First-order miniKanren representation: Great for tooling and search

GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada
WILLIAM E. BYRD, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA

We present a first-order implementation of miniKanren that makes it easy to build a miniKanren debugger, and allows any other processes (including a human or a neural network) to guide the search. Typical miniKanren implementations use procedures to represent data structures like goals and streams. Instead, our implementation uses Racket structs, which are transparent, decomposible, manipulable, and not coupled with a particular search strategy. We obtain this first-order implementation by carefully applying defunctionalization rules to a higher-order implementation, deriving two compatible versions with the same search behavior and comparable performance. Decoupling the search in the first-order implementation makes it possible to analyze, transform, and optimize miniKanren programs, even while that program is running. We use a "human guided" search as a miniKanren debugger, and to demonstrate the breath of supported search strategies. The flexibility in how we interpret goals and streams opens up possibilities for new tools, and we hope to inspire the community to build better miniKanren tooling.

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

Additional Key Words and Phrases: miniKanren, microKanren, logic programming, relational programming, Scheme, Racket, program synthesis

1 INTRODUCTION

While miniKanren has been applied successfully to at least seven different programming problems [1], understanding how miniKanren arrives at answers is challenging for all but the simplest examples. Well-hidden typos and inefficient ordering of conjuncts can be difficult to detect. Why is it that after more than a decade of work, miniKanren still lacks tools such as support for basic debugging?

A seemingly different challenge is changing the way the search strategy is implemented in miniKanren. Different strategies may be better-suited to particular miniKanren programs, so flexibility in choosing search strategies can make programs run faster. In Byrd et al. [1], changing the search strategy required a complete and hacky reimplementation of miniKanren.

These two problems—building a debugger, and implementing various search strategies—can seem very different on the surface. However, they are actually symptoms of one design decision common to most miniKanren implementations: *a higher-order representation of programs coupled to a particular search strategy.*

Most miniKanren implementations represent an executing relational program using *goals* and *streams*. In a *higher-order* implementation, goals and streams are represented using procedures. Procedures encapsulate

Authors' addresses: Gregory Rosenblatt, University of Alabama at Birmingham, USA, gregr@uab.edu; Lisa Zhang, University of Toronto Mississauga, Canada, lczhang@cs.toronto.edu; William E. Byrd, University of Alabama at Birmingham, USA, webyrd@uab.edu; Matthew Might, University of Alabama at Birmingham, USA, might@uab.edu.

miniKanren.org/workshop/2019/8-ART2

computational behavior, defining the search strategy for satisfying constraints. Procedures can be applied but can't be modified or decomposed, so printing one will yield:

```
> (fresh (x y)
    (conde ((== 10 x) (== 20 y))
           ((== 30 x) (== 40 y))))
#<procedure:...>
```

Our proposal is to use a *first-order* program representation, where goals and streams are data structures not coupled to any search strategy. These data structures are decomposible, inspectable and manipulatable:

```
> (fresh (x y)
    (conde ((== 10 x) (== 20 y))
           ((== 30 x) (== 40 y))))
#s(disj
   #s(conj #s(== 10 #s(var x 1)) #s(== 20 #s(var y 2)))
   #s(conj #s(== 30 #s(var x 1)) #s(== 40 #s(var y 2))))
```

The rest of the paper is organized as follows: Section 2 shows a motivating example of our miniKanren stepper in action. Section 3 describes what is required of a miniKanren implementation so that we can write such a stepper. In Section 4, we implement a first-order version of miniKanren alongside a higher-order version with the same search behavior. In Section 5, we explain the program transformations that makes the stepper human-readable. Section 6 describes a neural guided search strategy, and provides more ideas for better miniKanren tools that leverage the first-order program representation. We hope to inspire you to build some of these tools.

2 APPENDOH THE DEFICIENT: A MINIKANREN EMERGENCY

We begin this paper with a *miniKanren emergency* that we want you—dear readers—to help us resolve. Nub Let, a hapless but ernest beginner miniKanren programmer, has accidentally replaced our beloved "appendo the Magnificent" relation with **appendoh the deficient**.

As might be expected of such a dubious relation, appendoh has been failing left and right. Even worse, Nub Let forgot to email the code for appendoh as part of the bug report. Fortunately, Nub Let included a transcript of an interactive stepper session, enabled by using a first-order representation of miniKanren code.

Will you help us save this paper from the embarrassment of including a buggy five-line program?

Recall that a correct implementation of appendo looks like this:

```
(define-relation (appendo l s ls)
  (conde
   ((== '() l) (== s ls))
   ((fresh (a d res)
      (== `(,a . ,d) l)
      (== `(,a . ,res) ls)
      (appendo d s res)))))
```

The output of interactively stepping through an appendoh query is shown in Figure 1. The execution of the query is broken down into "steps". Each step of evaluation corresponds to the expansion of either a disjunct or a relation definition. In each step, we can choose a "choice" or a disjunct to expand. In the appendoh case, the disjuncts correspond to conde branches, each of which constrains the query variable in a different way. The stepper shows the values the query variables will have if we take that choice. The stepper also shows zero or more constraints associated with the choice. These constraints correspond to goals that must be satisfied if we take the choice.

To prevent exponential explosion of choices, the stepper only displays the *new choices*: the descendants of the choice we just made in the previous step. We retain a history of user choices, so that users can revisit previous choices by backtracking. Not all choices will have descendants: some may fail entirely, and some successful choices may not have descendants.

Based on the stepper output in Figure 1 alone, can you deduce what the error is in the implementation of appendoh? The stepper output of the correctly specified appendo is in Appendix A.

It turns out that Nub Let hid the appendoh implementation in Appendix B. Even without the code, a keen reader may have noticed that in all the recursive calls to appendoh shown in the interactive stepper (look at the constraints), the value of the *last* argument never changes from `(1 2 3)`. This suggests that the bug is somewhere in the recursive call portion of the appendoh code, and that perhaps the last argument is incorrect.

3 DESIDERATA

The appendoh example is simple and has a linear search space. Even so, we hope that this example convinces you that a miniKanren stepper can be a useful debugging tool. In particular, being able to choose which disjuncts to explore is even more helpful for complex miniKanren programs with many disjuncts. Only some of the disjuncts might be buggy, so bugs would only manifest in queries that explore that disjunct. Being able to choose the order of execution allows the programer to determine which portion of the search space is of interest.

To build a stepper like the one in Section 2, our miniKanren implementation must satisfy two requirements:

(1) **Streams should be decomposable, and thus inspectable.** We need some other data structure that, unlike procedures, can be programmatically transformed into a human-readable, textual representation.
(2) **Small-step executions should be possible.** We need to be able to take one small step to expand the chosen disjunct. Fine-grained control is important: the idea is not to search this disjunct exhaustively, but to provide more information to make another choice in the next step.

In short, we need the search state—the data representation of the stream—to be decoupled from the search behavior—the search strategy that miniKanren uses. Unfortunately, typical miniKanren implementations use a higher-order representation where these are hard to tease apart. Streams are represented as procedures that bundle the search state (the data) with the search behavior (a single interpretation of that data). To satisfy our requirements, we need a representation where the search state is accessible, so we can use it in at least two different ways: as information to be examined by the user, and as a stream to be stepped. If we instead use a first-order representation of the search state, we can view the stream as syntax, writing a different interpreter to satisfy each requirement.

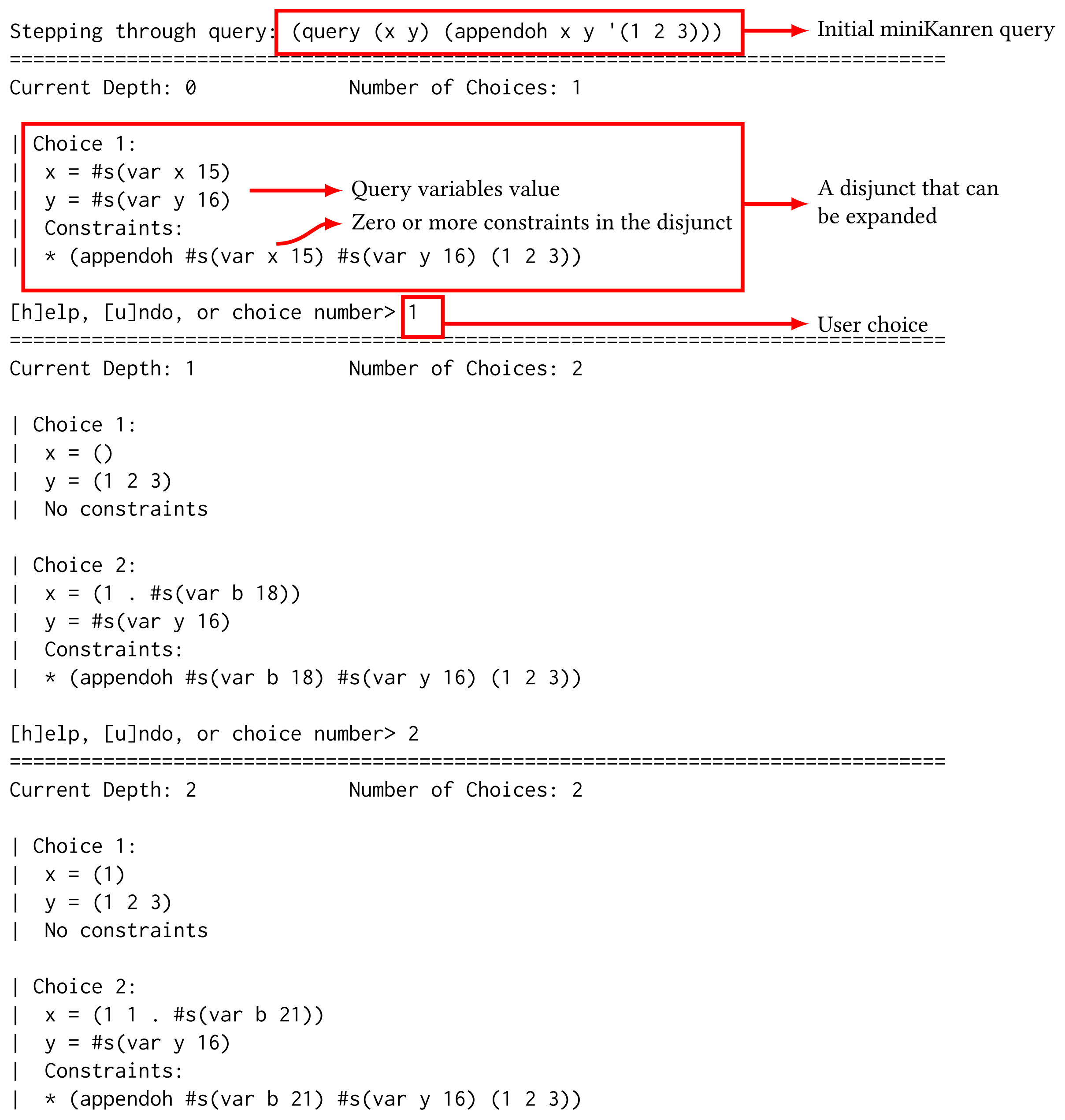
If both functionalities are present, then the interactive stepper is actually very simple. We need only to write code to extract the list of disjuncts, present these disjuncts as choices to the user, display the query variable values and constraints associated with each choice, read user input, take a step to expand the chosen disjuct, and repeat.

4 IMPLEMENTATIONS OF MINIKANREN

In this section, we build two miniKanren implementations, one of which decouples the search state from the search behavior. The miniKanren implementation satisfies the two requirements from Section 3: that streams should be decomposable, and small-step executions should be possible.

Our plan is to build miniKanren on top of a version of μKanren. We start by implementing a higher-order μKanren to use as a familiar reference point. Our implementation is inspired by that of Hemann and Friedman [2], but modified for improved performance. We encourage readers interested in implementation to review Hemann and Friedman [2] for a detailed presentation of the μKanren language.

To obtain first-order μKanren, we will defunctionalize the higher-order version. In particular, the first-order μKanren uses Racket `struct`, which can be decomposed using `match` expressions (Section 4.1.1). This version of



```
Stepping through query: (query (x y) (appendoh x y '(1 2 3)))
================================================================================
Current Depth: 0             Number of Choices: 1

| Choice 1:
|  x = #s(var x 15)
|  y = #s(var y 16)
|  Constraints:
|  * (appendoh #s(var x 15) #s(var y 16) (1 2 3))

[h]elp, [u]ndo, or choice number> 1
================================================================================
Current Depth: 1             Number of Choices: 2

| Choice 1:
|  x = ()
|  y = (1 2 3)
|  No constraints

| Choice 2:
|  x = (1 . #s(var b 18))
|  y = #s(var y 16)
|  Constraints:
|  * (appendoh #s(var b 18) #s(var y 16) (1 2 3))

[h]elp, [u]ndo, or choice number> 2
================================================================================
Current Depth: 2             Number of Choices: 2

| Choice 1:
|  x = (1)
|  y = (1 2 3)
|  No constraints

| Choice 2:
|  x = (1 1 . #s(var b 21))
|  y = #s(var y 16)
|  Constraints:
|  * (appendoh #s(var b 21) #s(var y 16) (1 2 3))
```

Initial miniKanren query
Query variables value
Zero or more constraints in the disjunct
A disjunct that can be expanded
User choice

Fig. 1. Interactive stepper output for the buggy appendoh

μKanren also has a small step interpreter step that performs a small, finite expansion of a stream (Section 4.3.1). For comparability, both μKanren versions implement the same search behavior.

The first-order and higher-order implementations share a portion of code, presented in Section 4.1. The remaining portions of both implementations are shown side by side in Figure 2 on page 8. We discuss the higher-order portion in Section 4.2, and compare with the first-order portion in Section 4.3. We recover a version of miniKanren in Section , compatible to both first-order and higher-oder μKanren.

4.1 Common Implementation

The common portion defines *logic variables*, constraint *states*, *unification*, and *reification*:

4.1.1 Logic Variables. Logic variables[1] are defined using `struct`, which has the benefit of avoiding type collision with vectors. This allows support for vectors as terms, though for simplicity we do not include this support.

The declaration `(struct NAME (FIELD...) #:prefab)` defines a new record type, automatically generating a constructor named `NAME` and field accessors named `NAME-FIELD` for each `FIELD`. The `match` form can pattern match against values of this new type. Additionally, using `#:prefab` gives values of this type the printable notation `#s(NAME FIELD ...)`.

A logic variable contains the name it was given in the program, to help with debugging, and a non-negative integer index used to identify it. Unlike the implementation of Hemann and Friedman [2], a fresh logic variable is constructed with an index allocated from a shared, mutable counter, using `var/fresh`. This simplifies the role of states, which would otherwise carry a functional counter [2]. Additionally, without this simplification, a first-order implementation would be burdened with managing the functional counter to resolve unbound logic variables[2].

```
;; Logic variables
(struct var (name index) #:prefab)
(define (var=? x1 x2)
  (= (var-index x1) (var-index x2)))
(define initial-var (var #f 0))
(define var/fresh
  (let ((index 0))
    (lambda (name) (set! index (+ 1 index))
      (var name index))))
```

4.1.2 States. States containing constraints are defined using `struct` and track equality information using a substitution. Other kinds of constraints could be supported, but for simplicity we will only consider equality constraints in this implementation.

Substitutions are implemented as in Hemann and Friedman [2], though we choose to enforce acyclic term structure using the `occurs?` check when extending a substitution via `extend-sub`. Our choice is consistent with typical miniKanren implementations.

```
;; States
(define empty-sub '())
(define (walk t sub)
  (let ((xt (and (var? t) (assf (lambda (x) (var=? t x)) sub))))
    (if xt (walk (cdr xt) sub) t)))
(define (occurs? x t sub)
  (cond ((pair? t) (or (occurs? x (walk (car t) sub) sub)
```

[1]Recall that a logic variable is a term that acts as a placeholder for an unknown value.

[2]Because we would like to implement many interpreters, this burden would multiply, because each interpreter would have to include support for resolving unbound logic variables.

```
                      (occurs? x (walk (cdr t) sub) sub)))
        ((var? t)  (var=? x t))
        (else      #f)))
(define (extend-sub x t sub)
  (and (not (occurs? x t sub)) `((,x . ,t) . ,sub)))

(struct state (sub) #:prefab)
(define empty-state (state empty-sub))
```

4.1.3 Streams and Unification. Unification enforces equality between two terms and returns a substitution. It is defined as `unify`, and is implemented as in Hemann and Friedman [2].

```
;; Streams
(define mzero     #f)
(define (unit st) (cons st mzero))

;; Unification
(define (unify/sub u v sub)
  (let ((u (walk u sub)) (v (walk v sub)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) sub)
      ((var? u)                            (extend-sub u v sub))
      ((var? v)                            (extend-sub v u sub))
      ((and (pair? u) (pair? v))           (let ((sub (unify/sub (car u) (car v) sub)))
                                             (and sub (unify/sub (cdr u) (cdr v) sub))))
      (else                                (and (eqv? u v) sub)))))
(define (unify u v st)
  (let ((sub (unify/sub u v (state-sub st))))
    (if sub (unit (state sub)) mzero)))
```

4.1.4 Reification. Reification normalizes the presentation of terms containing logic variables for easier reading and comparison. We implement `reify` as in Hemann and Friedman [2], except using a local, mutable counter to simplify allocation of normalized variable identifiers.

```
;; Reification
(define (walk* tm sub)
  (let ((tm (walk tm sub)))
    (if (pair? tm)
        `(,(walk* (car tm) sub) .  ,(walk* (cdr tm) sub))
        tm)))
(define (reified-index index)
  (string->symbol
    (string-append "_." (number->string index))))
(define (reify tm st)
  (define index -1)
  (walk* tm (let loop ((tm tm) (sub (state-sub st)))
              (define t (walk tm sub))
```

```
             (cond ((pair? t) (loop (cdr t) (loop (car t) sub)))
                   ((var? t)  (set! index (+ 1 index))
                              (extend-sub t (reified-index index) sub))
                   (else      sub)))))
(define (reify/initial-var st)
  (reify initial-var st))
```

4.2 Higher-order μKanren

Goals, streams, and stream maturation are implemented differently in higher- vs first-order μKanren. The remaining implementation of higher-order μKanren is shown on the left-hand-side of Figure 2 on page 8.

4.2.1 Goals. Goals include binary *disjunctions* and *conjunctions*, equality constraints, and calls to user-defined relations.

Binary disjunctions and conjunctions are constructed by `disj` and `conj`, which are defined as in Hemann and Friedman [2], except that our implementation introduces explicit pauses.

Equality constraints are constructed with `==`, which uses `unify`.

A call to a user-defined relation applies the relation to argument terms. Each call is represented as a *thunk*, wrapped with the constructor `relate`. A user-defined relation is expressed as a Racket procedure, such that forcing a call's thunk corresponds to applying the procedure to the call's arguments. (Higher-order μKanren places no restrictions on what a Racket procedure implementing a relation may do, but Section 4.3.2 will describe a few restrictions when defunctionalizing relation definitions in first-order μKanren.) A use of `relate` is also passed first-order metadata describing the call, but this information is discarded by our higher-order implementation.

4.2.2 Streams. Streams define a search strategy for finding answers, which are states that satisfy all of a program's constraints. For improved performance[3], we approximate the behavior of the interleaving search used in Byrd et al. [1].

A mature stream is either the empty stream `#f`, or a pair of an answer and a remaining stream. An immature stream is a thunk that has suspended the search before its completion. Forcing this thunk causes the search to continue. Streams will typically be constructed with the help of `mplus`, `bind`, and `pause`.

In our first-order implementation, we represent suspension more explicitly than in Hemann and Friedman [2]. A `pause` produces an immature stream that delays transition of a goal to a stream by introducing a thunk. When forced, the thunk applies the goal to a state.

The stream constructor `mplus` combines two streams. To improve answer diversity, our implementation of `mplus` interleaves more often, to not allow a highly-productive branch to monopolize search resources. We interleave each time an answer is popped off of a mature stream, rather than popping all available answers before interleaving. To make this possible, `bind` needs to introduce a `pause` every time an answer is discovered.

Our implementation of `mplus` also delays forcing a child stream until it is ready to be examined. This small detail can significantly reduce wasted work and memory usage.

4.2.3 Stream maturation. Stream maturation is implemented by `mature`. In order to extract an answer from a stream, `mature` will continue forcing execution of an immature stream until it is mature.

[3]Compared with the search described by Hemann and Friedman [2], this implementation can synthesize quines about twice as fast, and twines and thrines nearly three times as fast, using a simple relational interpreter.

```
;; higher-order microkanren









(define (mature? s) (or (not s) (pair? s)))
(define (mature s)
  (if (mature? s) s (mature (s))))




(define (disj g1 g2)
  (lambda (st) (mplus (pause st g1)
                      (pause st g2))))
(define (conj g1 g2)
  (lambda (st) (bind (pause st g1) g2)))
(define (relate thunk _)
  (lambda (st) (pause st (thunk))))
(define (== t1 t2) (lambda (st) (unify t1 t2 st)))



(define (mplus s1 s2)
  (let ((s1 (if (mature? s1) s1 (s1))))
    (cond ((not s1) s2)
          ((pair? s1)
           (cons (car s1)
                 (lambda () (mplus s2 (cdr s1)))))
          (else (lambda () (mplus s2 s1))))))
(define (bind s g)
  (let ((s (if (mature? s) s (s))))
    (cond ((not s) #f)
          ((pair? s)
           (mplus (pause (car s) g)
                  (lambda () (bind (cdr s) g))))
          (else (lambda () (bind s g))))))
(define (pause st g) (lambda () (g st)))
```

```
;; first-order microkanren
(struct disj   (g1 g2)                   #:prefab)
(struct conj   (g1 g2)                   #:prefab)
(struct relate (thunk description)       #:prefab)
(struct ==     (t1 t2)                   #:prefab)
(struct bind   (bind-s bind-g)           #:prefab)
(struct mplus  (mplus-s1 mplus-s2)       #:prefab)
(struct pause  (pause-state pause-goal) #:prefab)

(define (mature? s) (or (not s) (pair? s)))
(define (mature s)
  (if (mature? s) s (mature (step s))))

(define (start st g)
  (match g
    ((disj g1 g2)
     (step (mplus (pause st g1)
                  (pause st g2))))
    ((conj g1 g2)
     (step (bind (pause st g1) g2)))
    ((relate thunk _)
     (pause st (thunk)))
    ((== t1 t2) (unify t1 t2 st))))

(define (step s)
  (match s
    ((mplus s1 s2)
     (let ((s1 (if (mature? s1) s1 (step s1))))
       (cond ((not s1) s2)
             ((pair? s1)
              (cons (car s1)
                    (mplus s2 (cdr s1))))
             (else (mplus s2 s1)))))
    ((bind s g)
     (let ((s (if (mature? s) s (step s))))
       (cond ((not s) #f)
             ((pair? s)
              (step (mplus (pause (car s) g)
                           (bind (cdr s) g))))
             (else (bind s g)))))
    ((pause st g) (start st g))
    (_            s)))
```

Fig. 2. Comparison of higher-order vs. first-order μKanren

4.3 First-order μKanren

We defunctionalize the higher-order representation of *goals* and *streams* to obtain the first-order implementation of μKanren shown in Figure 2. This new version of μKanren represents goal and streams as decomposable data structures, and interprets streams using `step`, with the help of a goal interpreter `start`.

The goal constructors `disj`, `conj`, `relate`, and `==`, along with the stream constructors `bind`, `mplus`, and `pause`, are now defined using `struct` to allow decomposition by pattern matching via the keyword `match`.

We add two new procedures, `step` and `start`, which are small-step interpreters for streams and goals respectively. These interpreters work together to implement the same interleaving search strategy defined by the higher-order representation. More specifically, `step` interprets an immature stream by making a finite amount of progress in searching for an answer. It interprets a mature stream by simply returning the stream. The interpreter `start` interprets a goal in the context of a state, producing a new stream.

4.3.1 Defunctionalizing goals and streams. Each higher-order goal and stream definition maps to a pattern matching clause in the corresponding interpreter. We have aligned the higher-order and first-order implementation listings in Figure 2 to make this correspondence easy to see: for each line in the higher-order definition on the left, scan along the same line to see the first-order version on the right.

We need to be careful when translating from higher-order to first-order. Failing to apply `step` at the right time may impede search progress, and change the order in which answers are found. Applying `step` too soon may lead to an infinite loop when a recursive relation is involved. These issues may not manifest immediately, so when they show up, it is unclear whether they are due to a particular query, the definition of a relation, or a problem with the search implementation. To avoid causing these issues when defunctionalizing the higher-order representation, we must match its order of evaluation. We follow these rewrite rules to defunctionalize:

Rule: Top-level higher-order constructors become `match` clauses

- Every top-level higher-order goal constructor becomes a `struct` pattern clause of the `match` expression in `start`.
- Every top-level higher-order stream constructor becomes a `struct` pattern clause of the `match` expression in `step`.
- In each case, the pattern decomposes the `struct`, naming its components the same way the higher-order constructors name their parameters. The clause bodies will then be defunctionalized.

Rule: Delaying goal/stream interpretation

- Anywhere that `lambda` is used in the higher-order implementation to delay construction of a goal or stream, we simply apply the corresponding `struct` constructor.
- **Justification**: though the higher-order constructors are procedures which immediately execute their interpretation when applied, the enclosing `lambda` leaves them inert until forced. Likewise, newly-constructed `struct` data remains inert until interpreted by `step` or `start`.

Rule: Interpreting a goal/stream

- Anywhere that the higher-order implementation does not delay construction of a goal, we apply `start` to the result of the corresponding `struct` constructor.
- Except in the case of `pause`, anywhere that the higher-order implementation does not delay construction of a stream, we apply `step` to the result of the corresponding `struct` constructor.
- Anywhere the higher-order implementation uses `pause`, we simply apply the `pause` `struct` constructor.
- **Justification**: We apply `start` and `step` in this way because the higher-order constructors also immediately execute their interpretation. But `pause` is an exception because it does not immediately execute its interpretation. Instead, it delays interpretation by producing an immature stream (a thunk).

Rule: Forcing immature streams

- Anywhere that the higher-order implementation forces progress of an immature stream, which corresponds to applying a thunk, we apply `step` to that stream to also force progress. Look at the definition of `mature` in Figure 2 for a simple example of this.

4.3.2 Defunctionalizing user-defined relations. As in the higher-order implementation, we represent calls to user-defined relations using thunks wrapped with the constructor `relate`. Forcing such a thunk applies a Racket procedure to the call's arguments. Unlike the higher-order implementation, uses of first-order `relate` will retain the call description metadata passed alongside the thunk. While an entirely first-order representation of user-defined relations is possible, we prefer retaining this use of thunks because it allows us to leverage Racket's implementation[4] of lexical scope and procedure abstraction, rather than reimplement this ourselves.

Despite continuing to use thunks to represent calls to relations, it is still possible to perform arbitrary manipulation of relation definitions if we make some assumptions about what such definitions may do. Specifically, we will assume the use of Racket is limited to constructing μKanren goals. When these goals include terms, the terms may be expressed using constants, references to lexical variables, pair construction, or fresh logic variable construction.

Given our assumptions, we can obtain a first-order representation of a user-defined relation by applying it to fresh logic variables to represent the parameters. Any goals in the result are those specified in the body of the relation. Any terms in those goals are also either those specified, or they are logic variables. If a logic variable is one of those used to represent a parameter of the relation, then the relation definition must reference that parameter at that location. All other logic variables were created fresh. Additionally, since each use of `relate` wraps metadata describing the call, we can always reconstruct a complete first-order description of a relation, even if it is recursive.

4.4 Recovering miniKanren

In Appendix C we implement miniKanren on top of μKanren, using `syntax-rules` definitions similar to those in Hemann and Friedman [2], with a notable addition: we define `query`, which describes the initial stream built by a corresponding `run`. Since we are interested in manipulating streams in more ways than just extracting answers, we need to separate stream construction from answer extraction. To extract answers from a stream, we provide `stream-take` such that:

```
(run N body ...) ==> (stream-take N (query body ...))
```

We will make use of the metadata wrapped by `relate` when visualizing program representation in Section 5. The metadata is a list containing the relation procedure (to allow unambiguous reference), the name of that procedure, and all of the argument terms.

4.5 Performance

When comparing the higher-order and first-order implementations, we notice a small performance difference when generating twines and thrines using a simple relational interpreter, and when running relational arithmetic benchmarks. On these tests, the first-order implementation is roughly 5 percent slower and spends about 10 percent more time in the garbage collector. Chez Scheme runs a Scheme version of our benchmarks about 25 percent faster than Racket, but the first-order vs. higher-order performance ratio remains about the same.

[4]Hemann and Friedman [2] leverages Scheme in the same way.

5 PROGRAM TRANSFORMATIONS

With a first-order miniKanren implementation in place from Section 4, streams are decomposable (and therefore printable), and small-step executions are possible.

You might wonder whether we could stop here. After all, we can apply `step` to progress through a query, and print the structure of the stream at each step. Appendix D shows the output of stepping through the query `(query (xs) (appendo xs xs '(a a)))` using interleaving search.

Unfortunately, this presentation of execution is intolerably verbose, even for such a simple query. It is difficult to extract useful information about actual available choices. In addition, many useless steps are taken to expand goals that are obviously failing. In order to obtain a simpler program representation that is actually human-readable, we need to **extract the useful information about each disjunct**, and **remove obviously failing disjuncts**.

This section recovers the useful trace information that we saw in Figure 1, by introducing stream transformations. Section 5.1 describes how to prune obviously failing disjuncts, so they need not be considered by a human. Section 5.2 describes how to transform the stream into disjunctive normal form (DNF), to provide a flat tree of disjuncts to choose from.

5.1 Pruning obvious failures

Some streams and goals in Appendix D are obviously failing. Examples include streams of the form `(bind #f _)`, `(mplus #f #f)`, goals of the form `(== A B)`, where `A` and `B` are unequal terms, and other streams and goals that depend on failing components. We would prefer to eliminate these immediately so we can focus on meaningful parts of the search state.

We can define a *pruning* transformation that uses equality information to identify constraints that will definitely fail. Since streams and goals depending on these constraints may fail to produce any results, we will prune them accordingly, producing a simplified stream.

Before taking a look at the implementation, it is important to consider that a transformation may change the observed behavior of our program. We should decide what sorts of changes we will tolerate. Viewing a miniKanren program as a search for answers that satisfy a query, we feel the following concessions are acceptable:

(1) Answers may be reordered.
(2) The contents of a state may be rearranged as long as its constraints express the same meaning.
(3) The number of steps required to find an answer may change, but will remain finite.
(4) Unproductive, infinite streams may become finite.

With that in mind, let us take a look at the implementation. The interpreters `prune/stream` and `prune/goal` partially evaluate == constraints, using the resulting `state` to propagate equality information. Any failing stream or goal will cause its parent `bind` and `conj` to also fail. A `mplus` or `disj` containing a failing child will be replaced by the remaining child.

```
(define (prune/stream s)
  (match s
    ((mplus s1 s2) (match (prune/stream s1)
                     (#f (prune/stream s2))
                     (s1 (match (prune/stream s2)
                           (#f s1)
                           (s2 (mplus s1 s2))))))
    ((bind s g)    (match (prune/stream s)
                     (#f          #f)
                     (`(,st . #f) (prune/goal st g))
                     ((pause st g1)
                    (match (prune/goal st g)
                      (#f         #f)
                      ((pause _ g) (pause st (conj g1 g)))))
                   (s (match (prune/goal empty-state g)
                        (#f         #f)
                        ((pause _ g) (bind s g))))))
    ((pause st g)  (prune/goal st g))
    (`(,st . ,s)   `(,st . ,(prune/stream s)))
    (s             s)))

(define (prune/goal st g)
  (define (prune/term t) (walk* t (state-sub st)))
  (match g
    ((disj g1 g2)
     (match (prune/goal st g1)
       (#f (prune/goal st g2))
       ((pause st1 g1)
        (match (prune/goal st g2)
          (#f          (pause st1 g1))
          ((pause _ g2) (pause st (disj g1 g2)))))))
    ((conj g1 g2)
     (match (prune/goal st g1)
       (#f           #f)
       ((pause st g1) (match (prune/goal st g2)
                        (#f           #f)
                        ((pause st g2) (pause st (conj g1 g2)))))))
    ((relate thunk d) (pause st (relate thunk (prune/term d))))
    ((== t1 t2)
     (let ((t1 (prune/term t1)) (t2 (prune/term t2)))
       (match (unify t1 t2 st)
         (#f         #f)
         (`(,st . #f) (pause st (== t1 t2)))))))))
```

Since pruning performs obvious clean up work without spending steps to do so, pruning may also lead to a trace containing fewer steps.

5.2 Disjunctive Normal Form

In Figure 1, we actually do one more thing—we provide the user with a flat list of choices. We can achieve this by transforming the stream into disjunctive normal form (DNF).

The general DNF transformation rewrites goals and streams in the following way:

```
(conj (disj A B) C)              ==> (disj (conj A C)
                                           (conj B C))

(conj C (disj A B))              ==> (disj (conj C A)
                                           (conj C B))
```

```
(pause st (disj A B))           ==> (mplus (pause st A)
                                           (pause st B)

(bind (mplus A B) C)            ==> (mplus (bind A C)
                                           (bind B C))

(bind C (disj A B))             ==> (mplus (bind C A)
                                           (bind C B))
```

This transformation has the effect of lifting all disjunctions out of conjunctions.

```
(define (dnf/stream s)
  (match s
    ((bind s g)
     (let loop1 ((s (dnf/stream s)) (g (dnf/goal g)))
       (define (loop2 s g)
         (match g
           ((disj ga gb) (mplus (loop2 s ga) (loop2 s gb)))
           (g            (bind s g))))
       (match s
         ((mplus sa sb) (mplus (loop2 sa g) (loop1 sb g)))
         (`(,st . ,s)   (mplus (pause st g) (loop1 s g)))
         (s             (loop2 s g)))))
    ((pause st g)  (let loop ((g (dnf/goal g)))
                     (match g
                       ((disj g1 g2) (mplus (loop g1) (loop g2)))
                       (g            (pause st g)))))
    ((mplus s1 s2) (mplus (dnf/stream s1) (dnf/stream s2)))
    (`(,st . ,s)   `(,st . ,(dnf/stream s)))
    (s             s)))

(define (dnf/goal g)
  (match g
    ((conj g1 g2)
     (let loop1 ((g1 (dnf/goal g1)) (g2 (dnf/goal g2)))
       (define (loop2 g1 g2)
         (match g2
           ((disj g2a g2b) (disj (loop2 g1 g2a) (loop2 g1 g2b)))
           (g2             (conj g1 g2))))
       (match g1
         ((disj g1a g1b) (disj (loop2 g1a g2) (loop1 g1b g2)))
         (g1             (loop2 g1 g2)))))
    ((disj g1 g2) (disj (dnf/goal g1) (dnf/goal g2)))
    (g            g)))
```

Aside from preparing a stream for choice extraction, DNF has the added benefit of allowing equality information to propagate more deeply, making pruning more effective. Consider the following goal:

```
(conj (disj (== x 5) A) B)
```

In this situation, information flow is impeded. Though we constrain x to be equal to 5, we cannot propagate this information into the goal B, because it is possible that goal A may not express the same constraint.

We can allow this information to flow more freely by lifting the disjunction out of the conjunction like so:

```
(disj (conj (== x 5) B)
      (conj A        B))
```

Because each child of the disjunction now has its own copy of the goal B, it is safe to propagate the equality information into the affected copy of B.

Another interesting property of DNF is that if we apply the full pruning transformation on a DNF stream, all remaining == constraints will be trivial in the sense that they are guaranteed to succeed, and will not provide new information. This is because all available equality information will have already been gathered in a pause state. These trivial constraints may be safely removed.

One downside of DNF transformation is that program representations can become larger due to copying children of conjunctions. In the worst case, the increase is exponential in the size of the original program. This potential increase in size may be mitigated by an increased likelihood of prunable, obvious failures, thanks to the improved flow of equality information.

Once we have a pruned, DNF stream, we are finally in a position to provide choices of the form shown in Figure 1. In section 6.1 we will explain how we extract this list of choices from such a stream.

6 APPLICATIONS

6.1 The miniKanren stepper

In this section, we implement |verb|explore/stream|, the miniKanren stepper described in Figure 1. The implementation of explore/stream is shown in Figure **??**, and the main stepping loop follows the outline given at the end of Section 3. We describe each facet of this loop in chronological order.

The first thing the explore/stream loop does is use the procedure stream->choices to extract a list of choices from a stream. To prepare for extraction, we transform the stream into DNF using dnf/stream, and perform pruning with prune/stream. Recall that these procedures were implemented in Section 5. With a pruned, DNF stream, we are guaranteed to find all disjunctions at the top (after any available answers), as a mplus tree, with leaves of the form (pause STATE GOAL). Each of these leaves represents a choice, and all choices can be extracted while traversing the tree.

Once we have our list of choices, we need to present each choice in a digestible form. We handle this with print-choice. The GOAL in each choice is either a trivial == constraint (i.e., both terms are guaranteed to be equal), a relate call, or a conj tree whose leaves are goals of this form. Traversing this conj tree, as in goal->constraints, extracts these goals as a list of the choice's constraints. As mentioned in Section 5.2, pruning in DNF has the emergent property that any remaining == constraints are guaranteed to be trivial, so we may discard them, allowing us to focus on the relate calls. Finally, pruning in DNF also guarantees that the only unknown logic variables referenced by the remaining constraints are reachable from the query variable values. Therefore, we can discard STATE, keeping only the query variable values.

With the interesting part of the implementation out of the way, all we have left to do is read user input and act accordingly. When the user makes a choice, we use step to expand the choice, providing the resulting stream as input to the next iteration of explore/stream.

```
(define (explore/stream qvars s)
  (define margin "|␣")
  (define (pp prefix v) (pprint/margin margin prefix v))
  (define (pp/qvars vs)
    (define (qv-prefix qv) (string-append "␣" (symbol->string qv) "␣=␣"))
    (define qv-prefixes (and qvars (map qv-prefix qvars)))
    (if qv-prefixes
      (for-each (lambda (prefix v) (pp prefix v)) qv-prefixes vs)
      (for-each (lambda (v) (pp "␣" v)) vs)))
  (define (print-choice s)
    (match s
      ((pause st g)
       (pp/qvars (walked-term initial-var st))
       (define cxs (walked-term (goal->constraints st g) st))
       (unless (null? cxs)
         (displayln (string-append margin "␣Constraints:"))
         (for-each (lambda (v) (pp "␣*␣" v)) cxs))
       (when (null? cxs)
         (displayln (string-append margin "␣No␣constraints"))))))
  (let loop ((s (stream->choices s)) (undo '()))
    (define previous-choice
      (and (pair? undo)
           (let* ((i.s (car undo)) (i (car i.s)) (s (cdr i.s)))
             (list-ref (dropf s state?) (- i 1)))))
    (define results (takef s state?))
    (define choices (dropf s state?))
    (display "\n========================================")
    (displayln "========================================")
    (unless (= (length results) 0)
      (printf "Number␣of␣results:␣~a\n" (length results))
      (for-each (lambda (st)
                  (pp/qvars (walked-term initial-var st))
                  (newline))
                results))
    (when (and previous-choice (null? results))
      (printf "Previous␣Choice:\n")
      (print-choice previous-choice)
      (newline))
    (printf "Current␣Depth:␣~a\n" (length undo))
    (if (= 0 (length choices))
      (if (= (length results) 0)
        (printf "Choice␣FAILED!␣␣Undo␣to␣continue.\n")
        (printf "No␣more␣choices␣available.␣␣Undo␣to␣continue.\n"))
      (printf "Number␣of␣Choices:␣~a\n" (length choices)))
    (for-each (lambda (i s)
                (printf (string-append "\n" margin "Choice␣~s:\n") (+ i 1))
                (print-choice s))
              (range (length choices)) choices)
    (printf "\n[h]elp,␣[u]ndo,␣or␣choice␣number>␣")
    (define (invalid)
      (displayln "\nInvalid␣command␣or␣choice␣number.\nHit␣enter␣to␣continue.")
      (read-line) (read-line)
      (loop s undo))
    (define i (read))
    (cond ((eof-object? i) (newline))
          ((or (eq? i 'h) (eq? i 'help))
           (displayln
             (string-append "\nType␣either␣the␣letter␣'u'␣or␣the"
                            "␣number␣following␣one␣of␣the␣listed␣choices."
                            "\nHit␣enter␣to␣continue."))
           (read-line) (read-line)
           (loop s undo))
          ((and (or (eq? i 'u) (eq? i 'undo)) (pair? undo))
           (loop (cdar undo) (cdr undo)))
          ((and (integer? i) (<= 1 i) (<= i (length choices)))
           (loop (stream->choices (step (list-ref choices (- i 1))))
                 (cons (cons i s) undo)))
          (else (invalid))))
```

6.2 Search Strategies

A first-order miniKanren with the small-step interpreter `step` gives us fine-grained control over the search strategy. Even though we begin Section 4.2 by reimplementing the biased-interleaving search, we can use whatever search strategy we would like. In this section, we describe a few different search strategies to illustrate the potential.

6.2.1 Human-guided search. One way of interpreting what we did in Section 6.1 is that we *hijacked the search strategy*. That is, the miniKanren stepper is actually an implementation of a human-guided search strategy! Since human behavior is as unpredictable as any program can be, the miniKanren stepper shows that any kind of search strategy is possible.

6.2.2 Neural-guided search. Another idea is to replace the human-guided search strategy with an artificial human, namely a neural network. In Zhang et al. [3] a neural network learns to guide the search of first-order miniKanren performing a Programming by Example (PBE) task. In this task, miniKanren is used to synthesize a procedure specified using input/output pairs.

For example, if we have a relational interpreter `(eval-expo program environment output)`, then an example PBE query with two input-output pairs is:

```
(query (p)
  (eval-expo p '(a) '(a a a))
  (eval-expo p '(b) '(b b b)))
```

The neural network evaluates each choice (disjunct) independently. It takes as input all the constraints corresponding to each choice, and chooses the highest-scoring choice to expand in each step.

There are several deviations from Section 6.1 required for this setup. First, unlike in Figure 1, the neural network can choose to expand any choice in each step, not just decendants of the previous step. So, the neural network would not need the "undo" functionality at all. This revised setup means that the neural network do not need to remember any past information, and can treat each step independently.

Another deviation is including a powerful strategy appropriate to PBE problems, which specializes the expansion behavior of `eval-expo` constraints. In a PBE problem, each input-output example imposes constraints on the same program to be synthesized, using `eval-expo`. However, a typical miniKanren search will attempt to satisfy each `eval-expo` constraint completely before moving on to the next. This is unfortunate because it means only information from one input-output example is being considered at any given time, leading to deep exploration of impossible synthesis choices that could have been ruled out immediately. Instead, we implement a search that will simultaneously expand all `eval-expo` constraints that share the same program argument. Combined with DNF transformation and pruning, this leads to immediate feedback being provided by all input-output examples for each synthesis choice made. Aside from refuting bad choices more quickly, this simultaneous expansion allows the neural network to observe related constraints all at once.

The network in Zhang et al. [3] is trained on generated PBE problems to which we know the answer, so that the correct choice is known for each step. During training, the neural network guides the search by making choices, and incorrect choices are penalized. We refer interested readers to Zhang et al. [3] for more details about the model architecture and training.

6.2.3 Dynamic Goal Reordering. Though we have mostly discussed transformations while developing the interactive stepper, there are other ways to improve a search strategy. One source of possibilities is Byrd et al. [1], which makes use of a relational interpreter that performs various forms of conjunction reordering based on heuristics specific to the interpreter definition. For instance, goals are reordered based on determinism annotations on `conde` expressions, where it may be known that at most one clause of a `conde` may succeed if we know the value of a particular subset of logic variables. This reordering would be more natural to implement with a decoupled representation, where it would be easier to compose with transformations and other strategies.

6.3 Other

Beyond improvements to the search, a first-order representation is a natural medium for program analysis and compilation. For instance, though we have not explored these ideas, we believe it should be possible to implement garbage collection of constraints and substitutions, and just-in-time compilation, possibly benefiting from mode analysis.

7 RELATED WORK

There have been previous unpublished implementations of miniKanren search using a first-order implementation. For example, Michael Ballantyne has implemented a small-step miniKanren interpreter that uses a first-order representation.[5] Ballantyne has also implemented a big-step interpreter with a first-order representation of the search tree, in order to implement backjumping search [6]. Both of these implementations decouple the program representation from the search.

To our knowledge, this paper is the first first-order representation of a miniKanren search that preserves the exact search order of the standard higher-order search representation, or that provides rules for a defunctionalization that preserves this search order.

8 CONCLUSION

Both higher-order and first-order representations of miniKanren have their advantages. For both pedagogical reasons and for the beauty and abstractness of the implementation, higher-order representations have been dominant in the miniKanren literature. On multiple occassions, however, implementors have had to switch to first-order representations in order to inspect or control the execution of miniKanren code.

[5]https://github.com/michaelballantyne/mk-interp/blob/master/mk.rkt
[6]https://github.com/michaelballantyne/backjumping-miniKanren/blob/master/backjumping2-defun4.rkt

Currently it is a burden to switch representations, not just because of the effort of defunctionalization, but because any changes to the search order may break existing tests, and make it difficult to compare benchmark results.

The rules for defunctionalization presented in Section 4.3, which preserve the exact search order, allow us to have the best of both worlds. We can have a higher-order implementation for pedagogical purposes, for example, and switch to a first-order implementation when it is time to debug or trace the code. And we can be confident that our existing tests, applications, and benchmark programs will not need to change. Of course, while we are living in the first-order world, we can easily change the search, or use an external process to guide the search.

We hope that there will be two positive outcomes to making it easier to switch between higher-order and first-order implementations of miniKanren. First, the wider availability of first-order implementations should make it much easier to produce the tools that the miniKanren ecosystem desperately needs, such as tracers, steppers, and debuggers. The rules in Section 4.3 imply that a programmer can debug a program in a first-order implementation of miniKanren, and be sure the program would show equivalent behavior in a higher-order miniKanren. Secondly, by decoupling the search from the program representation, it should be easier to experiment with novel search techniques, such as driving the search from an external process, or mixing different searches within a single miniKanren program. This should make it easier and faster to explore program synthesis and other advanced relational programming topics.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number 3OT2TR002517-01S1.

REFERENCES

[1] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

[2] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A minimal functional core for relational programming. In *Scheme and Functional Programming Workshop 2013*.

[3] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*. 1737–1746.

A STEPPER OUTPUT FOR APPENDO

This appendix shows the interactive stepper output for the query `(query (x y) (appendo x y '(1 2 3)))` of a correctly implemented appendo shown in Section 2. Compare this output with that of a buggy appendoh implementation in Figure 1

```
Stepping through query: (query (x y) (appendo x y '(1 2 3)))
================================================================================
Current Depth: 0             Number of Choices: 1

| Choice 1:
|  x = #s(var x 1)
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var x 1) #s(var y 2) (1 2 3))

[h]elp, [u]ndo, or choice number> 1
================================================================================
Current Depth: 1             Number of Choices: 2

| Choice 1:
|  x = ()
|  y = (1 2 3)
|  No constraints

| Choice 2:
|  x = (1 . #s(var b 4))
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var b 4) #s(var y 2) (2 3))

[h]elp, [u]ndo, or choice number> 2
================================================================================
Current Depth: 2             Number of Choices: 2

| Choice 1:
|  x = (1)
|  y = (2 3)
|  No constraints

| Choice 2:
|  x = (1 2 . #s(var b 7))
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var b 7) #s(var y 2) (3))
```

B IMPLEMENTATION OF APPENDOH

This appendix shows the implementation of the buggy appendoh that generated the stepper output in Figure 1.

```
(define-relation (appendoh l s ls)
  (conde
    ((== '() l) (== s ls))
    ((fresh (a d res)
       (== `(,a . ,d) l)
       (== `(,a . ,res) ls)
       (appendoh d s ls)))))
```

C RECOVERING MINIKANREN

We show an implementation miniKanren on top of μKanren. This version of miniKanren is compatible with both the higher-order μKanren implemented in Section 4.3, and the first-order μKanren implemented in Section 4.2.

```
(define-syntax define-relation
  (syntax-rules ()
    ((_ (name param ...) g ...)
     (define (name param ...)
       (relate (lambda () (fresh () g ...)) `(,name name ,param ...))))))
;; Low-level goals
(define succeed (== #t #t))
(define fail    (== #f #t))
(define-syntax conj*
  (syntax-rules ()
    ((_)                succeed)
    ((_ g)              g)
    ((_ gs ... g-final) (conj (conj* gs ...) g-final))))
(define-syntax disj*
  (syntax-rules ()
    ((_)          fail)
    ((_ g)        g)
    ((_ g0 gs ...) (disj g0 (disj* gs ...)))))
;; High level goals
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 gs ...)
     (let ((x (var/fresh 'x)) ...) (conj* g0 gs ...)))))
(define-syntax conde
  (syntax-rules ()
    ((_ (g gs ...) (h hs ...) ...)
     (disj* (conj* g gs ...) (conj* h hs ...) ...))))
;; Queries
(define-syntax query
  (syntax-rules ()
    ((_ (x ...) g0 gs ...)
    (let ((goal (fresh (x ...) (== (list x ...) initial-var) g0 gs ...)))
      (pause empty-state goal)))))
(define (stream-take n s)
  (if (eqv? 0 n) '()
    (let ((s (mature s)))
      (if (pair? s)
        (cons (car s) (stream-take (and n (- n 1)) (cdr s)))
        '()))))
(define-syntax run
  (syntax-rules ()
    ((_ n body ...) (map reify/initial-var (stream-take n (query body ...))))))
(define-syntax run*
  (syntax-rules () ((_ body ...) (run #f body ...))))
```

D RAW APPENDO TRACE

This appendix shows how the search representation changes as we step through the query `(query (xs) (appendo xs xs '(a a)))` using interleaving search. We need to transform this representation to recover the easier-to-read textual representations in Figure 1 and Appendix A.

```
Step 0:
(pause
 (state ())
 (conj
  (== (#s(var xs 1)) #s(var #f 0))
  (relate (appendo #s(var xs 1) #s(var xs 1) (a a)))))

Step 1:
(mplus
 (bind #f (relate (appendo #s(var xs 1) #s(var xs 1) (a a))))
 (pause
  (state ((== #s(var #f 0) (#s(var xs 1)))))
  (disj
   (conj (== #s(var xs 1) ()) (== #s(var xs 1) (a a)))
   (conj
    (conj
     (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
     (== (#s(var a 2) . #s(var c 4)) (a a)))
    (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4)))))))

Step 2:
(pause
 (state ((== #s(var #f 0) (#s(var xs 1)))))
 (disj
  (conj (== #s(var xs 1) ()) (== #s(var xs 1) (a a)))
  (conj
   (conj
    (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
    (== (#s(var a 2) . #s(var c 4)) (a a)))
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))))

Step 3:
(mplus
 (pause
  (state ((== #s(var #f 0) (#s(var xs 1)))))
  (conj
   (conj
    (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
    (== (#s(var a 2) . #s(var c 4)) (a a)))
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4)))))
 (bind #f (== #s(var xs 1) (a a))))

Step 4:
(mplus
 (bind #f (== #s(var xs 1) (a a)))
 (mplus
  (bind
   (mplus (bind #f (== (#s(var a 2) . #s(var c 4)) (a a))) #f)
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))
  (pause
   (state
    ((== #s(var c 4) (a))
     (== #s(var a 2) a)
     (== #s(var xs 1) (a . #s(var b 3)))
     (== #s(var #f 0) ((a . #s(var b 3))))))
   (disj
    (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
    (conj
     (conj
      (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
      (== (#s(var a 5) . #s(var c 7)) (a)))
     (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7))))))))

Step 5:
(mplus
 (bind
  (mplus (bind #f (== (#s(var a 2) . #s(var c 4)) (a a))) #f)
  (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))
 (pause
  (state
   ((== #s(var c 4) (a))
    (== #s(var a 2) a)
    (== #s(var xs 1) (a . #s(var b 3)))
    (== #s(var #f 0) ((a . #s(var b 3))))))
  (disj
   (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
   (conj
    (conj
     (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
     (== (#s(var a 5) . #s(var c 7)) (a)))
    (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7)))))))

Step 6:
(pause
 (state
  ((== #s(var c 4) (a))
   (== #s(var a 2) a)
   (== #s(var xs 1) (a . #s(var b 3)))
   (== #s(var #f 0) ((a . #s(var b 3))))))
 (disj
  (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
  (conj
   (conj
    (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
    (== (#s(var a 5) . #s(var c 7)) (a)))
   (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7))))))

Step 7:
Answer: (state ((== #s(var #f 0) ((a)))))
Remaining stream:
(mplus
 (pause
  (state
   ((== #s(var c 4) (a))
    (== #s(var a 2) a)
    (== #s(var xs 1) (a . #s(var b 3)))
    (== #s(var #f 0) ((a . #s(var b 3))))))
  (conj
   (conj
    (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
    (== (#s(var a 5) . #s(var c 7)) (a)))
   (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7)))))
 (mplus (bind #f (== #s(var xs 1) #s(var c 4))) #f))
```

E PRUNING

We show the code used to elimiated obviously failing goals from our stream.

```
(define (prune/stream s)
  (match s
    ((mplus s1 s2) (match (prune/stream s1)
                     (#f (prune/stream s2))
                     (s1 (match (prune/stream s2)
                           (#f s1)
                           (s2 (mplus s1 s2))))))
    ((bind s g)    (match (prune/stream s)
                     (#f          #f)
                     (`(,st . #f) (prune/goal st g))
                     ((pause st g1)
                      (match (prune/goal st g)
                        (#f          #f)
                        ((pause _ g) (pause st (conj g1 g)))))
                     (s (match (prune/goal empty-state g)
                          (#f          #f)
                          ((pause _ g) (bind s g))))))
    ((pause st g)  (prune/goal st g))
    (`(,st . ,s)   `(,st . ,(prune/stream s)))
    (s             s)))

(define (prune/goal st g)
  (define (prune/term t) (walk* t (state-sub st)))
  (match g
    ((disj g1 g2)
     (match (prune/goal st g1)
       (#f (prune/goal st g2))
       ((pause st1 g1)
        (match (prune/goal st g2)
          (#f           (pause st1 g1))
          ((pause _ g2) (pause st (disj g1 g2)))))))
    ((conj g1 g2)
     (match (prune/goal st g1)
       (#f             #f)
       ((pause st g1) (match (prune/goal st g2)
                        (#f             #f)
                        ((pause st g2) (pause st (conj g1 g2)))))))
    ((relate thunk d) (pause st (relate thunk (prune/term d))))
    ((== t1 t2)
     (let ((t1 (prune/term t1)) (t2 (prune/term t2)))
       (match (unify t1 t2 st)
         (#f          #f)
         (`(,st . #f) (pause st (== t1 t2))))))))
```

F TRANSFORMATION TO DISJUNCTIVE NORMAL FORM

We show the code used to transform our streams into a Disjunctive Normal Form (DNF).

```
(define (dnf/stream s)
  (match s
    ((bind s g)
     (let loop1 ((s (dnf/stream s)) (g (dnf/goal g)))
       (define (loop2 s g)
         (match g
           ((disj ga gb) (mplus (loop2 s ga) (loop2 s gb)))
           (g            (bind s g))))
       (match s
         ((mplus sa sb) (mplus (loop2 sa g) (loop1 sb g)))
         (`(,st . ,s)   (mplus (pause st g) (loop1 s g)))
         (s             (loop2 s g)))))
    ((pause st g)  (let loop ((g (dnf/goal g)))
                     (match g
                       ((disj g1 g2) (mplus (loop g1) (loop g2)))
                       (g            (pause st g)))))
    ((mplus s1 s2) (mplus (dnf/stream s1) (dnf/stream s2)))
    (`(,st . ,s)   `(,st . ,(dnf/stream s)))
    (s             s)))

(define (dnf/goal g)
  (match g
    ((conj g1 g2)
     (let loop1 ((g1 (dnf/goal g1)) (g2 (dnf/goal g2)))
       (define (loop2 g1 g2)
         (match g2
           ((disj g2a g2b) (disj (loop2 g1 g2a) (loop2 g1 g2b)))
           (g2             (conj g1 g2))))
       (match g1
         ((disj g1a g1b) (disj (loop2 g1a g2) (loop1 g1b g2)))
         (g1             (loop2 g1 g2)))))
    ((disj g1 g2) (disj (dnf/goal g1) (dnf/goal g2)))
    (g            g)))
```