Relational Interpreters in miniKanren

(WORKING ROUGH DRAFT — DRAFT 0)

William E. Byrd

January 19, 2025

To Dan Friedman

Contents

Preface

The intent of this book is to share the techniques, knowledge, pitfalls, open problems, promising-looking future work/techniques, and literature of writing interpreters as relations in miniKanren. Someone who reads this book actively should be ready to understand, implement, modify, and improve interpreters wtitten as miniKanren relations, read the related literature, and perform original research on the topic.

0.1 What this book is about

This book is about writing interpreters for programming languages, especially for subsets of Scheme. While there are many books on writing interpreters, this book is unusual in that it explores how to write interpreters as *relations* in the miniKanren relational programming language. By writing interpreters as relations, and by using the implicit constraint solving and search in the `faster-miniKanren` implementation, we can use the flexibility of relational programming to allow us to experiment with programs in the language being interpreted. For example, a relational interpreter can interpret a program with missing subexpressions[1], or *holes*, attempting to fill in the missing subexpressions with values that result in valid programs in the language being interpreted. Or we can give both a program containing holes and the value we expect the program to produce when interpreted, and let `faster-miniKanren` try to fill in the holes in a way to produce the expected output. We can even write an interpreter that explicitly handles errors, and ask `faster-miniKanren` to find inputs to the program that trigger these errors.[2]

0.2 What you need to know to read this book

This book assumes you are familiar with the basics of Scheme or Racket, and are comfortable with the ideas of functionals programming. The book also assumes you understand the notions of evaluation order of expressions, lexical scope, environments, environment-passing interpreters.

[1] Such programs are often called *program sketches* [TODO cite].

[2] This is known in the literature as "angelic execution".

[TODO add topics that the reader should know, including Scheme, miniKanren, lexical scope, environment-passing interpreters, etc]

[TODO add pointers to resources]

0.3 Running the code in this book

The code in this book was tested with Chez Scheme and Racket. It should be possible to run most code in other Scheme implementations, with few or no changes, with the exception of code that makes extensive use of Chez-specific or Racket-specific features, which I will point out in those chapters, as appropriate.

0.3.1 Getting `pmatch` from GitHub

0.3.2 Getting `faster-miniKanren` from GitHub

https://github.com/michaelballantyne/faster-miniKanren

```
git clone git@github.com:michaelballantyne/faster-minikanren.git
```

Alternatively, you can click on `<> Code` button and select `Download ZIP` to download and uncompress the `.zip` file containing the entire `faster-miniKanren` directory.

0.3.3 Using this book with Chez Scheme

Installing Chez Scheme

Starting a Chez Scheme REPL

Loading a file in Chez Scheme

Loading `faster-miniKanren` in Chez Scheme

0.3.4 Using this book with Racket

Installing Racket

https://racket-lang.org/

https://download.racket-lang.org/

Important differences between Chez Scheme and Racket

representation of quoted values

evaluation order

language levels

macros

The DrRacket IDE and the Racket REPL

Starting and configuring DrRacket

changing default language
changing default memory limit

Starting a Racket REPL

Requiring a module in Racket

Requiring the `faster-miniKanren` module in Racket

0.4 Acknowledgements

Dan Friedman and Michael Ballantyne both encouraged me to continue working on this book, and independently encouraged me to break down one giant book into more than one book, each book being more manageable. Both Dan and Michael encouraged me to avoid getting bogged down with a lot of introductory material, which had caused me to abandon previous writing efforts. Michael also encouraged me to continue working on the book in the open.

Darius Bacon wrote me a very helpful email about how using two separate lists to represent a lexical environment, rather than a single association list, can result in better performance and divergence behavior. I had played around with this representation in the past, but had abandoned it before I understood its advantages. Thank you, Darius.

My mother has continually encouraged me to work on this book, and most importantly, to finish it!

[TODO add other acknowledgements]

[TODO add acks for typesetting tech, such as the fonts; also can add colophon if I'm so inclined]

Chapter 1

A simple environment-passing Scheme interpreter in Scheme

call-by-value (CBV) λ-calculus (variable reference, single-argument *lambda*, and procedure application), plus `quote` and `list`

association-list representation of the environment

empty initial environment

`list` is implemented as if it were a special form rather than as a variable bound, in a non-empty initial environment, to a procedure. As a result, although `list` can be shadowed, `(list list)` results in an error that there is an attempt to reference an unbound variable `list`.

tagged list to represent closure

grammar for the language we are interpreting

```
(load "pmatch.scm")
```

[TODO make sure I explain MIT vs Indiana syntax for `define`]

```
(define (eval expr)
  (eval-expr expr '()))

(define (eval-expr expr env)
  (pmatch expr
    ((quote ,v)
     (guard (not-in-env? 'quote env))
     v)
    ((list . ,e*)
```

```
     (guard (not-in-env? 'list env))
     (eval-list e* env))
    (,x
     (guard (symbolo? x))
     (lookup x env))
    ((,rator ,rand)
     (let ((a (eval-expr rand env)))
       (pmatch (eval-expr rator env)
         ((closure ,x ,body ,env^)
          (guard (symbol? x))
          (eval-expr body `((,x . ,a) . ,env^))))))
    ((lambda (,x) ,body)
     (guard (and (symbol? x)
                 (not-in-env? 'lambda env)))
     `(closure ,x ,body ,env))))

(define (not-in-env? x env)
  (pmatch env
    (((,y . ,v) . ,env^)
     (if (equal? y x) ;; TODO eq? vs eqv? vs equal?, with equal? being semantically cl
         #f
         (not-in-env? x env^)))
    (() #t))) ;; TODO empty env clause comes second; Dijkstra guard, and all that

(define (eval-list expr env)
  (pmatch expr
    (() '())
    ((,a . ,d)
     (let ((t-a (eval-expr a env))
           (t-d (eval-list d env)))
       `(,t-a . ,t-d)))))

(define (lookup x env)
  (pmatch env
    (() (error 'lookup "unbound variable")) ;; TODO make sure error is introduced, and
    (((,y . ,v) . ,env^)
     (if (equal? y x)
         v
         (lookup x env^)))))
```

Chapter 2

Rewriting the simple environment-passing Scheme interpreter in miniKanren

In this chapter we will translate the evaluator for the simple environment-passing interpreter from the previous chapter from a Scheme function to a miniKanren relation.

[TODO cite the code from the Quines interp in faster-miniKanren, and point to the 2012 SW paper on Quines]

[TODO this interp uses defrel—do I want to stick with defrel, or use define + lambda? Or maybe the book shows both (probably needs to show both at some point)]

```
(load "mk-vicare.scm")
(load "mk.scm")

(defrel (evalo expr val)
  (eval-expro expr '() val))

(defrel (eval-expro expr env val)
  (conde
    ((fresh (v)
       (== `(quote ,v) expr)
       (not-in-envo 'quote env)
       (absento 'closure v) ;; TODO discuss the tradeoffs of moving this absento to evalo
       (== v val)))
    ((fresh (e*)
       (== `(list . ,e*) expr)
```

```
       (not-in-envo 'list env)
       (absento 'closure e*) ;; TODO is this absento really needed, if we have absento
       (eval-listo e* env val)))
    ((symbolo expr) (lookupo expr env val))
    ((fresh (rator rand x body env^ a)
       (== `(,rator ,rand) expr)
       (eval-expro rator env `(closure ,x ,body ,env^))
       (eval-expro rand env a)
       (eval-expro body `((,x . ,a) . ,env^) val)))
    ((fresh (x body)
       (== `(lambda (,x) ,body) expr)
       (symbolo x)
       (not-in-envo 'lambda env)
       (== `(closure ,x ,body ,env) val)))))

(defrel (not-in-envo x env)
  (conde
    ((fresh (y v env^)
       (== `((,y . ,v) . ,env^) env)
       (=/= y x)
       (not-in-envo x env^)))
    ((== '() env))))

(defrel (eval-listo expr env val)
  (conde
    ((== '() expr)
     (== '() val))
    ((fresh (a d t-a t-d)
       (== `(,a . ,d) expr)
       (== `(,t-a . ,t-d) val)
       (eval-expro a env t-a)
       (eval-listo d env t-d)))))

(defrel (lookupo x env t)
  (fresh (y v env^)
    (== `((,y . ,v) . ,env^) env)
    (conde
      ((== y x) (== v t))
      ((=/= y x) (lookupo x env^ t)))))
```

Chapter 3

Quine time

McCarthy challenge given in 'A Micromanual for LISP'

```
(run 1 (e) (evalo e e))

=>

((((lambda (_.0) (list _.0 (list 'quote _.0)))
   '(lambda (_.0) (list _.0 (list 'quote _.0))))
  (=/= ((_.0 closure)) ((_.0 list)) ((_.0 quote)))
  (sym _.0)))

> ((lambda (_.0) (list _.0 (list 'quote _.0)))
    '(lambda (_.0) (list _.0 (list 'quote _.0))))
((lambda (_.0) (list _.0 (list 'quote _.0)))
  '(lambda (_.0) (list _.0 (list 'quote _.0))))
```

We replace `_.0` with the arbitrary free variable name `x` to produce the canonical LISP/Scheme Quine:

```
((lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))))

> ((lambda (x) (list x (list 'quote x)))
   '(lambda (x) (list x (list 'quote x))))
((lambda (x) (list x (list 'quote x)))
  '(lambda (x) (list x (list 'quote x))))
```

Twines

every Quine is trivially a Twine; we can add a disequality constraint to ensure `p` and `q` are distinct terms

```
> (run 1 (p q)
    (=/= p q)
```

```
    (evalo p q)
    (evalo q p))
[TODO add the answer]
```

[1]

Thrines

```
> (run 1 (p q r)
    (=/= p q)
    (=/= p r)
    (=/= q r)
    (evalo p q)
    (evalo q r)
    (evalo r p))
[TODO add the answer]
```

Structurally boring Quines, Twines, and Thrines
just moving quotes around
`absento` trick to generate more interesting Quines, Twines, and Thrines

```
> (run 1 (p q)
    (absento p q)
    (absento q p)
    (evalo p q)
    (evalo q p))
[TODO add the answer]
```

[similarly for Thrines]
Revisiting our original Quine query with the `absento` trick

```
(run 1 (p)
  (fresh (expr1 expr2)
    (absento expr1 expr2)
    (== `(,expr1 . ,expr2) p)
    (evalo p p)))

=>

((((lambda (_.0)
     (list (list 'lambda '(_.0) _.0) (list 'quote _.0)))
   '(list (list 'lambda '(_.0) _.0) (list 'quote _.0)))
  (=/= ((_.0 closure)) ((_.0 list)) ((_.0 quote)))
  (sym _.0)))
```

[1] I thank Larry Moss and the Indiana University Logic Symposium [TODO check the name of the symposium] for inviting me to give a talk where I demonstrated Quine generation, and where Larry suggested I tried generating Twines.

Gary P. Thompson II's Quine page (http://www.nyx.net/~gthompso/quine.htm) also describes *Quine Generating Programs* (http://www.nyx.net/~gthompso/self_gener.txt):

> It is possible (and actually in some cases easier) to write a program which outputs another program which is itself a quine.

using `q.scm` from faster-miniKanren:

```
(run 1 (nq q)
  (=/= q nq)
  (evalo nq q)
  (evalo q q))
=>
((('((lambda (_.0) (list _.0 (list 'quote _.0)))
     '(lambda (_.0) (list _.0 (list 'quote _.0))))
   ((lambda (_.0) (list _.0 (list 'quote _.0)))
    '(lambda (_.0) (list _.0 (list 'quote _.0)))))
  (=/= ((_.0 closure)) ((_.0 list)) ((_.0 quote)))
  (sym _.0)))
```

```
(run 1 (nq q)
  (absento q nq)
  (evalo nq q)
  (evalo q q))
=>
((((list
     '(lambda (_.0) (list _.0 (list 'quote _.0)))
     ''(lambda (_.0) (list _.0 (list 'quote _.0))))
   ((lambda (_.0) (list _.0 (list 'quote _.0)))
    '(lambda (_.0) (list _.0 (list 'quote _.0)))))
  (=/= ((_.0 closure)) ((_.0 list)) ((_.0 quote)))
  (sym _.0)))
```

[TODO: challenge to myself: generate a Kimian self-rep (http://www.nyx.net/~gthompso/self_kim.txt)

> Kimian self-rep, like quines, got it's name from Godel Escher Bach. A Kimian 'program' is actually the error produced by the system when it encounters the code. Kimian self reps are therefore very system-specific, and even implementation specific.

I want to add errors to the interp, any way. Once I've added errors, I should in theory be able to synthesize Kimian self-reps. Try it! And then move this description and code to the chapter that adds errors to the interp.]

[TODO: play with the ideas in this other Quines page http://www.madore.org/~david/computers/quine.html (thanks to Nada Amin for the pointer to this page, which I had forgotten about)]

Chapter 4

Using a two-list representation of the environment

association-list representation of an environment where `x` is mapped to the list `(cat dog)` and `y` is mapped to `5`:

```
((x . (cat dog))
 (y . 5))
```

"split" two-list representation of the same environment:

```
(x y) ; variables

((cat dog) 6) ; values
```

;; a-list env ;; ((x . (cat dog)) ;; (y . 5))
;; split env ;; (x y) ;; ((cat dog) 6)
`absento` trick for lazy `not-in-envo`
;; (absento 'closure expr)
;; (absento t1 t2)
;; (not-in-envo 'lambda env) ;; (absento 'lambda '(x y))

```
(defrel (evalo expr val)
  (eval-expro expr '(() . ()) val))

(defrel (eval-expro expr env val)
  (conde
    ;; quote, list, and variable reference/lookup clauses elided
    ((fresh (rator rand body env^ a x x* v*)
       (== `(,rator ,rand) expr)
       (== `(,x* . ,v*) env^)
```

```
       (eval-expro rator env `(closure ,x ,body ,envˆ))
       (eval-expro rand env a)
       (eval-expro body
                   `((,x . x*) . (,a . v*))
                   val)))
    ;; lambda clause elided
    ))

(defrel (not-in-envo x env)
  (fresh (x* v*)
    (== `(,x* . ,v*) env)
    (absento x x*)))
```

[TODO discuss tradeoffs between asserting (symbolo x) in these helper relations—how stand-alone do we want them?]

```
(defrel (lookupo x env t)
  (fresh (y x* v v*)
    (== `((,y . ,x*) . (,v . ,v*)) env)
    (conde
      ((== y x) (== v t))
      ((=/= y x) (lookupo x `(,x* . ,v*) t)))))
```

Chapter 5

Extending the interpreter to handle `append`

add `cons`, `car`, `cdr`, `null?`, and `if`

extend `lambda` and application to handle multiple arguments and variadic

Chapter 6

Using a non-empty initial environment

new case to handle prim app rather than user-defined closure app

`cons`, `car`, `cdr`, and `null?` bound in the initial env to prims

`list` bound in the initial env to the closure that results from evaluating the variadic `(lambda x x)`

Chapter 7

Adding explicit errors

So far our interpreters handle Scheme errors implicitly by failing to produce a result, rather than producing an explicit error. This implicit representation of errors has advantages, in both keeping the implementation code simple and in performance/fail-fast behavior.

[TODO see the code and comments in `mk-daily/2025_01_19` — this code needs to be updated to use non-empty initial environment, to avoid unbound variable errors for built-ins, such as `(error (unbound variable lambda))`]

[TODO discuss disadvantage of threading through values monadically instead of lexically, esp. regarding losing fail-fast behavior, breaking the wires, and losing flexibility in reordering conjuncts]

Chapter 8

Angelic execution

[TODO look at my code from PolyConf 2015, which includes an interpreter for an imperative language, along with angelic execution]

Chapter 9

Adding mutation

[TODO look at my code from PolyConf 2015, which includes an interpreter for an imperative language, along with angelic execution]

support `set!` (can we get away with supporting `set!` without adding a store?)

support mutiple pairs and have an explicit store

Chapter 10

Adding delimited control operators

delimited continuations and/or effect handlers—can we do so in such a way that avoids "breaking the wires"?

talk about the problem with `call/cc` and breaking the wires

Chapter 11

Writing a parser as a relation

Chapter 12

Writing a type inferencer as a relation

Chapter 13

Build your own Barliman

Chapter 14

Speeding up the interpreter

[restrict to interpreter changes that don't require hacking faster-miniKanren or in-depth knowledge of the implementation]

dynamic reordering of conjuncts, especially for application

fast environment lookup for environments that are sufficiently ground

Chapter 15

Open problems