Relational Interpreters in miniKanren (Fascicles I&II)

William E. Byrd

February 7, 2026

Contents

An Unreasonable Schemer Production

This combination of Fascicles I&II is Book 07 of Will's ShovelWare Book Pack

Edition 1 (Version 1.0) Sunday, February 7, 2026

For Dan.

About Will's ShovelWare Book Pack

In an experiment to see if I can train myself to finish writing a book on my own—instead of just poking at it for a few weeks before abandoning it and starting another—I have publicly committed to writing and releasing seventeen (17) books in one year, releasing one book every three weeks. I made the original announcement on Will Radio Part XXVIII (https://www.youtube.com/watch?v=d6_cB-jtxYI) on September 15, 2025, with the experiment to end on September 6, 2026, when the 17th book is due.

Good or bad, short or long, my goal is to finish and release a book every 21 days, like clockwork.

The idea for this experiment came from my kiloTube Video challenge from a couple years ago. I had been wanting to make more videos, but was suffering from perfectionism—I would record 70 or 80 takes of a video before deleting it in disgust. So I publicly announced that I was going to make 1,024 (or 2^{10}) videos in a single year (one kiloTube worth), in an attempt to force myself to just make video after video, with the idea that I'd eventually learn the mechanics, and learn my own style for making videos. Although I didn't come close to making 1,024 videos that year (partly because making videos involved making noise, which caused some social issues), the practice of making an average of 3 videos a day for weeks quickly got me over my perfectionism block. I now have no trouble sitting down and making a video in one take, editing it, and uploading it in a single sitting.

My hope is that after finishing and releasing a dozen or so books, I'll start to relax out a little, and will be over the fear of releasing something less than perfect. There's a go adage, "Lose your first hundred games as quickly as possible." I'm trying to write and release my first dozen (or so) terrible books as quickly as possible. With 17 books to write, I might write a few decent ones, as well.

I am allowing myself to update books I've already released, so long as I continue releasing a new book every 3 weeks.

This is the seventh of the 17 books, due on Sunday, February 8, 2026, Anywhere on Earth (or whenever I go to sleep!). However, I'm releasing this book on Saturday, February 7, since I'll be traveling the next week. (Originally I was going to release the books on Monday nights, but Sundays work better.) The next book, book 08, will be due Sunday, March 1, 2026.

Preface

This book explores an unusual approach to writing interpreters. An *interpreter* is a computer program that can evaluate an expression, producing a value. This is a very general idea, and an amazing variety of computer programs can be considered interpreters of some form. The most expressive and powerful class of interpreters can evaluate expressions that represent arbitrary computer programs; such interpreters can even evaluate themselves! Because interpreters appear in so many areas of computation and computer science and programming, they are well worth studying and exploring, from both typical and unusual perspectives.

The unusual point of view taken in this book is that interpreters should be viewed as—and implemented as—*relations* rather than as *functions*. The typical perspective is that an interpreter is a function (or machine) that takes as an *input* an expression e (in some language) and produces as an *output* some value v. The notions of input and output are central to this view of an interpreter. The view taken in this book is that the distinction between input and output is unnecessary and overly restrictive. Instead, we view an interpreter as *relating* one or more expressions with one or more values.

For the most part we will write interpreters that interpret expressions in simple variants of the Scheme programming language. The interpreters themselves, however, will generally be written in miniKanren, a programming language designing for writing programs as relations. We will use the faster-miniKanren implementation of miniKanren, which is itself written in Scheme. Sometimes we'll do something trickier, such as writing a Scheme interpreter that is interpreted by a different Scheme interpreter that is written in the faster-miniKanren implement of miniKanren that is running in Scheme! This can get confusing, so we'll build up in complexity a little at a time, and draw diagrams and give examples and exercises to help keep everything straight.

The faster-miniKanren implementation of miniKanren we'll be using runs well in both Chez Scheme and in Racket. The book begins with a brief introduction to the faster-miniKanren version of the miniKanren programming language, and instructions on how to run faster-miniKanren in Chez Scheme and in Racket.

Intended audience and background knowledge

This book is intended for anyone interested in exploring a *relational programming* view of interpreters and interpretation.

However, this is *not* a beginning programming book, nor is it a self-contained course in miniKanren, relational programming, Scheme, or interpreters. This book assumes that the reader has basic knowledge of the Scheme programming language, basic functional programming (including pattern matching), and basic knowledge of how to write an interpreter for Scheme, in Scheme. The book also assumes the reader understands basic relational programming in miniKanren—for example, as provided by the second edition of Friedman, Byrd, Kiselyov, and Hemann's *The Reasoned Schemer* (MIT Press, 2018), or from the various miniKanren tutorials and videos available online or in academic papers.

As much as I'd like to include all of the required background information in a single book (and I *have* tried, several times), to adequately cover this material would require presenting the equivalent of an entire series of university courses. It is my dream to eventually write such a series, but I think it makes sense to start with the most unique material first, and then work backwards.

Fortunately, there are many fine books, tutorials, videos, and courses that teach the Scheme programming language, functional programming, pattern matching, program transformations, and interpreters. A few well-known books include: *Structure and Interpretation of Computer Programs, 2nd edition* by Harold Abelson and Gerald Jay Sussman, with Julie Sussman (MIT Press, 1996); *The Scheme Programming Language, 4th edition* by R. Kent Dybvig (MIT Press, 2009); *Essentials of Programming Languages, 3rd edition* by Daniel P. Friedman and Mitchell Wand (MIT Press, 2008); *The Little Schemer, 4th edition* by Daniel P. Friedman and Matthias Felleisen (MIT Press, 1995); *Lisp in Small Pieces* by Christian Queinnec (Cambridge University Press, 2003); and *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002)

Pointers to miniKanren-related resources can be found at https://minikanren.o rg/. The standard introductory book on miniKanren is *The Reasoned Schemer, 2nd edition* by Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann (MIT Press, 2018). (Full disclosure: I'm one of the authors of both editions of *The Reasoned Schemer*!)

Goals of this book

In addition to showing the basics of how to write interpreters as relations, a goal of this book is to show a *way of thinking* about writing non-trivial relational programs in miniKanren. In other words, to reify the implicit knowledge shared by miniKanren hackers that hasn't been written down, or is spread out among talks, papers, and dissertations. What does an experienced miniKanren hacker/researcher think about when writing or improving a relational program? What are the key ideas, techniques, traps, and pitfalls? What are alternative approaches to standard problems, and what are their tradeoffs? How do we think about relationality? How do we decide when a new constraint is warranted? Which problems do we seem to have some handle on, and which do we try to avoid or consider core research problems? Why do we use a more sophisticated approach or encoding when a simpler, more direct approach seems like it might work? (To give a specific example, why do we tend to write big-step interpreters rather than small-step term reducers, and is that preference due to an essential difference, or just to the lack of our understanding and technique?)

I will have succeeded, at least in part, if readers of this book come away with the ability to critique and improve relational programs, especially relational interpreters. I hope readers also will understand the motivation behind topics that interest miniKanren researchers that might otherwise appear esoteric, such as nominal logic programming, higher-order logic programming, and the Extended Andorra model. Readers should be able to pick up miniKanren-related research papers and get a sense of how those works fits into the bigger picture of relational programming, especially with respect to relational interpretation.

How this book is being written

This is a book “in progress” on writing interpreters are relations in miniKanren. The idea is to release one fascicle, or “bundle,” every three weeks for the remainder of the ShovelWare Book Pack experiment. The first fascicle included the draft of this preface. I plan to release one fascicle (which may or may not correspond to a chapter) every three weeks. I should end up with 12 fascicles worth of work on this book by September 15, 2025, when the 17th ShovelWare book is due. I can then either revise the book, continue extending it, move onto another book, or take a break.

Originally I had planned on releasing the book in old-school serial fashion by concatenating the fascicles as I write them, releasing the latest version of the book’s “prefix.” After a few days of work on the second fascicle, though, I think I’ll try doing passes over the entire book, refining and expanding every chapter with each complete pass in stages: rough notes for each chapter, refined into an outline for each chapter, refined into a complete draft down to the paragraph level for each chapter, refined into a complete draft down to the sentence level for each chapter, refined into completed chapters. (I did something similar with my PhD dissertation.) One way to think about how I’m writing this book is as notes for teaching a course, being refined over many passes.

Acknowledgments

Marco Marchesin suggested releasing fascicles of a larger book instead of writing one self-contained tiny book each week as part of my ShovelWare Book Pack experiment, which I’m attempting with this book. Thank you, Marco!

The first five ShovelWare books I’ve released, as bad and incomplete as they may be, helped me get in the position mentally to try this approach. So, thank you to those books, and to anyone who subjected themselves to reading them.

A proper acknowledgment of all the work on miniKanren and relational interpreters would fill many pages. As always, there is the danger of leaving people out, or mixing up contributions from years ago. I will write a proper acknowledgments section, although I know from experience it may take multiple fascicles before I dare show the acknowledgments publicly. For now, let me thank everyone in the miniKanren community, especially my teachers, mentors, collaborators, and students.

Will Byrd

Mt. Pleasant, South Carolina

February, 2026

(Rough high-level outline, subject to change)

Here is the current high-level outline for this book, subject to reordering, addition, and deletion:

1. Brief refresher of core miniKanren and relational programming, brief description of the faster-miniKanren implementation and the language it supports (including constraints other than `==`), and a few examples of relations using this language (with pointers to resources that cover these topics);
2. techniques for debugging miniKanren relations;
3. design and write Scheme macros that provide alternative interfaces to `run n` and `run*` that return *sets* rather than *lists*, and for which the `run*` equivalent uses depth-first search rather than interleaving search, along with justification for why this is worth experimenting with;
4. refresher of a simple environment-passing Scheme interpreter, written in Scheme, using a first-order representation of procedures and environments, written using pattern matching and quasiquotation, and supporting lexical scope and shadowing (with pointers to resources that cover these topics);
5. techniques for debugging interpreters in Scheme;
6. introduce simple tagged representation of expressions and values, and simple association list representation of environments, working up to a simple relational environment-passing Scheme interpreter using a fully tagged representation of Scheme expressions and values;
7. use constraint extensions to core miniKanren, along with the `not-in-envo` helper relation, to untag the representations of Scheme variables, numbers, procedure application, and add `quote` to the interpreter (with the help of `absento` to prevent quoted closures as expressions), working up to an untagged relational Scheme interpreter that can synthesize values such as `(I love you)`, along with Quines, twines, and thrines (using nother `absento` trick to generate more interesting twines and thrines);
8. techniques for debugging interpreters in miniKanren;
9. (as an aside from improving the relational Scheme interpreter) explore tradeoffs of tagged versus untagged representations, including the issues of writing parsers/unparsers for tagged terms containing fresh variables in tag position, and investigate a fully-tagged version of the Quine-generating relational interpreter that uses `==` as the only constraint, first explored by Nada Amin;
10. (as an aside from improving the relational Scheme interpreter) explore what it means to be relational, the role of constraints in expressing limited forms of negation and avoiding branching, unbounded recursion/unbounded number of successes/enumerating infinite (or gigantic) domains, discuss

the role of constraints in the relational interpreter, and discuss when and how to design and implement a new constraint;

11. explore issues around the apparently attractive approach of using logic variables and first-order syntactic unification to represent lexical scoping, binding, and α-equivalence with tagged lists representing λ terms:

    ```
    (fresh (a b c d)
      (== `(lambda (,a) (lambda (,b) ,a))
          `(lambda (,c) (lambda (,d) ,d))))
    ```

    or:

    ```
    (fresh (a b c)
      (== `(lambda (,a) (lambda (,a) ,a))
          `(lambda (,b) (lambda (,c) ,b))))
    ```

    and the resulting motivation for nominal unification, or other approaches;

12. explore alternative environment representations, including a "split" two-list representation, and another representation using association list constraints, working up to a relational interpreter with `lookupo` and `not-in-envo` relations that succeed finitely many times for environments that are not length instantiated;

13. write relational interpreters that handle mutation using store-passing style;

14. write interpreters with explicit handling of errors, showing how angelic execution can be used to synthesize expressions that, when evaluated, generate a given error;

15. add `letrec` and recursive environments, and add a simple pattern matcher to the relational Scheme interpreter, taking special care to explicitly handle the cases of not matching (which is the trickier part!), working up to a relational Scheme interpreter that can interpret `append`, can synthesize arguments to `append` using idioms like `(append '(a b c) ',x)`, and can synthesize small code expressions inside the definition of `append`;

16. discuss the similarities and differences between `append` in Scheme, `appendo` in miniKanren, and `append` in the relational Scheme interpreter running inside miniKanren;

17. explore a neat example from the 2017 ICFP pearl: synthesizing an expression that evaluates to different values under lexical scope versus dynamic scope through sharing a fresh variable in the expression position for two relational interpreters, and using a disequality constraint to enforce the the value arguments differ;

18. write a relational Scheme "anti"-interpreter, `not-evalo`, where `(not-evalo expr val)` succeeds when Scheme expression `expr` does *not* evaluate to value `val`;

19. write relational Scheme interpreters that implement call-by-name and call-by-need, rather than call-by-value;
20. write a relational Scheme interpreter that supports memoized functions;
21. write a relational abstract interpreter;
22. add delayed goals to faster-miniKanren, and explore uses of—and tradeoffs of—delayed goals in relational interpreters;
23. explore techniques for radically speeding up the interpreter in certain cases, such as dynamic reordering of conjunctions of recursive calls based on groundness of terms, and using a Scheme function rather than a miniKanren relation for variable lookup in the interpreter's environment when the arguments are sufficiently ground, working up to a relational Scheme interpreter that can synthesize `append` from expression/value pairs;
24. explore synthesizing `append` and other recursive definitions using properties such as *associativity*, rather than only input/output pairs;
25. explore sophisticated example programs from the 2017 ICFP pearl: proof checker becomes theorem prover via relational interpretation, and quasiquote Quine generation using a micro-Scheme in mini-Scheme in miniKanren in Scheme;
26. build a simple clone of Barliman, including Barliman's technique of trying to parse an expression relationally in the hope of failing fast, and throwing away any information gained upon success to avoid premature grounding before calling the evaluator, and also running each input/output pair on its own, in parallel with the entire synthesis problem (once again, hoping to fail fast);
27. (as an aside from implementing relational interpeters) discuss lack of a relational hygienic macro expander, its implications, possible work arounds, and open problems;
28. (as an aside from implementing relational interpeters) discuss relational compiler, rather than interpreter, for Scheme, along with tradeoffs, opportunities, and problems;
29. use the staged-miniKanren system to perform staged evaluation of relational interpreters, giving cool examples from the PLDI paper and repo;
30. consider the apparently simple notions of free and bound variables from a relational context, implementing C311-style functions relationally using setKanren/CLP(Set);
31. consider the apparently simple task of adding arithmetic to a relational Scheme interpreter, and explore the various ways this may be done, including using variable projection, delayed goals, tagged Peano numerals, "Kiselyov" numerals, finite domain constraints, and CLP(SMT);

32. write a small-step λ-calculus reducer, based on capture-avoiding substitution, exploring issues related to names and binding;
33. write a small-step λ-calculus reducer, using De Bruijn notation, and explore issues and tradeoffs;
34. write a small-step term reducer for the SKI combinator calculus, and explore the tradeoffs in terms of avoiding nominal issues with capture-avoiding substitution and α-equivalence, and the difficulties related to failing fast, showing Ω as a small-step Quine, working up to synthesizing combinator terms and fixpoint combinators (Question: can this reducer be staged by staged-miniKanren?);
35. explore the technique of normalization-by-evaluation (or *NBE*) from a relational standpoint, resulting in two relations, `evalo` and `unevalo`, that together can synthesize a fixpoint function in Scheme (and explore technique for ensuring freshness of lexical variables using `absento` and a list of variables encountered so far (which do not have to be ground));
36. run a slightly massaged version of the Scheme code for microKanren in the Barliman-style relational Scheme interpreter, then explore the behavior of microKanren programs in this relational context, including the behavior of extra-logical operators such as `varo`;
37. implement shallow, medium, and deep embeddings of miniKanren-in-miniKanren, and run queries that share arguments between shallower and deeper embeddings to expore tradeoffs between performance versus expressive power;
38. explore the relational behavior of extra-logical operators in a deep embedding of miniKanren in miniKanren, including aggregate operators inspired by Prolog's `findall` and `setof` predicates, implemented using setKanren/CLP(Set).

Worth thinking about for each chapter:

- Concepts and themes
- Techniques
- Code
- Examples
- Exercises
- Research problems

Brief refresher of core miniKanren and relational programming

Brief refresher of core miniKanren and relational programming, brief description of the faster-miniKanren implementation and the language it supports (including constraints other than `==`), and a few examples of relations using this language (with pointers to resources that cover these topics)

core miniKanren, in this order: `==`, `run*`, `conde`, `fresh`, `run n`

`appendo`

reordering of goals

faster-miniKanren—Chez Scheme and Racket (what about Schemes other than Chez?)

`=/=`, `symbolo`, `numbero`, `stringo`, `absento`

disjointness of type constraints

common confusion about what `=/=` means; common confusion about `=/=` versus `absento`

`project`, `onceo`, `conda`, `condu`

difference in behavior between TRS1 and “modern” miniKanrens

what about “classic” miniKanren implementations, other than faster-miniKanren? Also, staged-miniKanren, clpset-faster-miniKanren, microKanren implementation (and variant in TRS2), TRS1 and TRS2 implementations (and pointers to code), etc.

Techniques for debugging miniKanren relations

printing/tracing, both using `project` explicitly and by writing a helper macro, using all fresh vars as arguments, using all ground arguments, `run 1` when `run*` or `run n` for $n > 1$ is taking a long time, comment out recursive clauses and try with all fresh vars, replace recursive calls with a succeeding goal

Alternative `run` interfaces for miniKanren

design and write Scheme macros that provide alternative interfaces to `run n` and `run*` that return *sets* rather than *lists*, and for which the `run*` equivalent uses depth-first search rather than interleaving search, along with justification for why this is worth experimenting with

introduce new `run-set n` and `run-set*` syntax, and keep existing `run n`/`run*` syntax

`run-set*` and DFS (which would be enabling for both more efficient incremental CLP(SMT) calling, and easier tracing/debugging)

sets in Chez and Racket

start with existing macros

macro part versus function part

`take`, streams, and all of that

how sets differ from multisets

`run-set n` runs until *n distinct* answers are produced: `(run-set 2 (q) (fives-forevero q))` would diverge for this reason, while the `run 2` variant would return the list `(5 5)`, which seems like a strong argument for `run-set n` and `run-set*` to be distinct from `run n` and `run*`, instead of replacing them

overlapping answers/subsumption, etc.

(connections with and differences from clpset-faster-miniKanren/CLP(Set))

show how `run-set*` can be faster/use less memory than `run-set 1`, `run*`, and `run 1`, due to DFS versus interleaving search

`run-set n` answers as a subset of `run-set*` answers (if they both terminate)

if using CLP(SMT), the DFS nature of `run-set*` could enable efficient incremental calling of the solver, possibly resulting in better fail-fast behavior of `run-set*` than `run-set n`, `run*`, or `run n`.

Refresher of a simple environment-passing Scheme interpreter, written in Scheme

refresher of a simple environment-passing Scheme interpreter, written in Scheme, using a first-order representation of procedures and environments, written using pattern matching and quasiquotation, and supporting lexical scope and shadowing (with pointers to resources that cover these topics)

variables, binding, environment, shadowing, lexical scope

representing a variable as a symbol, equality comparison for comparing variable equality

shadowing via `cons`, look up, empty environment

higher-order versus association list versus tagged list representation, and lookup

Pattern matching, `quasiquote`, `unquote`, call-by-value λ-calculus core, single-argument λ and application

tagged/untagged procedure application

first-order/higher-order procedure and environment representations

representation independence

Techniques for debugging interpreters in Scheme

Simple relational environment-passing Scheme interpreter using a fully tagged representation of Scheme expressions and values

introduce simple tagged representation of expressions and values, and simple association list representation of environments, working up to a simple relational environment-passing Scheme interpreter using a fully tagged representation of Scheme expressions and values

Tagged expressions, tagged application, tagged values, lookupo (and its behavior), simple language, issues with fresh logic variables and tag positions

write parser/unparser, and show issues with those programs when fresh variables are involved

Shadowing and duplicate answers (shadowing built-in primitives like `cons`)

show synthesis

Alas, can't just go to or from Scheme (motivating parser and parser, and eventually motivating desire for untagged representation).

Duplicate answers due to shadowing of primitives, leading to desire for `not-in-envo`.

Call-by-value λ-calculus, plus `cons` and `quote` (Maybe hold off on `quote`, until tagging is removed in a subsequent chapter.

issue with quoted closures.

(Do I want to break some of these issues down to be addressed in different chapters?)

Untagged relational Scheme interpreter that can generate Quines

use constraint extensions to core miniKanren, along with the `not-in-envo` helper relation, to untag the representations of Scheme variables, numbers, procedure application, and add `quote` to the interpreter (with the help of `absento` to prevent quoted closures as expressions), working up to an untagged relational Scheme interpreter that can synthesize values such as `(I love you)`, along with Quines, twines, and thrines (using another `absento` trick to generate more interesting twines and thrines)

`(I love you)` synthesis

Shuould already have introduced `not-in-envo` to prevent duplicate answers due to shadowing of primitives. Will also need it for untagging procedure application

Untagged procedure application only leads to overlapping behavior if the expression for application overlaps with that of the other interpreter clauses (if there lengths overlap, basically)

So, `(lambda (,x) ,e)` doesn't overlap with `(,e1 ,e2)`

of course, once we have multi-argument λ and application, supporting with any number of arguments, the application case will overlap with all of the other cases that have non-empty list sytax (won't overlapped with `numbero` or `symbolo` or `stringo` cases that use type constraints)

(where do we put multi-argument and variadic λ and application?)

`quote`, quoted closures, whack-a-mole once you have `cons`, `car`, `cdr`. `absento` on the `closure` tag. where to put `absento` call? driver relations vs. ensuring helpers are self contained

```
(run 1 (q r)
  (=/= q r)
  (evalo q r)
  (evalo r q))
```

Every Quine is a trivial twine. Need the disequality constraint for non-triviality

(This chapter has a lot, maybe break it down)

Techniques for debugging interpreters in miniKanren

Tradeoffs of tagged versus untagged representations

(as an aside from improving the relational Scheme interpreter) explore tradeoffs of tagged versus untagged representations, including the issues of writing parsers/unparsers for tagged terms containing fresh variables in tag position, and investigate a fully-tagged version of the Quine-generating relational interpreter that uses `==` as the only constraint, first explored by Nada Amin

Revisit Quine generation using fully tagged representation, using Nada Amin's `==`-only approach. Issues with that approach, possible fixes, and possible remaining issues.

(An example of critique of a relational interpreter, and of relational programs in general.)

Formal verification or proof would be very useful in this area.

Can I recapitulate the confusion with fully tagged terms, including tagged application and variables, in generated Quines, from Clojure/conj? Back then, I was never quite sure which terms were legal, what was a legal Quine, etc. Made me want to get rid of the tags and be able to run synthesized code directly in Scheme, and to run Scheme code directly in the relational intepreter. (An approach I think has served us well.)

What it means to be relational, and the role of constraints

(as an aside from improving the relational Scheme interpreter) explore what it means to be relational, the role of constraints in expressing limited forms of negation and avoiding branching, unbounded recursion/unbounded number of successes/enumerating infinite (or gigantic) domains, discuss the role of constraints in the relational interpreter, and discuss when and how to design and implement a new constraint

Peano numerals: `‘(S ,n)` is one greater than `n`, and is therefore larger than `n`, and is also therefore a *positive* integer. Even though we don't know what the value of `n` is, we can do at least some reasoning *without* grounding or enumerating `n`. This abstract, numeral represents infinitely many concrete, ground numerals: `(s z)`, `(s (s z))`, `(s (s (s z)))`, ...

`zeroo` and `poso`: give their definitions, and show how ‘(s ,n) works with them.

Notions of relationality: reordering conjuncts and disjuncts, swapping argumentsto `==` (and to `=/=`), the ability to use ground, fresh, or partially ground terms for any argument

show why `mapo` is not relational. Similarly for `project`.

Limited notions of negation with constraints. Punching out one value in an infinite domain, versus `booleano`, which brings you into finite domain constraints.

The art, or trick, of effective relational programming is to come up with data representations and use constraints (perhaps create new constraints) to represent infinitely many possibilities (or gigantic but finite numbers of possibilities), thereby avoiding enumerating all of the possibilities one-by-one, which is hopeless.

```
(fresh (n) (numbero n))
```

or the tagged numeral

```
(fresh (n) (num ,n)
```

are both abstractions that avoid having to enumerate all of the infully many numerals.

```
(fresh (x)
  (symbolo x)
  (numbero x))
```

fails finitely, as does

```
(fresh (e x n)
  (== `(sym ,x) e)
  (== `(num ,n) e))
```

without grounding `x`, or `n`, or fully grounding `e`.

want to avoid generate and test at all costs.

Naive representation of λ terms using logic variables for α-equivalence

explore issues around the apparently attractive approach of using logic variables and first-order syntactic unification to represent lexical scoping, binding, and α-equivalence with tagged lists representing λ terms:

```
(fresh (a b c d)
  (== `(lambda (,a) (lambda (,b) ,a))
      `(lambda (,c) (lambda (,d) ,d))))
```

or:

```
(fresh (a b c)
  (== `(lambda (,a) (lambda (,a) ,a))
      `(lambda (,b) (lambda (,c) ,b))))
```

and the resulting motivation for nominal unification, or other approaches;

You sometimes see these sorts of encodings in Prolog, although I've never seen the encoding use in a way that was actually fully relational. You always seem to use need to use a `copy_term`.

Doesn't work in terms of lexical scope, binding, shadowing, et cetera.

Either avoid this type of encoding entirely, and use symbols to represent variables, and either use an explicit environment or use De Bruijn, or use combinatory logic and avoid variables entirely, or try to implement capture-avoiding-substitution (CAS) relationally, or go higher order and do higher order abstract syntax (HOAS). This stuff is subtle, and an ongoing area of research in programming languages in general.

De Bruijn and implementing-capture-avoiding substitution relationally are both trickier to implement purely relationally than it might appear at first glance, at least if you're using first-order syntactic unification, which is the vanilla unification algorithm for miniKanren.

miniKanren's default unification has no build-in reasoning with respect to scope or binding. `lambda` here is just an arbitrary symbol, and `a`, `b`, and `c` are just standard unification variables.

It's an attractive notion to do application in λ-calculus using unification of `a`, `b`, or `c` to represent the result of a procedure application. For example, reducing the term:

```
`(app (lambda (,a) (lambda (,b) ,a)) (lambda (,c) ,c))
```

results in the unification:

```
(== `(lambda (,c) ,c) a)
```

The steps in schematic form:

```
`(app ,e1 ,e2)
(== `(lambda (,a) (lambda (,b) ,a)) e1)
(== `(lambda (,c) ,c) e2)
(== a e2)
```

`a` becomes:

```
`(lambda ((lambda (,c) ,c)) (lambda (,b) (lambda (,c) ,c)))
```

If you just look at the body this term, it can appear okay. However, the unification ruins `a` and ruins the original term corresponding to `e1`, which means `copy_term` is needed to copy the "template" of the original term *before* unification, which isn't relational (because reordering the conjuncts changes the semantics of the program). (Note also that the constraint `(symbolo a)` is no good, because `a` is going to be unified with some representation of a λ-calculus term, which is probably a tagged list.)

This motivates why you might want something like nominal logic programming or to do something more like λProlog and use a higher order abstract syntax encoding. But those bring along their own complications.

Alternative environment representations

explore alternative environment representations, including a "split" two-list representation, and another representation using association list constraints, working up to a relational interpreter with `lookupo` and `not-in-envo` relations that succeed finitely many times for environments that are not length instantiated

Check that I included notions of `free?` and `bound?`, and showed how to implement using clpset-faster-miniKanren/CLP(Set), like in the beginning of C311 and TAPL.

`not-in-envo` is especially pesky since it is used so often. Show why recursive `lookupo` and `not-in-envo` can cause divergence through succeeding an unbounded number of times. Show various issues with association list representation of environments, for `lookupo` and `not-in-envo` with environment that aren't ground, including duplicate answers for length-instantiated environments with fresh logic variables as keys, and for non-length instantiated environments.

2-list/split-list representation (thank Darius Bacon for email) and tradeoffs (ability to use `absento` on the "keys" list to do lazy `not-in-envo`, making it non-recursive. Show issues that remain: `lookupo` can still succeed an unbounded number of times when the environment isn't length instantiated.

Show association list constraints in clpset-faster-miniKanren (and from the Beyond Cons paper) and its trade-offs, including duplicate answers. (Is that an issue?) Include and acknowledge Michael Ballantyne's observation about the a-list constraints not being as abstract as desirable, since ordering in the a-list still matters: `((x . 5) (y . 6))` differs from `((y . 6) (x . 5))`, even though they represent the same environment. Future work!

Handling mutation using store-passing style

write relational interpreters that handle mutation using store-passing style

Issues with store passing/monadic style, forcing a “natural ordering” for recursive calls. Can reordering of conjuncts based on the ground-ness of terms help here when “running backwards”? Maybe, in this sort of case, reordering of recursive calls is especially important.

What about delayed goals? What about Petr Lozov’s work?

Difficulty with failing fast.

Also, variable-to-value lookup now involves a second lookup. Perhaps the alternative environment representation really helps here, as well, with the store representation. combined environment/store experiments.

using alist constraints?

Maybe we need both environment and store constraints to get decent behavior.

Could recent work on constraint handling in miniKanren help here?

Lots of open research questions and opportunities.

Also, Nada Amin pointed out to me that you only need a store when you have `set-car!` or whatever. You don’t need to use a store to only support `set!` of variables. So, we can just use an environment, and shadowing to update the environment, but still need to thread the environment through monadically, I think. So it only helps avoid the 2nd lookup—you still have to thread the environment through monadically, I believe.

Can also implement an interpreter for a little imperative language like IMP, which I’ve done before. Look at Polyconf 2015, for example. I’ve also implemented Matt Might’s article on static analysis as a relational interpreter, which involves mutation, if I recall correctly.

Explicit handling of errors

write interpreters with explicit handling of errors, showing how angelic execution can be used to synthesize expressions that, when evaluated, generate a given error

Angelic execution example from Polyconf 2015—critique and improve upon/riff upon that interpreter.

Tradeoffs and representing errors explicitly versus as failure (and similarly for predicates in Scheme translated into miniKanren, whether or not false is represented as failure and truth as success, versus having an explicit Boolean as a second argument). Divergence and error, shortcuts in the relational interpreter semantics to fail (to produce an answer) versus possible divergence in Scheme. (I think I have another chapter for discussing these encoding trade-offs.)

Synthesizing inputs that result in the given error.

Under which circumstances can we show that no such input exists? That task is harder in miniKanren than in showing such an input does exist.

Eigen, `absento`, abstracted value techniques.

Can we push this in metaKanren, or other miniKanren-in-miniKanren implementations, synthesizing code that produces or doesn't produce errors? (Maybe can already do that. Seems like we should be able to.)

Adding enough to handle `append`

add `letrec` and recursive environments, and add a simple pattern matcher to the relational Scheme interpreter, taking special care to explicitly handle the cases of not matching (which is the trickier part!), working up to a relational Scheme interpreter that can interpret `append`, can synthesize arguments to `append` using idioms like `(append '(a b c) ',x)`, and can synthesize small code expressions inside the definition of `append`

Reynolds half-closure representation of recursive environments (may be best to show first in Scheme)

(Maybe I should always include both the regular Scheme interpreter along with miniKanren relational version. It's more honest, easier to understand, often can be used to check or debug the relational version. and can be used to show concepts like divergence or error-versus-failure and other encoding choices. Yes, I like this idea. I'm not afraid of making the book longer.)

(Make sure conjunction of recursive calls for `cons` and all that has been treated beforehand.)

(At some point need to show `define`, but maybe not now.)

(Pattern matcher is tricky—probably worth its own chapter.)

Multiple argument application and λ needs to be here, if not before.

Primitives as first class, and non-empty initial environment, either here or around here.

`',x` versus `,x` behavior with `run*`, `run 1`, `run 2`, showing the expressions of relational Scheme interpreter, compared with `appendo` in miniKanren directly.

Limits to synthesis due to divergence or inefficiency, especially due to issues with conjunction of recursive calls and procedure application.

Point to 2012 Scheme Workshop paper and ICFP 2017 pearl.

Comparison between `append` in Scheme, `appendo` in miniKanren, and `append` in the relational Scheme interpreter running inside miniKanren

discuss the similarities and differences between `append` in Scheme, `appendo` in miniKanren, and `append` in the relational Scheme interpreter running inside miniKanren

Where you can put logic variables, number of arguments, efficiency, `',x` idiom versus `,x` in `append` in Scheme-in-miniKanrin.

Create a figure/diagram showing the different layers of interpretation.

Importance of ordering of conjuncts and other decisions in both `appendo` and in the relational Scheme interpreter, including for divergence behavior

list constraints, segment variables, segment unification, variadic, Scheme `append` supporting variable number of arguments. Last argument to`append` not needing to be a list or even a pair.

Synthesis of a term showing difference between lexical scope and dynamic scope

explore a neat example from the 2017 ICFP pearl: synthesizing an expression that evaluates to different values under lexical scope versus dynamic scope through sharing a fresh variable in the expression position for two relational interpreters, and using a disequality constraint to enforce the the value arguments differ

Code for both interpreters: is there a nice way to abstract over the difference?

```
(run 1 (expr val1 val2)
  (=/= val1 val2)
  (eval-lexo expr val1)
  (eval-dyno expr val2))
```

something like that.

Any sort of `absento` trick that can be used to generate more interesting answers, like for twines?

Interesting idea/technique of sharing the same expression between interpreters for the same language, but with different semantics. Seems like we could push this further.

Similar example with call-by-value versus call-by-name or call-by-need. Synthesize expressions showing difference in behavior.

Scheme “anti”-interpreter

write a relational Scheme “anti”-interpreter, `not-evalo`, where `(not-evalo expr val)` succeeds when Scheme expression `expr` does *not* evaluate to value `val`

Similar in spirit to the “doesn’t match” cases of the pattern matcher in full-interp.

Need to make decisions about *how* the given expression doesn’t evaluate to the given value: unbound variable, syntax error, arity error, semantic error, divergence, etc.

Can do something equivalent with an anti-type inferencer version as well.

Might assume expression parses and type checks, but doesn’t evaluate to the given value.

How is not-evalo different from evalo along with a use of a disequality constraint?

I don’t understand this very well.

Ongoing related experiments

Handling call-by-name and call-by-need

write relational Scheme interpreters that implement call-by-name and call-by-need, rather than call-by-value

How do notions of call-by-name and call-by-need interact with trying to avoid generate-and-test/premature grounding? Are these call-by's useful in this context?

How to implement call-by-need? In a functional way? In a relational way? What is the connection, relationally, to memoization, Scheme's `force` and `delay`, tabling, etc.? How should this be implemented in minikanren? What interesting synthesis examples are possible?

Handling Scheme memoized functions

write a relational Scheme interpreter that supports memoized functions;

How does this interact with tabling?

Tabling built into miniKanren? Tabling implemented outside the miniKanren implementation?

connections with abstract interpretation?

I'm not sure how to do this. Does memoing in Scheme require mutation, or can it be done with a store, or whatever? A purely functional implementation in Scheme would be helpful to start with.

Is this an issue of memoization and relationality?

What about Scheme-in-Scheme-in-miniKanren? Or scheme-in-miniKanren-in-miniKanren? Any additional leverage we can get?

Writing a relational abstract interpreter

write a relational abstract interpreter

various attempts with collaborators over the years are worth revisiting

some of the current work on constraint extensions to miniKanren might be helpful

miniKanren-in-miniKanren with aggregate operations

is tabling actually useful here?

Seems a bit like shallow-versus-deep versions of miniKanren-in-miniKanren, with `run 1` versus `run*`, or `run n` semantics

For some of these topics, I think just clearly expressing the problems, apparent issues, et cetera, would already be very useful. (Problem finding versus problem solving.)

Still, would be fun to write attempts at relational abstract interpreters, and explore the issues and what seems to be lacking.

Delayed goals

add delayed goals to faster-miniKanren, and explore uses of—and tradeoffs of—delayed goals in relational interpreters

Speeding up the interpreter

explore techniques for radically speeding up the interpreter in certain cases, such as dynamic reordering of conjunctions of recursive calls based on groundness of terms, and using a Scheme function rather than a miniKanren relation for variable lookup in the interpreter's environment when the arguments are sufficiently ground, working up to a relational Scheme interpreter that can synthesize `append` from expression/value pairs

Synthesizing `append` using associativity

explore synthesizing `append` and other recursive definitions using properties such as *associativity*, rather than only input/output pairs

Exploring sophisticated behavior under relational interpretation: proof checker and quasiquote Quine generation

explore sophisticated example programs from the 2017 ICFP pearl: proof checker becomes theorem prover via relational interpretation, and quasiquote Quine generation using a micro-Scheme in mini-Scheme in miniKanren in Scheme

Build Your Own Barliman

build a simple clone of Barliman, including Barliman's technique of trying to parse an expression relationally in the hope of failing fast, and throwing away any information gained upon success to avoid premature grounding before calling the evaluator, and also running each input/output pair on its own, in parallel with the entire synthesis problem (once again, hoping to fail fast)

Wanted: a relational hygienic macro expander

(as an aside from implementing relational interpeters) discuss lack of a relational hygienic macro expander, its implications, possible work arounds, and open problems

Relational compiler, as opposed to interpreter

(Galois papers?)

(as an aside from implementing relational interpeters) discuss relational compiler, rather than interpreter, for Scheme, along with tradeoffs, opportunities, and problems

Staging interpreters using staged-miniKanren

use the staged-miniKanren system to perform staged evaluation of relational interpreters, giving cool examples from the PLDI paper and repo

Free and bound variables using setKanren/CLP(Set)

consider the apparently simple notions of free and bound variables from a relational context, implementing C311-style functions relationally using setKanren/CLP(Set)

Adding arithmetic to a relational Scheme interpreter

consider the apparently simple task of adding arithmetic to a relational Scheme interpreter, and explore the various ways this may be done, including using variable projection, delayed goals, tagged Peano numerals, “Kiselyov” numerals, finite domain constraints, and CLP(SMT)

A small-step λ-calculus reducer and capture-avoiding substitution

write a small-step λ-calculus reducer, based on capture-avoiding substitution, exploring issues related to names and binding

A small-step λ-calculus reducer, using De Bruijn notation

write a small-step λ-calculus reducer, using De Bruijn notation, and explore issues and tradeoffs

A small-step term reducer for the SKI combinator calculus

write a small-step term reducer for the SKI combinator calculus, and explore the tradeoffs in terms of avoiding nominal issues with capture-avoiding substitution and α-equivalence, and the difficulties related to failing fast, showing Ω as a small-step Quine, working up to synthesizing combinator terms and fixpoint combinators (Question: can this reducer be staged by staged-miniKanren?)

Normalization-by-evaluation, relational style

explore the technique of normalization-by-evaluation (or *NBE*) from a relational standpoint, resulting in two relations, `evalo` and `unevalo`, that together can synthesize a fixpoint function in Scheme (and explore technique for ensuring freshness of lexical variables using `absento` and a list of variables encountered so far (which do not have to be ground))

microKanren in the Barliman-style relational Scheme interpreter

run a slightly massaged version of the Scheme code for microKanren in the Barliman-style relational Scheme interpreter, then explore the behavior of microKanren programs in this relational context, including the behavior of extralogical operators such as `varo`

Shallow, medium, and deep embeddings of miniKanren-in-miniKanren

implement shallow, medium, and deep embeddings of miniKanren-in-miniKanren, and run queries that share arguments between shallower and deeper embeddings to expore tradeoffs between performance versus expressive power;

Relational behavior of extra-logical operators in a deep embedding of miniKanren in miniKanren

explore the relational behavior of extra-logical operators in a deep embedding of miniKanren in miniKanren, including aggregate operators inspired by Prolog's `findall` and `setof` predicates, implemented using setKanren/CLP(Set)