Relational Interpreters in miniKanren

(WORKING ROUGH DRAFT — DRAFT 0)

William E. Byrd

January 12, 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

Although this book contains a brief introduction to Scheme, and an introduction to miniKanren, the book is not intended as a tutorial on the fundamentals of programming, nor as an introduction to functional programming. Similarly, the book is not intended to be a primer on the fundamentals of programming

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

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

language theory, design, or implementation. While I do try to explain important Scheme and programming languages concepts as they arise (such as lexical scope, closures, and environment-passing interpreters), I assume the reader has enough experience and knowledge to follow along with minimal examples and explanations of these fundamental concepts. If you've encountered these ideas before, and just need a little refresher, I hope the level of explanations and examples will be helpful and sufficient. If you are familiar with functional programming and interpreters, but don't know Scheme, the examples and explanation should also be helpful and sufficient. If you are familiar with some version of miniKanren or microKanren, the chapters on miniKanren should be helpful, since we'll be using aspects of the `faster-miniKanren` implementation of miniKanren that extend (and may differ from) the languages described in the first and second editions of *The Reasoned Schemer*, the microKanren papers, my dissertation, and other miniKanren literature.

Since I know different readers will be coming to this book with very different backgrounds, I've added "pretests" to the Scheme and miniKanren introduction chapters, to help you determine if you already know the concepts well enough to skip ahead. Even if you are a Scheme expert, you should probably read the section on pattern matching to make sure you understand the syntax and semantics of the pattern-matcher we'll be using. If you haven't used `faster-miniKanren` before, or a miniKanren that supports the `=/=`, `symbolo`, `numbero`, and `absento` constraints, I strongly suggest you read the entire introduction to miniKanren.

0.3 What is not in this book

One important topic this book does not cover is how to implement a miniKanren—for example, how `faster-miniKanren` is implemented. While this is an interesting topic, and is especially important for some advanced optimizations and for implementing new constraints, this book focuses on writing interpreters as relations. There are other resouces on implementing simple miniKanrens, such as the papers on microKanren [TODO cite these], which is the basis for the miniKanren implementation in the second edition of *The Reasoned Schemer* [TODO cite].

0.4 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.

0.4.1 Getting `pmatch` from GitHub

0.4.2 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.

0.4.3 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

0.4.4 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

0.5 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. Dan encouraged me to write a short and direct primer on Scheme with only the needed parts of the language. 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

Enough Scheme to get by

We need to know some Scheme, since Scheme is the host language for the `faster-miniKanren` version of miniKanren we will be using. `faster-miniKanren` inherits Schemely features such as `cons` pairs, `quote`, and `letrec`.

We also need to know some Scheme because we will be writing interpreters for subsets of Scheme. In particular, we need to feel comfortable with the evaluation rules for Scheme, including the notions of expressions and values.

And we need to know some Scheme if we want to be able to read much of the miniKanren literature.[1]

1.1 A few comments on Scheme

small core

compositional

few exceptions to rules

very powerful—lots of ways to do meta-programming, including the ability to extend the syntax of the language

great for writing interpreters, compilers, and DSLs

[1] A reading knowledge of OCaml would also be helpful for reading the miniKanren literature that uses OCanren, a miniKanren-like language embedded in OCaml.

1.2 The Scheme reports, versions of Scheme, and implementations of Scheme

1.3 Which version and implementations of Scheme we are using, and why

1.4 What we need to know about Scheme, and when

1.5 Useful Scheme resources

[todo add full references and URLs; can point to the relevant sections as I describe aspects of Scheme]

The Scheme Programming Language, 4th Edition

The Chez Scheme User's Guide [TODO check spelling]

R6RS

1.6 Pretest

a "pre-test" for Scheme, so the reader can see if they need to read any of this

Even a reader who knows Scheme might want to read the pattern matching section

We also describe a few important differences between Scheme and Racket, to ensure the reader can use either one

1.7 Numbers

In this book we restrict ourselves to non-negative integers, which may be of any size:

```
5
42
0
37623489762387946782365476
```

1.8 Booleans

The Boolean `#f` represents "false", while the Boolean `#t` represents "true".

1.9 quote and symbols

In addition to numbers and Booleans, Scheme can represent abstract concepts and symbolic data using *symbols*, sometimes called *quoted symbols*.

If we want to create a symbol to represent the abstract concept of “love”, we can write `(quote love)` which produces the symbol `love`. Because symbols are used so often in Scheme, the equivalent shorthand notation `'love` can also be used to produce the symbol `love`.

1.10 Expressions, values, and evaluation

In Scheme terminology, `(quote love)` is an *expression*. In Scheme, expressions are *evaluated* to produce *values*. In this case, the expression `(quote love)` evaluates to the value `love`, which is a symbol. [todo consider pointing out that in Racket, by default, `'love` will be displayed, and how to adjust that setting]

All Scheme symbols are values. Numbers and the Booleans `#f` and `#t` are also values.

In Scheme we can also quote numbers and Booleans. For example, the expression `(quote 5)` evaluates to the value `5`, which is a number. Similarly, the expression `(quote #f)` evaluates to the value `#f`, which is a Boolean.

Actually, we don’t need to quote numbers or Booleans in Scheme—numbers and Booleans are “self-evaluating” (or “self-quoting”). For example, the expression `42` evaluates to the value `42`, which is a number. The expression `#t` evaluates to the value `#t`, which is a Boolean. Scheme symbols, on the other hand, are not self-evaluating, and must be explicitly quoted.[2]

As shorthand, we write “the expression `(quote 5)` evaluates to the value `5`” as:

```
(quote 5) => 5
```

where the arrow `=>` can be read as “evaluates to”.

Similarly, we can write

```
(quote #f) => #f
(quote love) => love
'love => love
(quote quote) => quote
'quote => quote
42 => 42
6375764356 => 6375764356
```

and

```
#t => #t.
```

1.10.1 The general evaluation rule for quote

We know that

[2]Scheme symbols must be explicitly quoted so that they are distinct syntactically from variable references, which will encounter shortly.

```
(quote 0) => 0
(quote 1) => 1
...
(quote 42) => 42
...
(quote 3765783657849) => 3765783657849
...
```

and so forth.

We can generalize our “evaluates to” `=>` notation; the more general *evaluation rule* for quoting numbers is:

```
(quote <num>) => <num>
```

for any number `<num>`. We use the name `num` surrounded by the angle brackets `<` and `>` to represent any number.

Similarly, the evaluation rule for quoting Booleans is:

```
(quote <bool>) => <bool>
```

for any Boolean `<bool>`. (Of course there are only two Boolean values, `#f` and `#t`.) And the evaluation rule for quoting symbols is

```
(quote <sym>) => <sym>
```

for any symbol `<sym>`.

More generally, the evaluation rule for `quote` is:

```
(quote <datum>) => <datum>
```

The word *datum* is the singular form of *data*. Numbers, Booleans, and symbols are three types of data we have encountered so far.

1.11 `define`, definitions, and variables

We can use `define` to give a name to a value.

For example,

```
(define x 5)
```

gives the name `x` to the number `5`, while

```
(define cool-cat (quote Sugie))
```

gives the name `cool-cat` to the symbol `Sugie`.[3]

More generally, we can write:

```
(define <id> <expr>)
```

where `<id>` is any Scheme *identifier* (such as `x`, `my-cat`, `Hello_there=137^`, or `関連-42`) and where `<expr>` is any expression.

A use of `define` is called a *definition*. A definition is neither an expression nor a value—it is a *statement*. While evaluation of expressions produces values, statements are evaluated for their *effects*. The effect of evaluating `(define x 5)` is to introduce a new *variable* named `x` that is *bound* to the number `5`.[4]

[3] Actually, in Scheme `(define x 5)` gives the name `x` to a *location* that contains the value `5`. It is possible to assign a different value to the location named by `x` using `set!`—for example, `(set! x 6)`. We will avoid the use of `set!` for now, which means we pretend that `define` just gives a name to a value.

[4] Actually, the variable `x` is bound to a *location* that contains the value `5`.

Once we have defined a variable (such as `x`), we can *reference* (or *refer to*) that variable to get the value to which it is bound (such as the number 5, in the case of the variable `x`).

We can see the behavior of `define` and variable reference at the Chez Scheme Read-Eval-Print Loop, or *REPL*. First we start Chez Scheme:

```
Chez Scheme Version 10.1.0
Copyright 1984-2024 Cisco Systems, Inc.

>
```

and then define `x` to be `5`:

```
Chez Scheme Version 10.1.0
Copyright 1984-2024 Cisco Systems, Inc.

> (define x 5)
>
```

The `>` prompt on the line following `> (define x 5)` indicates that Chez has evaluated the statement `(define x 5)` and is ready to evaluate the next expression or statement. To save space, we'll not show the `>` prompt whenever an expression evaluates to a value that is printed at the REPL.

Now that we have defined the variable `x`, we can refer to it:

```
> x
5
```

`x` is an expression (a variable reference) that evaluates to the value `5` (a number).

(In our arrow notation, we would write `x => 5`.)

Let's define another variable, like we did above:

```
> (define cool-cat (quote Sugie))
>

> cool-cat
Sugie
```

`cool-cat` is an expression (a variable reference) that evaluates to the value `Sugie` (a symbol).

What happens if we refer to an *unbound variable*—that is, a variable that has not been defined?

```
> w

Exception: variable w is not bound
Type (debug) to enter the debugger.
```

Chez Scheme evaluates the expression `w`, which is a variable reference. Since `w` is an unbound variable, Chez is not able to determine the value bound to `w`. Instead, Chez Scheme *throws an exception* [todo check terminology: throw exception?] indicating that `w` is a variable that is not bound.

Let's define `w` to have the same value as does the variable `x`:

```
> (define w x)
> w
5
```

Recall the syntax for uses of `define`:

`(define <id> <expr>)`

Also recall that `(define x 5)` is a statement rather than an expression. What happens if use a definition where an expression is required? Chez Scheme complains by throwing a different type of exception:

```
> (define z (define y 6))

Exception: invalid context for definition (define y 6)
Type (debug) to enter the debugger.
```

We have now encountered the crucial notions of Scheme expressions, values, and statements, which we will need in order to understand and write interpreters.[5]

1.12 Procedures and procedure application

What if we would like to add one to the number five? One way to do this in Scheme is to write the expression `(add1 5)`:

```
> (add1 5)
6
```

The expression `(add1 5)` is an example of a *procedure application*, or just *application* (sometimes called a *procedure call*, or just *call*).

If the expression `(add1 5)` is a procedure application, then what is `add1`? Let's find out at the REPL:

```
> add1
#<procedure add1>
```

Aha! `add1` is a variable that is bound to a *procedure*. In Scheme procedures are values, just like numbers, Booleans, and symbols. `add1` is a *built-in* procedure that is bound in Chez Scheme's *initial environment*, which is the default set of variable bindings that exist when Chez is started. We can extend or modify the initial environment, as we did before using `define`.

We can nest procedure applications in Scheme:

[5] miniKanren also has the notions of expressions, values, and statements, and introduces the new notion of *terms*, a generalization of the notion of values. [todo add crossref]

```
> (add1 (add1 5))
7
```

There are many built-in procedures in the initial Scheme environment, and even more in the initial Chez Scheme environment, since Chez extends Scheme with many additional procedures. If we want to add two numbers, we can use the built-in Scheme procedure `+`:

```
> (+ 3 4)
7
```

and:

```
> (+ 7835467856 98236472167)
106071940023
```

Of course, we can nest procedure applications:

```
> (+ (add1 (+ 3 (add1 7) (+ 5 6))))
23
```

As is the case with `add1`, `+` is a variable that is bound to a procedure in Chez Scheme's initial environment:

```
> +
#<procedure +>
```

The procedure bound to `add1` takes exactly one argument. In contrast, the procedure bound to `+` is *variadic*, meaning that it can take any number of arguments. For example, the procedure bound to `+` can take three arguments:

```
> (+ 5 6 7)
18
```

one argument:

```
> (+ 5)
5
```

or even zero arguments:

```
> (+)
0
```

The expression `(+)` evaluates to `0` because zero is the additive identity (the number that when added to another number preserves the value of that second number).

1.13 Predicates, including type predicates

In Scheme, a *predicate* is a procedure that, when called, always terminates (without signalling an error), and that always returns one of the two Boolean literals: `#f` or `#t`.

A *type predicate* is a predicate that can be used to determine the type of a value. For example, the predicate
`number?`
is a built-in procedure that determines whether its argument is a number:

```
> number?
#<procedure number?>
> (number? 5)
#t
> (number? (+ 3 4))
#t
> (number? (quote cat))
#f
> (number? #t)
#f
> (number? number?)
#f
```

It is a Scheme convention to end the names of predicates with a question mark. Also by convention, many people pronounce the `?` at the end of the predicate's name as "huh"; for example, `number?` is pronounced "number-huh".

Scheme's built-in type predicates also include `boolean?`, `symbol?`, and `procedure?`:

```
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? 5)
#f
> (boolean? (+ 3 4))
#f
> (boolean? (quote cat))
#f
> (boolean? boolean?)
#f
> (boolean? (number? 5))
#t
> (boolean? (number? #f))
#t

> (symbol? (quote cat))
#t
```

```
> (symbol? 5)
#f
> (symbol? (+ 3 4))
#f
> (symbol? #f)
#f
> (symbol? symbol?)
#f
> (symbol? (symbol? 5))
#f

> (procedure? procedure?)
#t
> (procedure? +)
#t
> (procedure? add1)
#t
> (procedure? 5)
#f
> (procedure? (+ 3 4))
#f
> (procedure? (quote cat))
#f
> (procedure? (quote +))
#f
> (procedure? #t)
#f
> (procedure? (symbol? (quote cat)))
#f
```

1.14 `if`, test expressions, and truthiness

We can make choices in Scheme using an `if` expression, which is of the form

```
(if <test> <consequent> <alternative>)
```

where `<test>`, `<consequent>`, and `<alternative>` are all expressions. For example, in the expression `(if #t 3 4)`, the *subexpression* `#t` is in the *test position*, the subexpression `3` is in the *consequent position*, and the subexpression `4` is in the *alternative position*.

The rule for evaluation of an `if` expression is that first the test subexpression is evaluated. If the test subexpression evaluates to a true value, then the consequent subexpression is evaluated, and the resulting value is the value of the entire `if` expression. If the test subexpression evaluates to a false value, then the alternative subexpression is evaluated, and the resulting value is the value of the entire `if` expression.

For example:

```
> (if #t 3 4)
3

> (if #f 3 4)
4
```

Of course we can use more complex expressions for the consequent and alternative subexpressions:

```
> (if #t (+ 3 4) (+ 5 6))
7
> (if #f (+ 3 4) (+ 5 6))
11
```

And we can use more complex expressions for the test subexpression:

```
> (if (number? 72634786) (+ 3 4) (+ 5 6))
7
> (if (symbol? 72634786) (+ 3 4) (+ 5 6))
11
```

#t is not the only true value in Scheme. In fact, *any* value in Scheme other than #f is considered true. For example, both 5 and 0 are considered true values in Scheme.

```
> (if 42 (+ 3 4) (+ 5 6))
7
> (if (* 6 7) (+ 3 4) (+ 5 6))
7
> (if 'cat (+ 3 4) (+ 5 6))
7
```

1.15 Evaluation order and special forms

special forms vs. application

keywords

quote, define, and if are keywords; (quote <datum>), (define <id> <expr>), and (if <expr> <expr> <expr>) are special forms.

Recall that in the initial Scheme environment + is a variable bound to a procedure that adds zero or more numbers:

```
> +
#<procedure +>
```

In contrast, in the initial Scheme environment quote is the *keyword* of a special form. Recall that (quote <datum>) is the general syntax for a quote expression. The expression (quote cat) evaluates to the symbol cat. However, evaluating the keyword quote by itself leads to a *syntax error*:

```
> quote

Exception: invalid syntax quote
Type (debug) to enter the debugger.
```

1.16 Comments

Any characters on a line following the `;` character will be ignored. For example,

```
(* 3 4) ; (/ 5 0)
```

evaluates to `12`.

The entire S-expression following `#;` will be ignored. For example,

```
(+ 4 5) #;(* 6
             7)
```

Any characters between matching `#|` and `|#` will be igored. For example:

```
(list
  (+ 3 4)
  #|
  erfjkhrj hfjk
  kjrhjkrheg rjghjer gj
  rghrejhj rjegh jrehk

  jehjkf klh fe
  |#
  (* 5 6)
  )
```

is equivalent to

```
(list
  (+ 3 4)
  (* 5 6)
  )
```

which is equivalent to `(list (+ 3 4) (* 5 6))`.

1.17 `cond`

1.18 A few other predicates

```
zero?
   even?
   odd?
```

1.19 Lists

`list`

`list?`

empty list (quoted)

`null?`

quoted non-empty lists

nested lists

1.20 Pairs and improper lists

`cons`

`pair?`

1.21 S-expressions

[todo need to introduce the concept of the s-expression. Now might be a good time, since we have symbols, numbers, booleans, pairs]

1.22 `lambda`

1.23 Equality predicates

`=`

`eq?`

`equal?`

1.24 Simple examples

1.24.1 `member?`

1.24.2 `length`

1.24.3 `append`

1.24.4 `assoc`

1.25 `let`

1.26 `letrec`

1.27 Lexical scope

1.28 More examples

1.28.1 `append` (`letrec` version)

1.28.2 `even?` and `odd?` (`define` version)

1.28.3 `even?` and `odd?` (`letrec` version)

1.28.4 Curried adder

spelling of Curried?

1.29 `eval`

1.30 Pattern matching

1.31 Grammar for our subset of Scheme

1.32 Differences between Scheme and Racket

evaluation order
printed rep of quoted values
pattern mathing
require vs load
repl usage
`eval` usage

1.33 Exercises

Chapter 2

A whirlwind introduction to relational programming in miniKanren

2.1 What is relational programming?

2.2 Which version of miniKanren we are using, and why

faster-miniKanren without defrel

2.3 Useful miniKanren resources

2.4 Pretest

someone who has read TRS1 or TRS2, or who has implemented microKanren, still needs to know about `=/=`, `symbolo`, `numbero`, `absento`, and the differences between miniKanren in those books and in this book

2.5 miniKanren as an embedded DSL, and otherwise

Scheme as host language

2.6 Core miniKanren

2.6.1 `==`

simlarity to equal? (but not to eq?)
first-order syntactic unification

2.6.2 `run`^{n}

2.6.3 `conde`

2.6.4 `fresh`

2.6.5 `run*`

2.6.6 What miniKanren inherits from Scheme

2.7 Logic variables (or, what does "variable" even mean?)

2.8 Expressions and terms

2.9 Groundness, and the parts of Scheme we can safely use

2.10 Relational vs. non-relational programming in miniKanren

2.11 Simple examples

2.11.1 `appendo`

2.11.2 `membero` (broken version)

2.12 Other useful constraints

2.12.1 `=/=`

disequality

2.12.2 `symbolo` and `numbero`

not needed in OCanren, for example

2.12.3 `absento`

prevention of quoted closures (not needed in OCanren) and other uses, such as `not-in-envo` in split env

2.13 miniKanren Grammar

beware nesting run or ==, calling Scheme eliminators, unifying with procedures, assuming a term is ground, assuming Scheme can handle even ground logic variables as values

revist in style and gotchas chapter

2.14 More examples

2.14.1 `membero` (fixed version)

2.14.2 Differences between the miniKanren in this book and other miniKanrens

TRS1

TRS2

microKanren

core.logic

OCanren

2.14.3 Exercises

Chapter 3

miniKanren style and common pitfalls

"Will's Rule"

syntactic issue 1: lambda (implicit begin) containing more than one goal expression (without a fresh wrapping those goals)—very hard to debug, since only one of the goals is actually run—defrel prevents this problem

syntactic issue 2: nesting a goal expression inside of a call to ==—can actually succeed, although rarely does what you would intend

use of car, cdr, +, etc.

assuming a Scheme function can operate on the value of a ground logic variable

unifying with a Scheme procedure

mixing Scheme and mk code in a way that doesn't preserve relationality

incorrect tagging

Chapter 4

Debugging miniKanren code

4.1 Debugging unexpected failure

leave all args fresh

comment out clauses and goals

4.2 Taming and debugging apparent divergence

`run 1` vs. `run*`

run program with all arguments ground

reordering conjuncts

adding a depth counter

adding bounds (as in rel interp)

tabling

using occur check, presumably?

4.3 Debugging interpreters (and interpreter-like programs)

how to build up a `conde`-based program, such as an interpreter, one expression at a time Dan Friedman-style and then run/test it

run program "forward" to test it

perhaps inclue alternative `run` interafce/streaming/alternative set-based test macro

Chapter 5

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

Chapter 6

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.

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

(defrel (evalo exp val)
  (eval-expo exp '() val))

(defrel (eval-expo exp env val)
  (conde
    ((fresh (v)
       (== `(quote ,v) exp)
       (not-in-envo 'quote env)
       (absento 'closure v)
       (== v val)))
    ((fresh (a*)
       (== `(list . ,a*) exp)
       (not-in-envo 'list env)
       (absento 'closure a*)
       (proper-listo a* env val)))
    ((symbolo exp) (lookupo exp env val))
    ((fresh (rator rand x body env^ a)
```

```
       (== `(,rator ,rand) exp)
       (eval-expo rator env `(closure ,x ,body ,env^))
       (eval-expo rand env a)
       (eval-expo body `((,x . ,a) . ,env^) val)))
    ((fresh (x body)
       (== `(lambda (,x) ,body) exp)
       (symbolo x)
       (not-in-envo 'lambda env)
       (== `(closure ,x ,body ,env) val)))))

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

(defrel (proper-listo exp env val)
  (conde
    ((== '() exp)
     (== '() val))
    ((fresh (a d t-a t-d)
       (== `(,a . ,d) exp)
       (== `(,t-a . ,t-d) val)
       (eval-expo a env t-a)
       (proper-listo d env t-d)))))

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

Chapter 7

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.

Chapter 8

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
```

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

;; 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 exp val)
  (eval-expo exp '(() . ()) val))

(defrel (eval-expo exp env val)
  (conde
    ((fresh (v)
```

```
       (== `(quote ,v) exp)
       (not-in-envo 'quote env)
       (absento 'closure v)
       (== v val)))
    ((fresh (a*)
       (== `(list . ,a*) exp)
       (not-in-envo 'list env)
       (absento 'closure a*)
       (proper-listo a* env val)))
    ((symbolo exp) (lookupo exp env val))
    ((fresh (rator rand x body env^ a env-vars^ env-vals^)
       (== `(,rator ,rand) exp)
       (== `(,env-vars^ . ,env-vals^) env^)
       (eval-expo rator env `(closure ,x ,body ,env^))
       (eval-expo rand env a)
       (eval-expo body
                  `((,x . env-vars^) . (,a . env-vals^))
                  val)))
    ((fresh (x body)
       (== `(lambda (,x) ,body) exp)
       (symbolo x)
       (not-in-envo 'lambda env)
       (== `(closure ,x ,body ,env) val)))))

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

(defrel (not-in-envo x env)
  (fresh (env-vars env-vals)
    (== `(,env-vars . ,env-vals) env)
    (absento x env-vars)))

(defrel (proper-listo exp env val)
  (conde
    ((== '() exp)
     (== '() val))
    ((fresh (a d t-a t-d)
       (== `(,a . ,d) exp)
       (== `(,t-a . ,t-d) val)
```

```
        (eval-expo a env t-a)
        (proper-listo d env t-d)))))

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

(defrel (lookupo x env t)
  (fresh (rest y v env-vars-rest env-vals-rest)
    (== `((,y . ,env-vars-rest) . (,v . ,env-vals-rest)) env)
    (conde
      ((== y x) (== v t))
      ((=/= y x) (lookupo x `(,env-vars-rest . ,env-vals-rest) t)))))
```

Chapter 9

Extending the interpreter to handle `append`

multi-argument and variadic `lambda` and application

Chapter 10

Using a non-empty initial environment

`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 11

Adding explicit errors

Chapter 12

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 13

Adding mutation

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

support mutiple pairs and have an explicit store

Chapter 14

Writing a parser as a relation

Chapter 15

Writing a type inferencer as a relation

Chapter 16

Build your own Barliman

Chapter 17

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 18

Open problems