Relational Interpreters in miniKanren

(WORKING ROUGH DRAFT — DRAFT 0)

William E. Byrd

December 25, 2024

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]

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.

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

[todo show that quote is more general:

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

and show that you can also quote numbers and Booleans, and that those expressions evaluate to the numbers or Booleans themselves. It's not needed to quote these "self-evaluating literals"]

[todo could nest quotes: show that even though the expressions 5 and (quote 5) both evaluate to the value 5, the expressions (quote (quote 5)) amd (quote 5) do not evaluate to the same value]

1.11 `define`

We can use `define` to name numbers and Booleans.

For example,

```
(define x 5)
```

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

```
(define cats-are-cool #t)
```

gives the name `cats-are-cool` to `#t`.

1.12 Variables

variable reference

1.13 Type predicates and procedure application

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.

`number?`

It is a Scheme convention to end the names of predicates with a question mark. Also by convention, many people "huh?"

`(number? 5) => #t`

`(number? #t) => #f`

`boolean?`

`symbol?`

1.14 `if`

`#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.

1.15 Evaluation order and special forms

special forms vs. application

keywords

1.16 Comments

`;`

`#;`

`#|` and `|#`

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 `lambda`

1.22 Procedures

`procedure?`
  variable ref to procs

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

CBV lambda-calc plus quote and cons

a list for env

tagged list to represent closure

grammar for the language we are interpreting

Chapter 6

Rewriting the simple environment-passing Scheme interpreter in miniKanren

Chapter 7

Quine time

McCarthy challenge given in 'A Micromanual for LISP'
Quines, Twines, Thrines
absento trick to generate more interesting Twines and Thrines

Chapter 8

Using a two-list representation of the environment

Chapter 9

Extending the interpreter to handle `append`

Chapter 10

Adding explicit errors

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