How to Debug miniKanren Programs

William E. Byrd

An Unreasonable Schemer Production

Book 01 of Will's ShovelWare Book Pack
Version 1.0 Sunday, October 5, 2025

To past, current, and future miniKanren programmers.
May your programs never diverge.

DISCLAIMER

This book presents suggestions, advice, and techniques for writing and debugging miniKanren programs, based on the author's personal experience. Which is, admittedly, vast, given that I've probably written more buggy miniKanren programs than anyone else (although I don't know if I should be proud of that particular fact). Writing and debugging software is a complex skill, even without coming face-to-face with the Halting Problem all the time. Use at your own risk, no implied warranty, and all that. If you follow the advice in this book, and your miniKanren software does something horrible, then don't blame me. Blame miniKanren, perhaps. (Which, to be fair, I helped create. I guess you got me on that one.) Also, why are you using miniKanren for anything you are that worried about, anyway!?

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 publically 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[1] 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 publically 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 first of the 17 books, due on Sunday, October 5, 2025, Anywhere on Earth (or whenever I go to sleep!). (Originally I was going to release the books on Monday nights, but Sundays work better.) The next book, book 02, will be due Sunday, October 26, 2025.

[1]`https://www.youtube.com/watch?v=d6_cB-jtxYI`

Contents

Preface

This is a book about how to debug miniKanren programs, and how to write miniKanren programs to make debugging easy—or better yet—unnecessary.

This book focuses on writing and debugging *purely relational* miniKanren programs, although sometimes using non-relational operators or techniques to aid in the debugging (and to be removed once the program has been fixed, to avoid ruining the relational behavior of the program). By a pure relation, I mean a miniKanren relation in which fresh variables can appear anywhere within any argument to the relation. In a pure relation the conjuncts and disjuncts can be reordered arbitrarily without affecting the *logical meaning* of the program, although such reordering can affect the *operational behavior* of the program, including the efficiency—and even the termination behavior—of queries using the relation. By the *termination behavior* of a query, I mean whether a query *diverges* (that is, runs forever) or *fails finitely* (that it, completes in a finite amount of time).

At this point miniKanren is really a family of closely related languages, or perhaps dialects, with major implementations embedded in Scheme, Racket, OCaml, and Clojure, and with various constraint extensions. In this book we will use the Scheme version of the faster-miniKanren[2] implementation, running in Chez Scheme[3] version 10.2.0.

Pure relational operators in faster-miniKanren include: the core miniKanren operators `conde`, `fresh`, and `==`; the disequality constraint `=/=`; the "absence" constraint `absento`; and the type constraints `numbero`, `symbolo`, `stringo`. Non-relational operators include `conda`, `condu`, and `project`. `run` and `run*` act as an interface

[2]`https://github.com/michaelballantyne/faster-miniKanren`
[3]`https://cisco.github.io/ChezScheme/`

between miniKanren and its host language—in this case, Scheme—and are not purely relational.

Other purely relational operators in other versions of miniKanren include `condi` and `freshi` from the first edition of *The Reasoned Schemer*, `disj2` and `conj2` from the second edition of *The Reasoned Schemer*, and `disj` and `conj` of microKanren.

Errors, Omissions, Requests

If you would like to report mistakes, share additional debugging techniques, inquire about techniques I have not fully explained, or suggest a topic for another volume in *Will's ShovelWare Book Pack*, please email me at `webyrd@gmail.com`.

Typographical Conventions

Unlike the first and second editions of *The Reasoned Schemer*, and some papers on miniKanren, this book does not use fancy typesetting of Scheme and miniKanren code. Instead, all code appear in `typewriter` font. While the fancy typesetting looks nice, it has several disadvantages: many readers find it difficult to translate the pretty fonts, superscripts, subscripts, and implicit quasiquotes and unquotates into code to type into a text editor; the fancy typesetting makes it very difficult to translate a book or paper to other formats, and inhibits accessibility; fancy typesetting takes *a lot* of time and attention; and, worst of all (in my opinion) it often isn't clear which font to use when typesetting code that is being run in an interpreter, which is itself being run in an interpreter, which is the sort of thing we often do in miniKanren (such as in the last example of the 2017 ICFP paper, *A unified approach to solving seven programming problems (functional pearl)*[4]).

Acknowledgements

I greatly appreciate the encouragement of my parents, brother, and sister to write books, to finish books, release books into the world.

[4]`https://dl.acm.org/doi/10.1145/3110252`

As always, I am grateful to Dan Friedman for introducing me to relational programming, and to Oleg Kiselyov for demonstrating sophisticated programs that are completely relational. I also thank Dan, Oleg, and Jason Hemann—my co-authors on *The Reasoned Schemer, 2nd Edition*—for all they have taught me.

Conversations with Jeff Henrikson about the lack of resources for learning about the actual practice of miniKanren programming encouraged me to create an earlier, unpublished paper on miniKanren debugging. I thank Jeff for our discussions and his encouragement.

Nada Amin helpfully suggested I break down the draft of a very long paper on miniKanren programming into multiple more focused papers, starting with the unpublished paper on debugging miniKanren.

I thank Nada Amin, Bharath Joshi, and Michael Ballantyne for their comments on drafts of the unpublished miniKanren debugging paper, and for sharing their miniKanren debugging techniques with me.

I thank the anonymous reviewers from the 2022 miniKanren and Relational Programming Workshop, for their comments on the unpublished draft paper on miniKanren debugging.

While this book differs significantly from the unpublished paper on miniKanren debugging, the ideas and discussions around the draft have influenced my thinking about miniKanren debugging, which is reflected in this book.

I thank Matt Might and the rest of the Hugh Kaul Precision Medicine Institute at the University of Alabama at Birmingham for their support and encouragement over the years. Greg Rosenblatt, my close collaborator at the Hugh Kaul Precision Medicine Institute, has taught me much about debugging sophisticated miniKanren programs, especially through our work on Barliman.

I've learned a great deal about miniKanren programming—and debugging!—from hundreds of budding and expert miniKanren programmers over the past 20 years. I am especially grateful to my many students, mentees, collaborators, and co-authors. Thank you all!

Chapter 1

General miniKanren Debugging Techniques and Advice

If you're spending lots of time tracking down an issue, try rewriting the code from scratch without looking at your current code. Why? Because perhaps you made a silly mistake that you are having trouble seeing.

For example, do you see the problem with this code?

```
(define (implicit-begin-strikes-againo x y)
  (helper-relation-1 x y)
  (helper-relation-2 x y))
```

In Scheme, that code is equivalent to:

```
(define implicit-begin-strikes-againo
  (lambda (x y)
    (helper-relation-1 x y)
    (helper-relation-2 x y)))
```

which is equivalent to:

```
(define implicit-begin-strikes-againo
  (lambda (x y)
    (begin
      (helper-relation-1 x y)
      (helper-relation-2 x y))))
```

which is *logically* equivalent to the miniKanren code:

```
(define implicit-begin-strikes-againo
  (lambda (x y)
    (begin
      (helper-relation-2 x y))))
```

which is *logically* equivalent to the miniKanren code:

```
(define implicit-begin-strikes-againo
  (lambda (x y)
    (helper-relation-2 x y)))
```

Due to the implicit `lambda`—containing an implicit `begin`—the first goal expression, `(helper-relation-1 x y)`, is *evaluated for effect*, and might diverge (in Scheme, before it is run in miniKanren!), or signal an error. However, if the first goal expression does not diverge in Scheme or signal an error, the value of the goal expression—that is, the actual goal—will be discarded. The value of a Scheme `begin` form is the value of the last expression appearing in the `begin`—the values of earlier expressions are discarded once they are evaluated for effect.

What a subtle and annoying error, which can produce baffling behavior. This is so annoying that the second edition of The Reasoned Schemer introduces a `defrel` form for defining relations:

```
(defrel (this-is-more-like-ito x y)
  (helper-relation-1 x y)
  (helper-relation-2 x y))
```

Alternatively, you can protect yourself from the infamous "implicit `begin`" error by using a `fresh` form in the body of the `lambda`:

```
(define this-is-more-like-ito
  (lambda (x y)
    (fresh ()
       (helper-relation-1 x y)
       (helper-relation-2 x y))))
```

or

```
(define (implicit-begin-strikes-againo x y)
  (fresh ()
    (helper-relation-1 x y)
    (helper-relation-2 x y)))
```

(Or you can use a `conj` if you are in microKanren.)

Now, even experienced miniKanren hackers can run into trouble with implicit `begin`, sometimes when doing something tricky, or when using a miniKanren without `defrel`, or when writing miniKanren the old-fashioned way. Usually the miniKanen programmer starts out with a definition that works fine, such as:

```
(defrel (this-is-more-like-ito x y)
  (helper-relation-1 x y))
```

The problem, of course, occurs when the programmer adds a second goal expression, intending to express a logical conjunction:

```
(define (implicit-begin-strikes-againo x y)
  (helper-relation-1 x y)
  (helper-relation-2 x y))
```

Whoops!
For this reason, it can be a good idea to *program defensively* by wrapping a `fresh` around the single goal expression in the original relation definition, anticipating the addition of future goal expressions in a conjunction:

```
(defrel (this-is-more-like-ito x y)
  (fresh ()
    (helper-relation-1 x y)))
```

Now, this is longer and uglier than the code without the defensive fresh, and can change the search order, the order of answers returned from queries, and the performance of queries. So, experienced miniKanren programmers might skip the defensive `fresh`, or remove it from existing code. And run into the problem later.

The bigger point of this story is to show that in at least some cases it makes sense to rewrite code that isn't working as you expect, without looking at your old code, since it is unlikely that an experienced miniKanren programmer will leave out the necessary `fresh` or explicit conjunction when writing the code from scratch, but it is quite likely (as I know from personal experience) that an experienced programmer might not notice the missing `fresh` in code that accreted over time.

Also, stopping to rewrite your code is a perfect time to rethink your approach. Maybe you would benefit from writing down an explicit grammar for the data you are interpreting, or writing down explicit inference rules for the logic of your relation. Or maybe it's time for to go back and write a version of the code in Scheme (or whatever your host language is), get it working functionally, test the Scheme version using a bunch of examples, and all that. And then massage the Scheme code, translating it step-by-step toward a relation, testing the code after every step.

So if you're trying to debug something for a long time, you're probably better off just trying to start over again. If you really understand the problem you are trying to solve at that point, rewriting the code should be relatively quick.

Are you loading the right code? It's surprisingly easy to be running the wrong code running the wrong code from the wrong file.

If you're developing at the REPL, or if you're developing by loading a file once, modifying the file, and then reloading that file over and over again—that is, if your development workflow involves mutating the state of your Scheme system—and you run into any sort of unusual or strange behavior, a very good technique is to quit your Scheme implementation and reload the file fresh from a new interpreter session. Or open a new terminal window, or reopen Emacs, or your IDE, or whatever environment you are running your code from. Then try to load your code again, from a totally fresh environment, to make sure you didn't get into some weird state, which I've seen happen many times. It's extremely annoying when your code works in the REPL because you've got some strange variant of a definition loaded, and then later, when you try running the code in a fresh session, it doesn't work. It's good practice to check periodically that you can run everything from a fresh session.

Are your macro definitions and goal definitions in the right order? In Scheme, macro definitions generally need to come first, before the definitions of the expressions that use the macros. If you notice that your code works when you loading a file twice after starting (or restarting) Scheme, but you get an error the first time you load the file, it's probably a code ordering problem. In general, you want your macro definitions to come first in a file. Similarly, if you are loading multiple files, be careful of the order in which you load those files.

A useful technique in debugging a miniKanren relation is to compare the relation with a correct functional equivalent written in plain Scheme. Ideally the Scheme function will be as close in behavior to the miniKanren relation as possible. One way to derive such a Scheme function is to start with a vanilla, fully tested Scheme definition in terms of `if` or `cond`, for example, and replace the conditional with pattern matching. Then, try to make the patterns matched against as close as possible to the equivalent desired term representations in the miniKanren relation, using the same list or pair structure, symbolic tags (for example, `clos` to represent a tagged closure), and type predicates such as `number?` or `symbol?` to match the use of dynamic type constraints in miniKanren, such as `numbero` and `symbolo`. With unambigous, non-overlapping patterns in place, try reordering the pattern matching clauses, checking that the Scheme function continues passes all tests. At this point you should be able to check that each `conde` clause in the miniKanren relation is equivalent to the corresponding pattern matching clause of the Scheme function (taking into account the inherent differences between miniKanren and Scheme behavior, of course).

Fully ground arguments are very useful for testing programs that diverge (or appear to diverge), or which return unexpected answers. Assuming you've written your relation to follow the non-overlapping principle, the relation should not produce duplicate answers, or answers that are subsumed by other answers.[1] In particular, passing fully ground arguments to a relation should result in *at most* one answer; if more than one answer is returned, this almost certainly indicates a problem with the logic or implementation or your relation, such as a missing type constraint, or the lack of a non-ambiguous tagging scheme.

Partially-ground terms containing fresh variables, or fully fresh arguments, are very helpful for testing programs that fail unexpectedly, when you expect to get an answer back. Using a `run 1` query, pass distinct fresh logic variables as the arguments to the relation. Avoid using the same logic variable in two argument positions, since

[1]Caveat: a few constraint extensions to miniKanren, such as CLP(Set) extensions, may result in duplicate or subsumed answers.

such sharing can cause failure or divergence.

You can and should independently test all helper relations that are called—directly or indirectly—from the relation you are debugging. Even if I'm not running into problems, I always test relations as I'm write them, including even very simple helper relations. If a helper relation contains any error, such as incorrect tagging, or using a `quote` instead of a `quasiquote`, or using an `unquote` in the wrong position, that can wreak havoc upon the rest of the program.

One great way to test any relation, including a helper relation, is to just feed in all fresh logic variables as arguments, using a `run 1` query, and then increase the value of the numeric argument to `run` to inspect additional answers as desired. If the relation you are testing is finite, you should be able to use a `run*`, which should terminate. This is not true in general, of course, for relations that are recursive or mutually recursive, or that call recursive relations.

Fully ground arguments are very good for debugging performance problems. If you're trying to run some query that seems to be taking eons to come back, you could try grounding the arguments as much as possible, if you know the answers. And if that terminates, then start relaxing the arguments by putting fresh logic variables within those terms. As you relax the groundness of the terms, watch how the runtime or memory usage of your queries change. I often use this technique when tracking down performance problems in synthesizing Scheme code from a relational interpreter, or from a relational type inferencer, but the technique is useful in general.

Chapter 2

Questions to Ask Yourself When Debugging

- Are you loading the right code, in the right files, in the right order?

- Are the macros and definitions in the proper order? In Scheme, macro definitions should generally come first in a file.

  If working from the REPL, or working by re-loading files as you change their contents, does a clean load from a fresh environment work the same? You may have to restart your IDE, Emacs, terminal, or whatever, in order to test this properly.

- Are you actually running the miniKanren goals you think you are running? Duplicate definitions, change definitions in the REPL, or incorrect syntax (such as incorrect parentheses in a `conde`, or an unintentional use of Scheme's implicit `begin` form inside a `lambda` or `let`, might mean that you're not running the goals you think you are, which can lead to extremely bizarre behavior.

- Have you double-checked (and triple-checked) your use of tags, if you are using explicitly tagged representations of expressions or values?

- Similarly, have you double-checked every use of `quote`, `quasiquote`, and `unquote`, a very common source of errors that can be hard to track down?

- Have you checked the grammatical correctness of your miniKanren code? Did you accidentally use a miniKanren goal (or goal constructor) as an argument to `==`, `=/=`, or `absento`, such as `(== (conde ((== y 5)) ((== z 6))) x)`?

- If you are writing any sort of relation that as an an interpreter over data—which includes most miniKnren relations, such as parsers, type inferencers, evaluators, theoerm provers, and arithmetic systems, to just name a few—have you checked the grammatical correctness of your code with respect to your term representations, including list or pair structure, tags, and type constraints? Are you sure that your tests examples using the proper grammar for the expressions being interpreted, or values produced?

- Are you accidentally trying to do higher-order logic programming in miniKanren by passing a relation as as an argument to another relation, or by unifying two relations, or by unifying two `lambda` expressions in Scheme, or by mapping a relation as if it were a regular Scheme procedure? While Scheme supports higher-order functional programming, miniKanren is based on first-order syntactic unification (with some extensions in some versions of miniKanren); in general, miniKanren does not support higher-order programming, or functional-logic programming, without advanced hackery that combines Scheme and miniKanren in subtle ways.

- Are you trying to use a Scheme function that deconstructs or inspects its arguments, such as `car`, `cdr`, `map`, `number?`, `null?`, or `append`, with one or more term that might contain logic variables? Even if the logic variables happen to have become ground through unification, unless you use `project`, the host language will misinterpret those logic variable. For example, in Scheme implementations of miniKanren, logic variables are often represented as vectors. Therefore, calling `car` on a logic variable that has become ground to a pair will produce an error in Scheme, since Scheme will try taking the `car` of a vector, which is illegal, instead of looking up the value of that vector in the substitution. Using `cons` or `list`, on the other hand, is safe, since these Scheme procedures are constructors that combine data, without inspecting or tearing apart the data on which they are operating.

- If your relation isn't coming back, have you tried to using a `run 1` instead or a `run*`, or instead of a `run` with a number larger than 1?
- Have you tried calling the relation with all ground arguments?
- Have you tried calling the relation with all fresh arguments?
- Have you tried reordering goals in a conjunction, ensuring that recurive calls and calls to recursive helper relations come after simple, non-recursive goals and helper relations?
- Have you compared each relation, line-by-line, with inference rules, grammars, or the Scheme or Racket functional version?
- Have you tried tracing the code, perhaps at the Scheme level, but probably at the miniKanren level?
- Have you tried commenting out `conde` clauses, conjuncts, or disjuncts? Always test the smallest program you can and build from there.
- If you are *really* stuck, have you tried re-rewriting the code from scratch, *without* looking at your current code (so you don't copy any silly mistakes that you aren't seeing)? Keep the original, non-working code, so you can diff it against your re-written code.