necessarily infinite; there may be finitely many ways to achieve a goal in a given state. We call these first two shapes *mature*.
- *immature*: An immature stream is a delayed computation that will return a stream when forced.

The final step of running a program is to continually force the resultant stream until it yields a list of answers. microKanren programs however, are not guaranteed to terminate. The stream we get from invoking the initial goal may be *unproductive*: repeated applications of `force` will never produce an answer [28]. This is the only potential cause of non-termination; all of the other core operations in our implementation are total.

3. microKanren

We now implement microKanren's six basic operators, namely ==, `call/fresh`, `disj`, `conj`, `define-relation`, and `call/initial-state`.

3.1 ==

The == constraint demands syntactic equality between two terms `u` and `v`. For microKanren, terms consist of variables, symbols, booleans, the empty list, and `cons` pairs of the preceding (Figure 2). To keep our presentation concise, we will sometimes use operators besides `cons` to construct terms. In principle, all of our terms could be built with `cons` though. Given `u` and `v`, we return a function expecting `s/c`. We then extract the substitution `s`, the first element of the pair `s/c`.

```
#| Term × Term → Goal |#
(define ((== u v) s/c)
  (let ((s (car s/c)))
    ...))
```

```
x ∈ Var
s ∈ Symbol
b ∈ Bool

Term ::= x | s | b | () | (Term . Term)
```

Figure 2. The microKanren term language

A substitution is a data structure that carries the meanings of *logic* variables. We use these logic variables differently than we use the standard lexical variables of functional programming. Unlike an environment, a substitution may associate variables with almost *any* other microKanren term—including other unassociated variables. The present substitution may associate a variable `x` with a term containing an unassociated variable `y`, and thus giving an association to `y` may also impact the meaning of `x`. Adding an association of a term and a previously unassociated variable can impact the values of an unbounded quantity of other variables.

We will use "variables" as a shorthand for "logic variables." And when we mean lexical variables, we'll say so explicitly. We represent our variables as natural numbers. To check if a term is a variable, we ask if it is a number.

```
#| Nat → Var |#
(define (var n) n)

#| Term → Bool |#
(define (var? t) (number? t))
```

Having extracted the substitution, we next look up each of the two terms in the substitution via `find`.

```
#| Term × Term → Goal |#
(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      ...)))
```

The `find` procedure "dereferences" a variable in a substitution: it finds the variable's associated term. If `find`'s argument is a variable with an association in the substitution, we look up the variable and return the associated term. If `find` receives a non-variable term, or a variable without an association, it returns its input.

```
#| Term × Subst → Term |#
(define (find u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (find (cdr pr) s) u)))
```

We represent substitutions as association lists, and we rely on the primitive function `assv` to check if `u` is the first element of a pair in `s`. If so, it returns the pair, if not, `#f`. Unlike many other languages, Racket's `if` accepts any value as its first argument, and any value except `#f` is considered true enough (or "truthy"). If `assv` returns a pair, we "dereference" the second element of that pair, `u`'s association. Otherwise, we return `u`.

The remainder of =='s definition relies on unify. It implements standard first-order syntactic unification with an occurs-check, as is standard in the literature [25]. The unify function creates so-called "triangular" substitutions [1] — a given variable x that we have now solved for may occur in previously-bound terms and we do nothing to remove indirections that result from extensions to the substitution. Such definitions of substitution are especially amenable to structure-sharing and implementation with persistent data structures. This decision necessitates the recursion in find. The function unify returns either a (possibly extended) substitution or #f connoting that the terms fail to unify in the substitution.

```
#| Var × Term × Subst → Maybe Subst |#
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else (cons `(,x . ,v) s))))

#| Var × Term × Subst → Bool |#
(define (occurs? x u s)
  (cond
    ((var? u) (eqv? x u))
    ((pair? u) (or (occurs? x (find (car u) s) s)
                   (occurs? x (find (cdr u) s) s)))
    (else #f)))

#| Term × Term × Subst → Maybe Subst |#
(define (unify u v s)
  (cond
    ((eqv? u v) s)
    ((var? u) (ext-s u v s))
    ((var? v) (unify v u s))
    ((and (pair? u) (pair? v))
     (let ((s (unify (find (car u) s) (find (car v) s) s)))
       (and s (unify (find (cdr u) s) (find (cdr v) s) s))))
    (else #f)))
```

If unify returns a substitution, we create a state using the current counter, and make a stream with only that state. We use list to construct singleton streams, and quasiquote and unquote to construct states. If unify returns #f, we return (), the empty stream.

```
#| Term × Term → Goal |#
(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      (if s (list `(,s . ,(cdr s/c))) `()))))
```

The return value of == is a goal, and thus == is a *goal constructor*. We will see that call/fresh, conj, and disj are also goal constructors. The last microKanren operator, call/initial-state, is not a goal constructor. Instead, it executes a goal and yields a list of states. For the time being though, we can explicitly invoke our goals in the initial state.

With == as our only goal constructor, the result of invoking any goal with the initial state is a mature stream. In fact, it's a mature stream of length zero or one. Either the stream is empty, indicating that the two terms do not unify, or the stream is non-empty, indicating that they do.

```
> ((== '#t 'z) '(() . 0))
'()
> ((== '#t '#t) '(() . 0))
'((() . 0))
> ((== '(#t . #f) '(#t . #f)) '(() . 0))
'((() . 0))
```

For the moment, no matter what terms we unify, the resulting substitution will remain empty. In order to grow the substitution, we need to have some variables.

3.2 call/fresh

The call/fresh operator is a goal constructor that allows us to introduce variables. To this end its argument is a lambda expression expecting a variable and returning a goal. We use this lambda expression to bind our logic variable to a lexical variable scoped over the resulting goal. In microKanren, call/fresh's argument should always be a λ expression.

```
#| (Var → Goal) → Goal |#
(define ((call/fresh f) s/c)
  ...)
```

The counter in the state indicates the number of the next available variable. Invoking var creates a new variable. (f (var c)) evaluates to a goal.

```
#| (Var → Goal) → Goal |#
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) ...)))
```

The resultant goal is then invoked in a newly created state with the previous substitution and an incremented counter. By incrementing the counter, we guarantee that each new logic variable is distinct from the previous ones and has no prior binding in the substitution.

```
#| (Var → Goal) → Goal |#
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))
```

Since we can now build terms that contain logic variables, a successful unification can now extend the substitution with variable bindings. Even with just the operators call/fresh, and ==, we can construct goals that when invoked produce quite complicated answers, as demonstrated in the following example. The number of parentheses start to stack up, and will only increase as our programs get more complicated still.

```
> ((call/fresh
    (λ (x)
      (== x 'a)))
   '(() . 0))
'((((0 . a)) . 1))
```

We read the stream ((((0 . a)) . 1)) as follows: the stream contains exactly one state, (((0 . a)) . 1), which is made up of a substitution ((0 . a)) and a counter 1. This substitution contains exactly one association: the variable 0 with the term a.

3.3 `conj` and `disj`

Using only the operators above we can't write programs with more than one equality constraint. The binary operators `disj` and `conj` act as goal combinators, and they allow us to write composite goals representing the disjunction or conjunction of their arguments.

```
#| Goal × Goal → Goal |#
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))

#| Goal × Goal → Goal |#
(define ((conj g1 g2) s/c) ($append-map g2 (g1 s/c)))
```

We define `disj` and `conj` in terms of two other functions, `$append` and `$append-map`. These will be defined in Section 3.4. With the examples below, though, we demonstrate the use of `conj` and `disj` in combination with our other goal constructors.

```
> ((disj
     (call/fresh
       (λ (x)
         (== 'z x)))
     (call/fresh
       (λ (x)
         (== '(s z) x))))
   '(() . 0))
'((((0 . z)) . 1) (((0 . (s z))) . 1))
> ((call/fresh
     (λ (x)
       (call/fresh
         (λ (y)
           (conj
             (== y x)
             (== 'z x))))))
   '(() . 0))
'((((0 . z) (1 . 0)) . 2))
```

With these operators, our streams will always be empty or answer-bearing; in fact, they will be fully computed. The result of a goal constructed from == must be a finite list, of length 0 or 1. If both of `disj`'s arguments are goals that produce finite lists, then the result of invoking `$append` on those lists is itself a finite list. If both of `conj`'s arguments are goals that produce finite lists, then the result of invoking `$append-map` with a goal and a finite list must itself be a finite list. If `call/fresh`'s argument `f` is a function whose body is a goal, and that goal produces a finite list, then `(call/fresh f)` evaluates to such a goal.

Invoking a goal constructed from these operators in the initial state returns a list of all successful computations, computed in a depth-first, preorder traversal of the search tree generated by the program.

3.4 Recursion and `define-relation`

It's important that we enrich our implementation to allow recursive relations. Much of the power of logic programming comes from writing relations (e.g. `append`) that refer to themselves or one another in their definitions. At present there are several obstacles. Suppose we'd used `define` to build a function that we hope would behave like a relation:

```
(define (peano n)
  (disj
    (== n 'z)
    (call/fresh
      (λ (r)
        (conj
          (== n `(s ,r))
          (peano r))))))
```

This function purports to be a relation that holds for a particular encoding of Peano numbers. What happens when we use the `peano` relation in the program below? We're hoping to generate some Peano numbers.

```
> ((call/fresh
     (λ (n)
       (peano n)))
   '(() . 0))
```

We invoke `(call/fresh ...)` with an initial state. Invoking that goal creates and lexically binds a new fresh variable over the body. The body, `(peano n)`, evaluates to a goal that we pass the state `(() . 1)`. This goal is the disjunction of two subgoals. To evaluate the `disj`, we evaluate its two subgoals, and then call `$append` on the result. The first evaluates to `(((0 . z)) . 1)`, a list of one state.

Invoking the second of the `disj`'s subgoals however is troublesome. We again lexically scope a new variable, and invoke the goal in body with a new state, this time `(() . 2)`. The `conj` goal has two subgoals. To evaluate these, we run the first in the current state, which results in a stream. We then run the second of `conj`'s goals over each element of the resulting stream and return the result. Running this second goal begins the whole process over again. In a call-by-value host, this execution won't terminate. Simply using `define` in this manner will not suffice.

We instead introduce the `define-relation` operator. This allows us to write recursive relations; with a sequence of uses of `define-relation`, we can create mutually recursive relations. Unlike the other operators, `define-relation` is a macro:

```
(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))
```

Racket's `define-syntax-rule` gives a simple way to construct non-recursive macros. The first argument is a pattern that specifies how to invoke the macro. The macro's first symbol, `define-relation`, is the name of the macro we're defining. Its second argument is a template to be filled in with the appropriate pieces from the pattern. We do implement `define-relation` in terms of Racket's `define`.

This macro expands a name, arguments, and a goal expression to a `define` expression with the same name and number of arguments and whose body is a goal. It takes a state and returns a stream, but unlike the others we've seen before, this goal returns an immature stream. When given a state `s/c`, this goal returns a promise that evaluates the original goal g in the state s/c when forced, returning a stream. A promise that returns a stream is itself an immature stream.

define-relation does two useful things for us: it adds the relation name to the current namespace, and it ensures that the function implementing our relation is total. It turns out that we will *never* re-evaluate an immature stream. Unlike delay, delay/name doesn't *memoize* the result of forcing the promise, so it is like a "by name" variant of delay.

We implement define-relation as a macro of necessity. It is critical that the expression g not be evaluated prematurely: the objective is to delay the invocation of g in s/c. In a call-by-value language, a function would (prematurely) evaluate its argument and would not delay the computation.

Below, we revisit the peano example, but this time using define-relation. Non-termination of relation invocations is no longer an issue. Instead, the goal (peano n), when invoked, immediately returns an immature stream.

```
(define-relation (peano n)
  (disj
    (== n 'z)
    (call/fresh
      (λ (r)
        (conj
          (== n `(s ,r))
          (peano r))))))
```

We can also write recursive relations whose goals quite clearly will never produce answers.

```
(define-relation (unproductive n)
  (unproductive n))
```

We can now introduce $append and $append-map. Their definitions are in fact those of append and append-map, functions over lists that are standard to many languages [27], but augmented with support for immature streams.

```
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))
```

If the recursive argument to $append is an immature stream, we return an immature stream, which, when forced, continues appending the second to the first. Likewise, in $append-map, when $ is an immature stream, we return an immature stream that will continue the computation but still forcing the immature stream. Rather than delay/name, force, and promise?, we could have used (λ () ...), procedure invocation, and procedure?. Using λ to construct a procedure delays evaluation, and procedure? would be our test for an immature stream. We choose Racket's special-purpose primitives for added clarity, but implementers targeting other languages can use anonymous procedures if these primitives aren't available. In languages without macros, the programmer could explicitly add a delay at the top of each relation; this has though the unfortunate consequence of exposing the implementation of streams.

```
#| Goal × Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) '())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

After these changes, it's possible to execute a program and produce neither the empty stream nor an answer-bearing one. We might produce instead an immature stream.

```
> ((call/fresh
     (λ (n)
       (peano n)))
   '(() . 0))
#<promise>
```

To resolve this we need to do something special when we invoke a goal in the initial state.

3.5 call/initial-state

At the very least, we would like to know if our programs are *satisfiable* or not. That is, we would hope to get at least one answer if one exists, and the empty list if there are none. The call/initial-state operator ensures that if we return, we return with a list of answers.

```
#| Maybe Nat⁺ × Goal→ Mature |#
(define (call/initial-state n g)
  (take n (pull (g '(() . 0)))))
```

call/initial-state takes an argument n which represents the number of answers to retrieve. n may just be a positive natural number, in which case we return at most that many answers. Otherwise, we provide #f, indicating microKanren should return *all* answers. It also takes a goal as an argument. The function pull takes a stream as argument, and if pull terminates, it returns a mature stream. As streams may be unproductive, it is not always possible to produce a mature stream. As a result, pull, and consequently take and call/initial-state, are partial functions. These are the only partial functions in the microKanren implementation.

```
#| Stream → Mature |#
(define (pull $) (if (promise? $) (pull (force $)) $))
```

take receives the mature stream that is the result of pull and, n, the argument dictating whether to return all, or just the first n elements of the stream.

```
#| Maybe Nat⁺ × Mature → List |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))
```

Our microKanren is now capable of creating, combining, and searching for answers in infinite streams.

```
> (call/initial-state 2
    (call/fresh
      (λ (n)
        (peano n))))
'((((0 . z)) . 1) (((1 . z) (0 . (s 1))) . 2))
```

Thus, we have brought microKanren programs into the delay monad [6, 12]: rather than always returning a list implementation of non-deterministic choice, we either have no values, a value now (possibly more than one), or something we can search later for a value. `pull`, since it forces an actual value out of a promise, is akin to run in the delay monad. `take` bears a similar relationship to run in the list monad.

3.6 Interleaving, Completeness, and Search

Although microKanren is now capable of creating and managing infinite streams, it doesn't manage them as well as we'd like. Consider what happens in the following program execution:

```
> (call/initial-state 1
    (call/fresh
      (λ (n)
        (disj
          (unproductive n)
          (peano n)))))
```

We would like the program to return a stream containing the `n`s for which `unproductive` holds and in addition the `n`s for which `peano` holds. We know from Section 3.4 that there are no `n`s for which `unproductive` holds, but infinitely many for `peano`. The stream should contain only `n`s for which `peano` holds. It's perhaps surprising, then, to learn that this program loops infinitely.

Streams that result from using `unproductive` will always be, as the name suggests, unproductive. When executing the program above, such an unproductive stream will be the recursive argument `$1` to `$append`. Unproductive streams are necessarily immature. According to our definition of `$append`, we always return the immature stream. When we force this immature stream, it calls `$append` on the forced stream value of (the delayed) `$1` and `$2`. Since `unproductive` is unproductive, this process continues without ever returning any of the results from `peano`.

Such surprising results are not solely the consequence of goals with unproductive streams. Consider the definition of `church`.

```
(define-relation (church n)
  (call/fresh
    (λ (b)
      (conj
        (== n `(λ (s) (λ (z) ,b)))
        (peano b)))))
```

The relation `church` holds for Church numerals. Using a newly created variable `b`, it constructs a list resembling a lambda-calculus expression whose body is the variable `b`. It uses `peano` to generate the body of the numeral. We can thus use it to generate Church numerals in a manner analogous to our use of `peano`. But consider the program below, wherein the resulting stream is productive, but only contains elements for which `peano` holds.

```
> (call/initial-state 3
    (call/fresh
      (λ (n)
        (disj
          (peano n)
          (church n)))))
'((((0 . z)) . 1)
  (((1 . z) (0 . (s 1))) . 2)
  (((2 . z) (1 . (s 2)) (0 . (s 1))) . 3))
```

Under the default Racket printing convention, "`.`" is suppressed when it precedes a "(". We retain the "`.`" for legibility—the Racket parameter `current-print` controls this behavior.

Our implementation of `$append` in Section 3.4 induces a depth-first search. Depth-first search is the traditional search strategy of Prolog and can be implemented quite efficiently. As we've seen though, depth-first search is an *incomplete* search strategy: answers can be buried infinitely deep in a stream. The stream that results from a `disj` goal produces elements of the stream from the second goal only after exhausting the elements of the stream from the first.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append (force $1) $2)))
    ...))
```

As a result, even if answers exist microKanren may fail to produce them. We will remedy this weakness in `$append`, and provide microKanren with a simple complete search. We want microKanren to guarantee each and every answer should occur at a finite position in the stream. Fortunately, this doesn't require a significant change.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append $2 (force $1))))
    ...))
```

That's it. This one change to the `promise?` line of `$append` is sufficient to make `disj` *fair* and to transform our search from an incomplete, depth-first search to a complete one.

When the recursive argument to `$append` is an immature stream, we return an immature stream which, when forced, continues with the *other* stream first. The stream `$2` may also be partially computed. If so, then `$append` will process `$2` until it reaches the immature stream at `$2`'s tail. The function `$append` will process this immature stream in the same way.

Our streams are either (potentially empty) lists of states in the case of a fully computed stream, or (potentially empty) improper lists of states with a promise in the final `cdr`, in the case of partially computed streams.

In the case that $1 is fully computed, $append appends $2 to $1. Fully computed streams are finite, so after producing the finite quantity of elements from $1, we can then produce elements from $2, if they exist.

In the second case, if $1 is only partially computed, then it has some potentially-empty finite prefix. We append those elements to a promise that, when forced, will continue by $appending $2 to the result of forcing the promise that was previously the last cdr of $1. The result of forcing this newly created promise, if $2 is immature, will be another promise, this time with a waiting call to $append on the stream that results from forcing the original last cdr of $1 and the stream that results from forcing $2. If $2 is productive, it will mature in a finite number of invocations (possibly 0, if it was mature to begin with). So if $2 is productive, there can be only a finite number of finite prefixes of $1 produced before $2 matures.

Of course, the stream that results from $appending $2 to $1 may *itself* be an argument to a call to $append. The stream that results from the execution of a program is created by successively $appending smaller streams, either in evaluating a disj, or as used in the implementation of conj. The reasoning we use above holds for arbitrary streams, so taking answers from the returned stream amounts to a complete search for the program.

```
> (call/initial-state 3
    (call/fresh
      (λ (n)
       (disj
         (peano n)
         (church n)))))
'((((0 . z)) . 1)
  (((1 . z) (0 . (s 1))) . 2)
  (((1 . z) (0 . (λ (s) (λ (z) 1)))) . 2))
```

Interestingly, we haven't reconstructed some particular, fixed, complete search strategy. Instead, the search strategy of microKanren programs is program- and query-specific. The particular definitions of a program's relations, together with the goal from which it's executed, dictates the order we explore the search tree. By contrast, Spivey and Seres implement breadth-first search, also a complete search, in a language similar to microKanren [29].

Relying on non-strict evaluation simplifies their implementation; manually managing delays would make the call-by-value version less elegant than their implementation. Even excepting that, their implementation requires a somewhat more sophisticated transformation than does ours. Kiselyov et al. describe a different mechanism to achieve a complete search, but they too rely on non-strict evaluation [19]. We achieve a simpler implementation of a complete search by using the delays as markers for interleaving our streams.

With this last change, we complete the definition of microKanren. It clocks in at precisely 54 lines of code—28 of which we need just to implement unification. As advertised, we can use microKanren to write real programs. Below is an expansion of the append relation from Section 1 into microKanren.

```
(define (append l s o)
  (λ (s/c)
    (delay/name
      ((disj
         (conj
           (== l '())
           (== s o))
         (call/fresh
           (λ (a)
             (call/fresh
               (λ (d)
                 (conj
                   (== l `(,a . ,d))
                   (call/fresh
                     (λ (r)
                       (conj
                         (== o `(,a . ,r))
                         (append d s r))))))))))
       s/c))))
```

We also translate here the first query from Section 1; for space we omit the translation of the others.

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (append '(t u v) '(w x) q))))
```

It appears that by simplifying the language in this way, we may have placed some additional burden on the user when writing programs and interpreting the results. miniKanren programs are often much larger than append and composed of multiple and more complicated relations. microKanren may not be especially convenient or friendly for the working logic programmer, but it is a serviceable logic programming language implementable in a call-by-value language and requiring only a minimal group of features from its host.

4. Impure Extensions

The microKanren presented in Section 3 is a complete purely declarative logic programming language. In this section we augment it with additional operators reminiscent of some of those found in Prologs. These operators provide additional control mechanisms.

Naish shows that Prolog's cut (!) is a combination of a deterministic if-then-else and don't-care nondeterminism [23]. We implement these as separate operators, ifte and once, inspired by Kiselyov et al. [19]; ifte is also similar to the cond/3 found in several Prologs [3].

The operator ifte takes three goals as arguments: if the first succeeds, then we execute the second against the result of the first and discard the third. If the first fails, then we execute the third and discard the second. Providing the identifier loop makes the body of the let recursively scoped. let scopes this name over the let's body. If (g0 s/c) returns a promise, we don't want to immediately continue forcing it. That might make our search incomplete again—$ might not be productive. So instead, we return a promise, which, when forced, itself forces $ and then tests the value against our three cases.

```
(define ((ifte g0 g1 g2) s/c)
  (let loop (($ (g0 s/c)))
    (cond
      ((null? $) (g2 s/c))
      ((promise? $) (delay/name (loop (force $))))
      (else ($append-map $ g1)))))
```

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (ifte (== 'a 'b) (== q 'a) (== q 'b)))))
'((((0 . b)) . 1))
```

`once` takes a goal `g` as an argument and returns a new goal as its result. This resulting goal behaves like `g` except that, where `g` would succeed with a stream of more than one element, this new goal returns a stream of only the first.

```
(define ((once g) s/c)
  (let loop (($ (g s/c)))
    (cond
      ((null? $) '())
      ((promise? $) (delay/name (loop (force $))))
      (else (list (car $))))))
```

For the same reasons as `ifte`'s definition, `once`'s definition creates a function named `loop` and uses it in the second clause of the `cond`.

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (once (peano q)))))
'((((0 . z)) . 1))
```

Together, these two operators provide the power of Prolog's cut. Use of these operators can increase the efficiency of our programs. These operators, however, can mangle the connection between logic programming and logic, ultimately costing us some of the flexibility of logic programs that `append` demonstrates.

5. Recovering miniKanren

The microKanren implementation of `append` exemplifies why users might want a more sophisticated set of operators with which to write programs and view the results. We layer the higher-level syntax of miniKanren (`fresh`, `conde`, `conda`, `condu`, and `run`) over microKanren in terms of macros and a couple of helper functions; the `==` function and the `define-relation` macro will transfer directly.

5.1 `conde` and `fresh`

As a first step to reconstructing miniKanren, we create operators `disj+` and `conj+` that allow us to write the disjunction and conjunction of more than two goals at a time. The `disj+` (`conj+`) of a single goal is just the goal itself. For more than one goal, we recursively `disj` (`conj`) the first goal onto the result of the recursion. We use `define-syntax` and `syntax-rules` to implement recursive macros.

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

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

With `disj+` and `conj+`, we are able to construct miniKanren's `conde` as a macro that merely rearranges its arguments. miniKanren's `conde` is the `disj+` of a sequence of `conj+`s:

```
(define-syntax-rule (conde (g0 g ...) (g0* g* ...) ...)
  (disj+ (conj+ g0 g ...) (conj+ g0* g* ...) ...))
```

We build the `fresh` of miniKanren, which introduces zero or more fresh variables, as a recursive macro using `call/fresh` and `conj+`:

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

5.2 `conda` and `condu`

We can also recover the impure miniKanren operators `conda` and `condu`, which provide committed choice and committed choice with a "don't-care" nondeterminism, respectively.

As a first step we implement `ifte*`, which nests `ifte` expressions. It takes a sequence of lists containing two goal expressions each, followed by a single goal expression at the end and transforms these into a sequence of nested `ifte` expressions, using the last goal as the final `ifte`'s else clause.

```
(define-syntax ifte*
  (syntax-rules ()
    ((_ g) g)
    ((_ (g0 g1) (g0* g1*) ... g)
     (ifte g0 g1 (ifte* (g0* g1*) ... g)))))
```

With this, we can implement `conda` and `condu` as macros. `conda` takes a sequence of sequences of two or more goal expressions each, except the last which is a sequence of one or more goals. With `conj+`, we transform this syntax into an `ifte*` expression:

```
(define-syntax-rule (conda (g0 g1 g ...) ... (gn0 gn ...))
  (ifte* (g0 (conj+ g1 g ...)) ... (conj+ gn0 gn ...)))
```

We implement `condu` by adding `once` to each first element of each sequence, and building a `conda` from the result:

```
(define-syntax-rule (condu (g0 g1 g ...) ... (gn0 gn ...))
  (conda ((once g0) g ...) ... ((once gn0) gn ...)))
```

5.3 `run`

The last miniKanren form we reconstruct is `run`, the external interface that allows us to execute a miniKanren program. The `run` operator, from Section 1, takes as arguments a positive natural number n or `#f`, indicating the number of answers to return (similar to `call/initial-state`); a query variable q (in parentheses); and a sequence of goal expressions. We format the returned answers in terms of the query variable and return them in a list. miniKanren programs, like microKanren programs, may not terminate.

5.3.1 Formatting and Structuring Answers

microKanren programs can create a large number of variables in the course of their execution. Rather than returning the values of all of these variables, we only return the value of the query variable (and variables associated with it). This process is called *answer projection* [9, 17].

The function `project-var0` takes a state as an argument and formats and returns the first variable with respect to the substitution of that state. We invoke `apply-subst` with a copy of the first variable that we recreate by calling `var`.

```
(define (project-var0 s/c)
  (let ((v (apply-subst (var 0) (car s/c))))
    ...))
```

The function `apply-subst` calls `find` recursively over a term. It fully dereferences the 0th variable, meaning `project-var0` dereferences that variable, any variables in the terms to which it's associated, and so on. This operation *grounds* the first variable with respect to the substitution. The resulting value is a tree whose remaining variables are free in the substitution.

```
(define (apply-subst v s)
  (let ((v (find v s)))
    (cond
      ((var? v) v)
      ((pair? v) (cons (apply-subst (car v) s)
                       (apply-subst (cdr v) s)))
      (else v))))
```

Given such an expression, as well as an empty substitution `s` and some starting variable index `c`, `build-r` returns a *rename substitution*. This is a substitution that, when applied to a term, will faithfully and consistently replace all variable names with elements from a different set. We build our rename substitution via a preorder walk and use the length of the growing substitution together with `c` to create unique variable names.

```
(define (build-r v s c)
  (cond
    ((var? v) `((,v . ,(+ (length s) c)) . ,s))
    ((pair? v) (build-r (cdr v) (build-r (car v) s c) c))
    (else s)))
```

From there, we can complete `project-var0`'s definition. We fully ground the earliest variable and build a rename substitution using the variable counter `c` as the starting point. This ensures we create a consistent renaming. We then pass over the answer term, dereferencing variables with respect to the rename substitution. We then repeat the process one last time, with our initial counter at 0. In this way we canonicalize all of our variable names.

```
(define (project-var0 s/c)
  (let ((v (apply-subst (var 0) (car s/c))))
    (let ((v (apply-subst v (build-r v '() (cdr s/c)))))
      (apply-subst v (build-r v '() 0)))))
```

5.3.2 Off and `run`ning

With these functions defined, `run` is easily implemented:

```
(define-syntax-rule (run n (q) g0 g ...)
  (map project-var0
    (call/initial-state n (fresh (q) g0 g ...))))
```

We use the variable `q` and the goals `g0 g ...` to build a `fresh` goal. We provide this goal to `call/initial-state`, and we map `project-var0` over the resulting list. Our miniKanren implementation relies directly on the microKanren `==` and `define-relation`, and so `run` completes our miniKanren reconstruction.

In fact, we can expand a miniKanren `run` expression into calls to the microKanren primitives and helper functions in terms of which they are defined. The expansion of `(3)` in Section 1 is below.

We print our answers using n as our canonical variable elements, rather than "`_.`n" like other miniKanrens. Numbers are already excluded from our term language; it's incidental that our external representation of variables matches the internal one. It only matters that they print differently than the non-variable part of the term language.

```
> (map project-var0
    (call/initial-state #f
      (call/fresh
        (λ (q)
          (call/fresh
            (λ (l)
              (call/fresh
                (λ (s)
                  (conj
                    (== `(,l ,s) q)
                    (append l s '(t u v w x))))))))))
```

6. Related Works

There has been an extensive research on logic programming implementation [2]. Carlsson limits his survey to Prolog implementation issues in a functional setting [7]. There have been several interpreter-based implementations in Lisp-like languages, Komorowski's being the first [20]. miniKanren certainly has a close connection with these functional Prolog implementations. As we have shown in Section 1, we can transliterate miniKanren definitions to pure Prolog.

Spivey and Seres's [29] present a Haskell embedding of a language quite similar to microKanren. They begin with depth-first search language, and through transformations derive an implementation of breadth-first search. Kiselyov's "A Taste of Logic Programming" [18] is another approach to implementing a functionally-embedded logic programming language, and it sidesteps managing variable creation, infinite streams, and interleaving. We explicitly aim to provide a practical complete search in an embedding easily portable to other host languages, and our choice of a call-by-value host and the emphasis on managing interleaving yield different trade-offs.

Hinze [15, 16] and Kiselyov et al. [19] implement backtracking with asymptotic performance improvements over stream-based approaches like that used in microKanren and the works cited above. These context-passing implementations are also more complicated to understand and to implement. We chose to use streams in part to more easily communicate ideas.

The fair search operators in Kiselyov et al.'s LogicT monad provide the basis of the interleaving search in earlier miniKanren implementations. The LogicT transformer augments an arbitrary monad with backtracking and control operators similar to those we use. Because we have access to the whole logic program in our embedding and take special care to control interleaving in recursions, we can use less frequent interleaving and maintain a complete search.

Both the original Kanren [10] and miniKanren [11] enmesh macros that provide the syntax with logic critical to control. microKanren by contrast relies on a small and limited feature set, and is by design relatively straightforward to port to other host languages. We suspect that part of the reason for the uptake of this work's earlier draft is the decoupling of macros that provide user-friendly syntax from the core logic components. This separation of concerns also aids developers in porting to other host languages.

7. Conclusion

Our development led us to a number of interesting, still-open problems. Hinze [15] shows our list-based implementation of nondeterminism is asymptotically slower than a continuation-based "context-passing" implementation. We would like to combine our manual control of delays with a context-passing implementation á la Hinze and Kiselyov et al. [19].

While `define-relation` is sufficient to ensure our search is complete, it in general causes more interleaving than necessary. For instance, mutually-recursive relations only need one interleaving point between them, and we don't need to interleave at all deterministic relations. We could statically "push down" the delays into the body of a relation, reducing the amount of interleaving we perform while retaining a complete search.

We would like to mechanically prove the correctness of microKanren's search with a dependently-typed implementation whose types encode correctness of substitutions and unification. Earlier work by Kumar [22] in mechanizing facets of a miniKanren implementation might provide a starting point.

While the only constraint in the micro- and miniKanren implementations presented here is ==, many of miniKanren's more interesting research applications require additional constraints. Hemann and Friedman [13] recently developed a microKanren framework for constraints [14]; we would like to extend this framework to miniKanren with a generic mechanism for simplifying and formatting constraints.

Seres [26] suggests functional-logic programming as a future direction for her embedding; this still remains an open avenue of research, and given the similarity of our approaches, microKanren might be an appropriate starting place for this as well.

Acknowledgments

We thank our reviewers, both known and anonymous, for their comments and suggestions. We thank also the many microKanren implementers and expositors for extending our work to new settings. This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[2] Isaac Balbin and Koenraad Lecot. *Logic Programming: A Classified Bibliography*. Springer Science & Business Media, 2012.

[3] Michel Billaud. Prolog Control Structures: a Formalization and its Applications. In K. Fuhi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 57–73. Elsevier Science Publishers, 1988.

[4] William E. Byrd and Daniel P. Friedman. αKanren: A Fresh Name in Nominal Logic Programming. In *Scheme 8*, pages 79–90 (*see also* `http://webyrd.net/alphamk/alphamk.pdf` *for improvements*), 2007.

[5] William E. Byrd, Eric Holk, and Daniel P. Friedman. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Scheme 12*, September 2012. URL `http://www.schemeworkshop.org/2012/papers/byrd-holk-friedman-paper-sfp12.pdf`.

[6] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[7] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[8] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[9] Andreas Fordan. *Projection in Constraint Logic Programming*. Ios Press, 1999.

[10] Daniel P. Friedman and Oleg Kiselyov. A declarative applicative logic programming system, 2005. URL `http://kanren.sourceforge.net/`.

[11] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, Cambridge, MA, 2005.

[12] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined Style. In *Proc. 4th ICFP*, pages 18–27. ACM, 1999.

[13] 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`.

[14] Jason Hemann and Daniel P. Friedman. A framework for extending microkanren with constraints. In *Scheme 15*, 2015. Forthcoming.

[15] Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.

[16] Ralf Hinze. Prolog's control constructs in a functional setting: Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(02):125–170, 2001.

[17] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland HC Yap. Output in CLP($\mathcal{R}$) . In *Fifth Generation Computing Systems*, pages 987–995, 1992.

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

[19] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN ICFP*, pages 192–203. ACM, September 2005.

[20] H. J. Komorowski. QLOG: The programming environment for PROLOG in LISP. In K. L. Clark et al., editors, *Logic Programming*, pages 315–324. Academic Press, 1982.

[21] Robert A. Kowalski. *Logic for Problem Solving*. North-Holland/Elsevier, 1979.

[22] Ramana Kumar. Mechanising Aspects of miniKanren in HOL, 2010. Australian National University. Bachelors thesis.

[23] Lee Naish. Pruning in logic programming. Technical report, Technical Report 95/16, Department of Computer Science, University of Melbourne, June 1995.

[24] Joseph P. Near, William E. Byrd, and Daniel P. Friedman. αlean*TAP*: A declarative theorem prover for first-order classical logic.

[25] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[26] Silvija Seres. *The Algebra of Logic Programming*. PhD thesis, University of Oxford, 2001.

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

[28] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, October 1989. doi: 10.1145/69558.69563. URL `http://doi.acm.org/10.1145/69558.69563`.

[29] JM Spivey and Silvija Seres. Embedding Prolog in Haskell. In E. Meier, editor, *Haskell 99*, 1999.