metaKanren: miniKanren Program Synthesis, Relationally

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science and Engineering by Research

by

Bharathi Ramana Joshi
2019121006
bharathi.joshi@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
October 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "metaKanren: A Relational Metacircular Language" by Bharathi Ramana Joshi, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. Suresh Purini

_____ _____

Date Co-Adviser: Dr. William E. Byrd

To all my teachers; but Ms. Krishnakumari, my high-school mathematics teacher in particular, whose love for mathematics sparked and nourished my own.

Acknowledgments

Amma: Words are inadequate to describe the gratitude I have towards my mother, ***Arpita Joshi***. I am ever grateful for her constant love and unwavering support.

Dr. William Byrd: Will has been a amazing advisor. I am particularly grateful for his patience, kindness, and encouragement. Needless to say, without Will's guidance, I would not be writing this thesis today.

Dr. Talia Ringer: Talia started the SIGPLAN-M initiative without which I never would have met Will. I am sure SIGPLAN-M has changed the lives of many others across the world, it has certainly changed mine.

Dr. Suresh Purini: Suresh Sir, my advisor at IIIT Hyderabad, gave me total research freedom to work on whatever my heart desired. I am not only grateful for this privilege, but also for his support in uncountable other ways.

Dr. Nazia Akhtar: I am grateful to Nazia ma'am for constantly believing in me, even when I found it difficult to believe in myself. I am thankful for her support and encouragement.

Friends at IIIT: I have made numerous friends through my years at IIIT Hyderabad and it is impossible to thank them all here. A few friends I'd like to mention here for their love and support are ***Rishav Kundu***, ***Ayan Biswas***, ***Maheswara Rao***, ***Siddharth Bhat***, ***Salman Ahmed***, ***Apoorva Tirupathi***, ***Abhinav S Menon***, ***Archit Goyal***, ***Tathagata Raha***, ***Ishan Sanjeev Upadhyay***, ***Mohammad Nomaan Qureshi***, ***Shivaan Sehgal***, ***KV Aditya Srivatsa***, and ***Monil Gokani***. I cannot imagine my life at IIIT without these people being a crucial part of it.

The Theory [Reading] Group: I am grateful for the inspiring people and curious environment I got to experience at the Theory Group.

KMIT: I began my adventures in programming languages at my previous college, *KMIT*, and my friends there are an important part of my journey. I am fortunate to have met ***Sai Sandeep Mutyala***, ***Bhoomi Chavan***, ***Kushal Mor***, and ***Meghana Palakodeti*** and grateful that they were such wonderful, caring, and supportive friends. Additionally, I am thankful for having been a part of ***kMITRA***.

I am grateful to my family for their invariant support.

I am sure I have missed out many other people who I owe my gratitude, and I sincerely apologize for the same.

Abstract

Relational interpreters, interpreters written for programming languages as mathematical relations, have been studied for at least over a decade. However, while relational interpreters are mostly written in miniKanren, a relational language where programs are mathematical relations, the interpreted language tends to be non-relational.

This dissertation presents *metaKanren*, a purely relational miniKanren-like language, and relational interpreter implemented as a deep embedding for metaKanren. We also construct a relational interpreter for a minimal relational language as a shallow embedding and point out the limitation of a shallow embedding, further emphasizing the need for a deep embedding. We show how one can perform program synthesis of miniKanren programs using our relational interpreter for metaKanren. We also show how to extend metaKanren and its relational interpreter to handle datatype constraints. Finally, we also show how metaKanren can be made fully metacircular, and precisely point out the problems in building a metacircular interpreter for miniKanren.

This dissertation argues that it is feasible and useful to build a metacircular relational interpreter for miniKanren.

Contents

List of Figures

Figure Page

List of Tables

Chapter 1

Introduction

Building a metacircular relational interpreter for miniKanren is useful and feasible.

Let us deconstruct the above thesis statement. The first technical term used is *metacircular*. A *metacircular* interpreter is an interpreter for a programming language that is written in the very language that is being interpreted [Abelson and Sussman, 1996]. Such an interpreter may be implemented by simply translating constructs in the interpreted language to the corresponding constructs in the interpreting language. For example, to evaluate an `if` expression in the interpreted language, the interpreter may evaluate a corresponding `if` expression in the interpreting language. Such an interpreter, in some sense, does not elucidate upon the semantics of the interpreted language. This is because in such an interpreter, the interpreted language inherits the semantics of the interpreting language, allowing the metacircular interpreter to get away without having to explicitly define the semantics of the interpreted language. An interpreter where the interpreting language merely translates interpreted language constructs to interpreting language constructs is called as a *shallow embedding* of interpreter, or a shallow interpreter in short. On the other hand, rather than a mere translation, a metacircular interpreter may explicitly define the semantics of the interpreted language as operations upon first order representations in the interpreted language. This latter type of interpreter is called a *deep embedding* of an interpreter, or a deep interpreter in short. We explore both shallow and deep metacircular interpreters in this dissertation. Also, we use the term "metacircular" to describe interpreters where the interpreted language is a subset of the interpreting language in this dissertation. We reserve the term "fully metacircular" to describe interpreters where the interpreted language is equivalent to the interpreting language.

The next technical term used is *relational*. *Relational* programming advocates for writing computer programs as mathematical relations. As an example, consider the relational program to append two lists, `appendo`. `appendo` is a three argument relation between three lists, where the third argument is the concatenation of the first two arguments. Consider the following call to `appendo`.

```
(appendo '(1 2) '(3 4) out)
```

Here the first two arguments are lists of numbers and the third argument is the variable `out`. In the above call, `appendo` associates the variable `out` with the list `(1 2 3 4)`. A defining characteristic

of relational programs is their indifference towards what is input and what is output. For example, consider the following call to `appendo`.

```
(appendo '(1 2) out '(1 2 3 4))
```

Here, the first and the third arguments are lists of numbers whereas the second argument is a variable. When called in this way, `appendo` associates the variable `out` with the list `(3 4)`. Another defining characteristic of relational programs is their ability to run even when multiple arguments to a relation are variables. For instance, consider the following call to `appendo`.

```
(appendo list-1 list-2 '(1 2 3 4))
```

Note that both the first and the second argument in this call to the relation `appendo` are variables. In this case, it produces many answers associating the variables `list-1` and `list-2` with various values such as `(() (1 2 3 4))`, `((1) (2 3 4))`, `((1 2) (3 4))`, etc. These two properties enable the same relational program to compute different things depending upon which arguments are variables. Thus, a single relational program may be used to solve multiple problems. For example, an interpreter for a programming language written as a relation can not only interpret but also synthesize programs. A type-checker written as a relation can not only check for type soundness of programs but also act as a type-inferencer. A proof-checker written as a relation can not only check for the validity of proofs, but also synthesize proofs for theorems. miniKanren is a family of languages built for relational programming. In this dissertation, unless specified otherwise, by miniKanren we specifically mean the language referred to as "Extended miniKanren" presented by Byrd et al. [2012]. We direct the interested reader to Byrd [2009] to learn more about relational programming.

A class of relational programs, *relational interpreters* – interpreters for programming languages written as relations, have been used to solve various problems. Consider an interpreter for a programming language as a two argument relation, `evalo`, between the interpreted program and the program's value. To concretize this example further, let us assume the interpreted language is Scheme. Then, for example, calling the relational interpreter `evalo` as `(evalo '(cons 1 2) out)` will associate the variable `out` with the pair `(1 . 2)`. One problem that the relational interpreter `evalo` can be used to solve is that of generating programs with required properties. For example, to generate quines, i.e. a program that prints a copy of its own source code as the output, one need only call the `evalo` relation as `(evalo q q)`. The relational interpreter `evalo` will then associate the variable `q` with a Scheme quine. Generating programs with required properties is just one use case of relational interpreters. Some other uses include test driven development, program synthesis based on input-output examples, etc [Byrd et al., 2017].

Relational interpreters for Scheme have been studied for at least over a decade now [Byrd et al., 2012, 2017, Zhang et al., 2018, Hemann and Friedman, 2020]. The relational interpreters themselves are usually written in miniKanren, while the interpreted language tends to be a non-relational language. As a consequence, while miniKanren is used to relationally interpret other languages, few efforts have been made to relationally interpret miniKanren itself (see Chapter 6). A metacircular relational inter-

preter for miniKanren would let us not only relationally interpret, but also perform program synthesis of miniKanren programs. Attempting this challenge would also further the understanding of relational interpreters. Also, a metacircular interpreter serves as a test of the expressivity of a language. Furthermore, Scheme, the host language in which miniKanren was built as an embedded domain-specific language, has a rich tradition of building metacircular interpreters. Hence, the first part of my thesis: building a metacircular relational interpreter for miniKanren is useful. Chapters 3, 4, and 5 show how to implement metacircular relational interpreters for various miniKanren-like languages. Hence, the second part of my thesis: building a metacircular relational interpreter for miniKanren is feasible.

Building a metacircular relational interpreter for miniKanren involves several important design decisions, two of which we talk about here. One decision is whether the metacircular relational interpreter be shallow or deep? We explore both of these choices in this dissertation. Another important design decision is how to interface between the interpreting miniKanren and metaKanren programs. We choose to go with a `run` interface similar to the one between Scheme and miniKanren. This choice requires us to implement a *reifier* in miniKanren for metaKanren. A *reifier* clarifies and presents the information about various constraints on variables in a metaKanren program to the programmer. Having to implement a reifier in turn dictates the kinds of first-order representations may use in the metacircular relational interpreter. Furthermore, this choice makes it problematic to achieve full metacircularity as reifying certain constraint information turns out to be non-trivial in a relational setting. We go into the intricacies of this problem in Chapter 7.

1.1 Contributions

Concretely, our contributions are as follows.

1. We present metaKanren, a miniKanren-like language with a relational deep interpreter capable of running programs with `run*` semantics at the interpreted level.

2. We show how to extend metaKanren to handle the `symbolo` and `numbero` type constraints.

3. We show how metaKanren can serve as a useful starting point towards a fully metacircular interpreter for miniKanren and precisely point out the problems in achieving full metacircularity.

1.2 Outline

The dissertation is laid out as follows.

- Chapter 1 is the introductory chapter, which discusses the dissertation's aims and scope, and provides some motivation for our work.

- Chapter 2 provides the background knowledge required to follow the rest of the dissertation by first defining the terminology used and then walking the reader through the basics of relational programming in miniKanren.
- Chapter 3 demonstrates how to implement a shallow metacircular relational interpreter, explores program synthesis via such an interpreter, and points out the limitations of such an approach.
- Chapter 4 presents the metaKanren language, demonstrates program synthesis of metaKanren programs using a relational interpreter for metaKanren, and walks the reader through an annotated implementation of a relational interpreter for metaKanren.
- Chapter 5 demonstrates how our relational interpreter can be extended to handle type constraints and explores metaKanren program synthesis with type constraints.
- Chapter 6 compares our work with other relational interpreters for miniKanren-like languages.
- Chapter 7 concludes with a summary of our work and outlines our future vision for metaKanren.

Chapter 2

Background

miniKanren is a family of logic programming languages that emphasize *logical purity*, i.e. writing computer programs as mathematical relations. The resulting paradigm of programming is referred to as *relational programming*. miniKanren supports a variety of relational programming idioms and techniques [Byrd, 2009]. Furthermore, miniKanren is quite minimal and has only a handful of operators. In fact, miniKanren may be viewed as an embedded domain specific language for Scheme that extends Scheme with three new operators for relational programming. Its minimality makes miniKanren convenient to implement. As a consequence, miniKanren serves as an appropriate relational language for which we can implement a *relational interpreter* (see Section 1). The purpose of this chapter is to familiarize the reader with miniKanren and provide context for our work.

2.1 Terminology

In this section, we define various technical terms we use throughout the thesis.
The following technical terminology is used when we discuss programming in miniKanren.

1. A *unification variable*, *miniKanren variable*, or a *logic variable* is a variable over *terms* (defined below). miniKanren computes the possible term values of logic variables for a given set of constraints.

2. A *term* is inductively defined as:

   (a) a Scheme constant, i.e., a Boolean constant `#t` or `#f`, the empty list `()`, a number, a string, a symbol; or,

   (b) a logic variable; or,

   (c) a cons pair of two terms.

   Consequently, by *term expressions* we mean expressions that evaluate to terms.

The following technical terminology is used when we discuss implementing interpreters for miniKanren languages.

1. A *state* encapsulates information about the logic variables in a miniKanren program. It contains information about the constraints on logic variables.

2. A *goal* is a single argument function that takes a *state* and returns a stream of states. Consequently, by *goal expressions* we mean expressions that evaluate to a goal.

   miniKanren operators are implemented in terms of goals. The intuition behind such an implementation is that given a state, a miniKanren operator adds to the constraints on the logic variables in the state.

2.2 miniKanren Programming Primer

The aim of this section is to provide the reader with an elementary understanding of programming in miniKanren that is necessary and sufficient to follow the rest of the thesis. As a consequence, this section is not a through exposition to miniKanren programming, and the reader is directed to Friedman et al. [2018], Byrd [2009], and Byrd et al. [2012] for the same. The reader is assumed to be familiar with Scheme as this section explores miniKanren programming in the Scheme embedding.

At a high-level, core miniKanren is Scheme extended with three new operators.

1. ==: The syntax of the == operator is as follows:

   ```
   (== term-expression-1 term-expression-2)
   ```

   This operator produces a goal that attempts to unify the two term expressions with respect to a given state. == is the fundamental core miniKanren operator which is used to construct miniKanren programs.

2. `fresh`: The syntax of the `fresh` operator is as follows:

   ```
   (fresh (variable ...)                    ; variables part
     goal-expression goal-expression ...) ; body part
   ```

   Throughout this thesis, the notation `thing ...` holds for "zero or more occurrences of `thing` and `thing thing ...` holds for "one or more occurrences of `thing`".

   The `fresh` operator produces a goal that evaluates the goal expressions in the body part in logical conjunction by introducing the new logic variables specified in the variables part.

3. `conde`: The syntax of the `conde` operator is as follows:

   ```
   (conde
     [goal-expression goal-expression ...]  ; clause-1
     [goal-expression goal-expression ...]  ; clause-2
   ```

```
...)
```

The `conde` operator produces a goal that evaluates the clauses in logical disjunction with respect to a provided state. All the goals in a single clause are evaluated in logical conjunction.

Before looking at examples for each of the above, we must familiarize ourselves with the interface between the host language Scheme and miniKanren. In order to run a miniKanren program from Scheme, either the `run` or the `run*` interface must be used. The syntax for `run` is:

```
(run n (variable variable ...)          ; variables part
  goal-expression goal-expression ...) ; constraints part
```

where `n` specifies the number of associations miniKanren must compute for the logic variables in the variables part satisfying the constraints specified by the goal expressions in the constraints part.

And the syntax for `run*` is:

```
(run* (variable variable ...)           ; variables part
  goal-expression goal-expression ...) ; constraints part
```

In a `run*`, miniKanren returns all possible associations for the logic variables in the variables part satisfying the constraints specified by the goal expressions in the constraints part..

As an example, consider the following program:

```
(run 1 (out)
  (== 5 out))
⟹ (5)
```

Throughout the thesis the symbol $\Longrightarrow$ will be used to indicate the evaluated value of an expression.

In the above code, we ask one value for the logic variable `out` such that the constraint specified by the == operator holds. The answer returned is a list whose only element is 5. The parentheses signify that miniKanren returns a list of values, which corresponds to the multiset of answers satisfying the constraints.

We may use the `fresh` operator to introduce new logic variables:

```
(run 1 (out)
  (fresh (x)
    (== 3 x)
    (== x out)))
⟹ (3)
```

In the above code, we ask miniKanren one value for the logic variable `out` such that the constraints in the body of the `fresh` operator hold. As the goal expressions in the body of the `fresh` operator

are evaluated in logical conjunction, the logic variable `x` is unified with the constant `3` in the first goal expression and the logic variable `x` is unified with logic variable `out` in the second goal expression. In effect, this unifies `out` with 3.

We may use `conde` to utilize non-determinism:

```
(run 1 (out)
  (conde
    [(== 3 out)]
    [(== 'cat out)]))
⟹ (3)
```

With non-determinism, we may see the difference between a `run 1` and `run 2`:

```
(run 2 (out)
  (conde
    [(== 3 out)]
    [(== 'cat out)]))
⟹ (3 cat)
```

We may also use `run*` to get all possible answers:

```
(run* (out)
  (conde
    [(== 3 out)]
    [(== 'cat out)]))
⟹ (3 cat)
```

Here is a more complicated example using all the above:

```
(run* (out)
  (letrec ([appendo (lambda (l1 l2 l)
                      (conde
                        [(== '() l1) (== l2 l)]
                        [(fresh (a d d-l2)
                            (== (cons a d) l1)
                            (== (cons a d-l2) l)
                            (appendo d l2 d-l2))]))])
```

```
    (appendo '(1 2) '(3 4) out)))
⟹  ((1 2 3 4))
```

`appendo` is defined as the list concatenation relation – it is a three argument relation, where all its arguments are Scheme lists and the third argument is the concatenation of the first two arguments. In the above code, we ask miniKanren all possible values of the third argument of `appendo`, given the first two arguments.

As miniKanren is relational, we can also run `appendo` "backwards":

```
(run* (out)
  (letrec ([appendo (lambda (l1 l2 l)
                      (conde
                        [(== '() l1) (== l2 l)]
                        [(fresh (a d d-l2)
                            (== (cons a d) l1)
                            (== (cons a d-l2) l)
                            (appendo d l2 d-l2))]))])
    (appendo '(1 2) out '(1 2 3 4)))
⟹  ((3 4))
```

In the above code we give `appendo` the first and the third argument and ask miniKanren the possible values of the second argument. This is precisely the flexibility of relational programming.

Chapter 3

Shallow Metacircular Relational Interpreter

One natural way to implement a metacircular interpreter is to implement every feature using itself – logic variables using logic variables, disjunction using disjunction, etc. Such an implementation of a language, where the interpreting language constructs syntactic representations for the expressions in the interpreted language, is known as *shallow embedding* (this is in contrast with a *deep embedding*, where the interpreting language constructs first-order semantic representations instead) [Gibbons and Wu, 2014].

As a materialization of this idea, consider the tiny relational language (the minimality of this language is inconsequential to the limitation we would like to point out by a shallow embedding but keeps the semantics and interpreter simple) with the following grammar:

⟨*program*⟩ ::= `(run 1 (`⟨*id*⟩`)` ⟨*goal-expr*⟩`)`

⟨*goal-expr*⟩ ::= `(==` ⟨*term-expr*⟩ ⟨*term-expr*⟩`)`
| `(fresh (`⟨*id*⟩`)` ⟨*goal-expr*⟩`)`
| `(conj` ⟨*goal-expr*⟩ ⟨*goal-expr*⟩`)`
| `(disj` ⟨*goal-expr*⟩ ⟨*goal-expr*⟩`)`

⟨*term-expr*⟩ ::= `(quote` ⟨*value*⟩`)`
| ⟨*id*⟩

⟨*value*⟩ ::= Any valid Scheme symbol

⟨*id*⟩ ::= Any valid Scheme symbol

Term expressions evaluate to terms, whose grammar is:

⟨*term*⟩ ::= ⟨*logic-var*⟩ | ⟨*symbol*⟩

⟨*logic-var*⟩ ::= Interpreting logic variable

⟨*symbol*⟩ ::= Any valid Scheme symbol

The difference between terms and values is that terms are unified with logic variables, but values may not be.

The corresponding shallow relational self-interpreter[1] implemented using faster-miniKanren's Scheme implementation [Ballantyne, 2022] is:[2]

```
(load "~/code/faster-minikanren/mk-vicare.scm")
(load "../faster-miniKanren/mk.scm")

; Evaluate a program by evaluating the goal expression after
; initializing the environment
(define (eval-programo program out)
    (fresh (q ge)
      (== `(run 1 (,q) ,ge) program)
      ; Interpreted logic variables are symbols
      (symbolo q)
      ; The environment (association list) maps interpreted logic
      ; variables to interpreting logic variables
      (eval-gexpro ge `((,q . ,out)))))

; Evaluate goal expression in an environment
(define (eval-gexpro expr env)
    (conde
      [(fresh (e1 e2 t)
         (== `(== ,e1 ,e2) expr)
```

[1]We reserve the term "metacircular" for an interpreter that interprets every construct it uses, and instead use the term "self-interpreter" for an interpreter that interprets only a subset of the constructs it uses

[2]This code is available here `https://github.com/iambrj/ms-thesis/blob/master/artifact/shallow.scm`.

```
         ; Evaluation of both terms should unify to the same term t
         (eval-texpro e1 env t)
         (eval-texpro e2 env t))]
      [(fresh (x x1 ge)
         (== `(fresh (,x) ,ge) expr)
         (symbolo x)
         ; Translate interpreted fresh logic variable into
         ; interpreting fresh logic variable by extending the
         ; environment
         (eval-gexpro ge `((,x . ,x1) . ,env)))]
      [(fresh (ge1 ge2)
         (== `(conj ,ge1 ,ge2) expr)
         ; Translate interpreted conjunction into interpreting
         ; conjunction
         (eval-gexpro ge1 env)
         (eval-gexpro ge2 env))]
      [(fresh (ge1 ge2)
         (== `(disj ,ge1 ,ge2) expr)
         ; Translate interpreted disjunction into interpreting
         ; disjunction
         (conde
           [(eval-gexpro ge1 env)]
           [(eval-gexpro ge2 env)]))]))

; Evaluate a term expression in an environment
(define (eval-texpro expr env val)
    (conde
      ; Quoted values are self-evaluating
      [(== `(quote ,val) expr)]
```

```
      ; Lookup interpreted logic variables in the environment
      [(symbolo expr) (lookupo expr env val)]))

; Search for a variable in an environment
(define (lookupo x env val)
    (fresh (y v d)
      (== `((,y . ,v) . ,d) env)
      (conde
        [(== x y) (== v val)]
        [(=/= x y)
         (lookupo x d val)])))
```

To get an intuition for how the interpreter works, consider the following example:[3]

```
(run* (x) (eval-programo `(run 1 (z) (== 'cat z))
                         x))
```

⇒

```
(cat)
```

By definition of the shallow interpreter, the inner query variable `z` is unified with the outer query variable `x`. Therefore, a `run*` on `x` will return all possible values of `z` that satisfy the given constraints, explaining the above output.

Now, let us perform program synthesis by providing the expected output and inserting a logic variable from the outer query in the goal expression of the inner query (here, and in the rest of the paper, changes between two consecutive code snippets are highlighted unless mentioned otherwise):

```
(run* (x) (eval-programo `(run 1 (z) (== ,x z))
                         'cat))
```

⇒

```
('cat z)
```

[3]Code for all examples in the shallow interpreter is available here `https://github.com/iambrj/ms-thesis/blob/master/artifact/shallow-examples.scm`

Perhaps surprisingly, in addition to the expected `'cat` we get back the unexpected `z`. Because interpreted logic variables are represented as interpreting logic variables, the unification `(== z z)` in the inner query succeeds for *all* values of the outer query variable, since by definition every value unifies with itself! Thus, in particular, the inner `run` succeeds when the outer query variable unifies with `'cat` because `(== 'cat 'cat)` succeeds.

Now consider this example, where we try to synthesize the branches of a disjunction:

```
(run 4 (e1 e2) (eval-programo `(run 1 (z) (disj ,e1
                                                ,e2))
                                    'cat))
```

⇒

```
'(((== '_.0 '_.0) _.1)
  (_.0 (== '_.1 '_.1))
  ((== 'cat z) _.0)
  (_.0 (== 'cat z)))
```

Note that we use a `run 4` instead of a `run*` for the outer query because there are infinitely many correct ways to synthesize `e1` and `e2`. The first two answers are explained by the behaviour explained above, except we have fresh logic variables (such as `'_.0` and `'_.1`) unifying with themselves instead of ground terms (such as `z`). The next two however, are more interesting. They tell us that if one of the branches unifies the inner query variable `z` with `'cat`, the other branch can be *anything*. This, while correct, is not always desirable and imposes a strong limitation on the expressivity of the language. Concretely, we can never "close-off" a disjunction (i.e. the interpreter always synthesizes branches with logic variables, instead of branches with concrete expressions) when the inner query is restricted to a `run 1` semantics, because with a `run 1` semantics we can only ever say "so-and-so should be *one of the* answers" but we can never say "so-and-so should be the *only* answer(s)"! If we had embedded `run*` semantics, then observe that we can express the latter.

However, we can never have a `run*` semantics for the inner query with a shallow interpreter, since that would require the interpreter to express conditions such as whether a particular goal succeeds/fails and whether a particular logic variable is fresh/ground to be able to collect all possible answers — which goes against the relational nature of the interpreter!

If we had a deep embedding (i.e. semantic representations) the above limitation can be overcome since success/failure of goals and freshness of logic variables can be expressed as conditions on the semantic representations, preserving relationality. This motivates the need for a deep embedding of miniKanren.

Chapter 4

metaKanren

The metaKanren language has been designed to be as close to miniKanren as possible. Thus, the only differences are there two new constructs and a different representation for query numbers. Readers familiar with miniKanren may skip the grammar and go to Section 4.1 where we discuss these differences. We direct readers unfamiliar with miniKanren to Chapter 2.

⟨*metaKanren-program*⟩ ::= `(run*` ⟨*id*⟩ ⟨*goal-expr*⟩`)`
| `(run` ⟨*peano*⟩ ⟨*id*⟩ ⟨*goal-expr*⟩`)`

⟨*goal-expr*⟩ ::= `(==` ⟨*term-expr*⟩ ⟨*term-expr*⟩`)`
| `(fresh (`⟨*id*⟩ ...`)` ⟨*goal-expr*⟩ ...`)`
| `(conde (`⟨*goal-expr*⟩ ...`)` ...`)`
| `(letrec-rel ([`⟨*id*⟩ `(` ⟨*id*⟩ ...`)` ⟨*goal-expr*⟩ ...`]` ...`)` ⟨*goal-expr*⟩ ...`)`
| `(delay` ⟨*goal-expr*⟩`)`

In addition to the above language we have a separate term expression language, consisting of expressions that are arguments to ==:

⟨*term-expr*⟩ ::= `(quote` ⟨*value*⟩`)`
| ⟨*number*⟩
| ⟨*boolean*⟩
| ⟨*id*⟩
| `(cons` ⟨*term-expr*⟩ ⟨*term-expr*⟩`)`

Term expressions evaluate to terms, whose grammar is:

⟨*term*⟩ ::= ⟨*logic-var*⟩
| ⟨*number*⟩

| ⟨*boolean*⟩
| ⟨*symbol*⟩ [other than 'var']
| `()`
| `(`⟨*term*⟩ `.` ⟨*term*⟩`)`

⟨*boolean*⟩ ::= `#t` | `#f`

⟨*number*⟩ ::= Any number `x` such that `(number? x)` evaluates to `#t`

⟨*symbol*⟩ ::= Any symbol `x` such that `(symbol? x)` evaluates to `#t`

⟨*id*⟩ ::= Any symbol `x` such that `(symbol? x)` evaluates to `#t`

⟨*peano*⟩ ::= `()` | `(`⟨*peano*⟩`)`

⟨*logic-var*⟩ ::= `(var .` ⟨*peano*⟩`)`

4.1 Differences between metaKanren and miniKanren

We delineate metaKanren's differences by showing how to transform the following miniKanren program, which evaluates to an infinite stream of alternating 5s and 6s from which the first 3 answers are picked, into a metaKanren program.

```
(run 3 (x)
  (letrec [(fives-and-sixes
             (lambda (x)
               (conde
                 [(== x 5)]
                 [(== x 6)])))]
    (fives-and-sixes x)))
```

The transformations we apply to make it a valid metaKanren program are:

1. Firstly, we note that since the interpreting language is miniKanren, using Arabic numerals such as 3 for the query's count would not be possible, since miniKanren has no support for Arabic numeral arithmetic (unless we were to complicate the interpreting language by adding more features for constraint logic programming Jaffar and Lassez [1987]). Therefore, we use a different representation for natural numbers:

- 0 is represented by the empty list.
- If the Arabic numeral n is represented by the list `l`, then the Arabic numeral $n + 1$ is represented by the list `(l)`.

We refer to natural numbers in the above representation as **Peano numerals**. Thus, the first change we make is replacing 3 by the Peano numeral `(((())))`:

```
(run (((()))) (x)
  (letrec [(fives-and-sixes
             (lambda (x)
               (conde
                 [(== x 5)]
                 [(== x 6)])))]
    (fives-and-sixes x)))
```

2. Next, we note that the relation uses the interpreting language's `letrec` and `lambda` forms to define a recursive relation and procedure application to use this relation. Once again, doing this would not be possible with miniKanren as the interpreting language because it does not have any of `letrec`, `lambda`, or procedure application. Therefore, we introduce a new language construct `letrec-rel` (see syntax 5.1) to define a recursive relation and reimplement application in metaKanren. Thus, by rewriting the program using this, we end up with:

```
(run (((()))) (x)
  (letrec-rel [(fives-and-sixes (x)
                (conde
                  [(== x 5)]
                  [(== x 6)])))]
    (fives-and-sixes x)))
```

3. Finally, we note that a hidden `lambda` is still being used to delay goal evaluation in the recursive call to `fives-or-sixes` (see Friedman [2013] for details on this delayed goal evaluation). For the same reason as above, we introduce a new construct `delay` to delay goal evaluation in metaKanren.

   With these changes, we finally arrive at the following valid metaKanren program:

```
(run (((()))) (x)
  (letrec-rel [(fives-and-sixes (x)
```

```
                    (conde
                      [(== x 5)]
                      [(== x 6)])))]
      (delay (fives-and-sixes x)))
```

Now that we have familiarized ourselves with the metaKanren language, let us see metaKanren in action by looking at some examples.

4.2 Program Synthesis Examples

Our interpreter is the two-argument relation `eval-programo`, where the first argument is a quoted `run`/`run*` expression and the second argument the list of answers. We use the Scheme procedure `peano` (provided in Appendix A) to convert Arabic numerals to Peano numerals (see 1) to improve the queries' readability. Section 4.3 is a commented tour of the interpreter itself, which is not a prerequisite to understand the examples here.

Of course our relational interpreter is capable of running metaKanren programs "forwards", i.e. evaluate a `run`/`run*` expression to generate a list of values for the query's logic variable satisfying the given constraints. For example, consider the following run of `appendo`:[1]

```
(run* (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((appendo (l1 l2 l)
                      (conde
                        [(== '() l1) (== l2 l)]
                        [(fresh (a d l3)
                           (== (cons a d) l1)
                           (== (cons a l3) l)
                           (delay (appendo d
                                           l2
                                           l3)))])))
         (appendo '(1 2) '(3 4) z)))
    x))
```

[1]Code for all examples in this section is available here `https://github.com/iambrj/ms-thesis/blob/master/artifact/chapter-3-examples.scm`

```
⟹ (((1 2 3 4)))
```

Since `x`, the logic variable from the outer `run`, is placed in the result position of `eval-programo`, it contains all possible values of `z` – the logic variable of the inner `run`. The result list has an extra layer of nesting because metaKanren itself runs inside of miniKanren.

On the other hand, since the interpreter is relational in nature, we can run it "backwards" as well, to synthesize parts of metaKanren. We show several interesting examples of metaKanren synthesis in this section.

4.2.1 Synthesis with nested `run*`

Let us go back to the motivating example discussed in Chapter 3. We can synthesize the branch of a disjunction (we provide the succeeding branch, as opposed to synthesizing both branches as discussed in Chapter 3, due to performance restrictions):

```
(run 1 (e)
  (eval-programo `(run* (z) (conde [(== 'cat z)]
                                   ,e))
                 '(cat)))
⟹ ((((delay (== () _.0))) (num _.0)))
```

And metaKanren indeed synthesizes the other branch to a failing goal!

On the other hand, if we were to use a `run 1` for the inner query:

```
(run 1 (e)
  (eval-programo `(run (()) (z) (conde [(== 'cat z)]
                                       ,e))
                 '(cat)))
⟹ (((delay _.0)))
```

We get back a goal that does not fail! When we have `run*` for the inner query, the semantics dictate that the *only answer* should be `cat`. This demonstrates the additional expressivity a deep embedding brings to the table that a shallow embedding lacks (see Section 3).

4.2.2 Synthesizing parts of a user-defined relation

Let us explore a more complicated synthesis example by replacing the arguments to `appendo` in the recursive call in its definition with logic variables instead (i.e. provide a partially defined `appendo` with expected output, and have metaKanren synthesize the left out parts).

Before we proceed to this example however, we first remark on a certain subtlety involved in synthesis of (parts of) relational programs. This is especially crucial to understand for readers familiar with program synthesis using the relational Scheme interpreter [Byrd et al., 2012] because certain intuitions do not carry over to the relational metaKanren interpreter, due to the fundamental difference between the semantics of the interpreted language (i.e. functional and relational).

Consider the following run of a one argument user-defined relation, which unifies its argument with `5`,

```
(run 1 (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== 5 f)))
         (five z)))
    x))
⟹ ((5))
```

We get the expected output. Now, let us try to synthesize parts of user-defined relation `five`, by providing the expected output:

```
(run 1 (e1 e2)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== ,e1 ,e2)))
         (five 5)))
    '((_.)))
⟹ ((() ()))
```

Intuition suggests that this must evaluate to `((5 f))` (i.e. `5` is synthesized for `e1` and `f` for `e2`), making the actual answer confusing. To understand what is going on, it is helpful to look at the following slightly modified example:

```
(run 1 (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
```

```
                        (== 7 7)))
          (five 'elephant)))
    x)
⟹  (((_.))
```

When the input-output examples are all ground (as in the synthesis example for `five`), metaKanren does not introduce extra constraints on `z` and gives a correct, yet spurious, answer. Indeed, if we were to ask for more answers (3 in this case), we would eventually get what we expect:

```
(run 4 (e1 e2)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== ,e1 ,e2)))
         (five 'elephant)))
    '((_.)))
⟹  ((() ()) ((_.0 _.0) (num _.0)) (#t #t) (5 f))
```

The solution to such a predicament is to ground the inner logic variable externally, a technique we call ***external grounding***:

```
(run 1 (e1 e2)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== ,e1 ,e2)))
         (five z)))
    '(5))
⟹  ((5 f))
```

The reason this solution works is when we ground `z` via the external query, it forces the metaKanren interpreter to introduce constraints on `z`, which was not the case when all the input-output examples were ground (which left `z` fresh).

Now, we get back to (partial) synthesis of `appendo` (changes from forward run example are highlighted):

```
(run 1  (x y w)
```

```
(symbolo x)
(symbolo y)
(symbolo w)
(eval-programo
  `(run* (z)
     (letrec-rel ((appendo (l1 l2 l)
                    (conde
                      [(== '() l1) (== l2 l)]
                      [(fresh (a d l3)
                         (== (cons a d) l1)
                         (== (cons a l3) l)
                         (delay (appendo ,x
                                         ,y
                                         ,w)))])))
       (appendo '(cat dog) '() '(cat dog))
       (appendo '(apple) '(peach) '(apple peach))
       (appendo '(1 2) '(3 4) z)))
   '((1 2 3 4)))
⟹ ((d l2 l3))
```

And we get what we expect!

In addition to external grounding, here we also:

1. provide two more examples as to how `appendo` should behave to avoid overfitting.

2. provide additional hints in the form of `symbolo` constraints to make the query return faster. Despite these extra hints, it takes the current naive interpreter about a minute to synthesize the arguments. Therefore, improving the interpreter's performance is crucial for any serious synthesis tasks (see Chapter 7).

4.2.3 Counting answers

Yet another interesting example is to place the logic variable in the run counter, so we get back the number of answers to a relation. For example (note the placement of the logic variable `count` instead of a Peano numeral in the inner run query):

```
(run* (count)
  (eval-programo
    `(run ,count (z)
       (conde [(== z 1)]
              [(== z 2)]))
    '(1 2)))
⟹ ((()) (((_.0)) (=/= ((_.0 ()))))))
```

As expected, the first answer says that there are exactly two assignments, and the second there are at least two assignments (`z = 1` and `z = 2` respectively). We get the second answer because the semantics dictate that any query that asks for more answers than maximum number of answers available just returns all the answers available.

We can go a step further and use another logic variable for the result of the inner query to get ordered pairs of counts and answers:

```
(run* (count answers)
  (eval-programo `(run ,count (z)
                   (conde [(== z 1)]
                          [(== z 2)]))
                 answers))
⟹ ((() ())
 ((()) (1))
 ((()()) (1 2))
 ((()(_.0)) (1 2)) (=/= ((_.0 ()))))))
```

This tells us that asking for 0 answers gives the empty list, 1 answer gives `(1)` and so on.

4.3 The metaKanren interpreter

This section is a commented tour of our interpreter for metaKanren. We assume familiarity with μKanren and relational interpreters. We direct readers unfamiliar with μKanren to Friedman [2013] and relational interpreters to Byrd et al. [2012]. Familiarity with both of these will greatly help in understanding this section as our implementation is inspired by mukanren.

The problems to be solved to implement metaKanren are:

1. Choosing a representation for logic variables that can be manipulated relationally.

2. Defunctionalizing goals.

3. Defunctionalizing streams.

4. Defunctionalizing user-defined relations.

We believe 1 is important, because the other current deep relational interpreters for miniKanren languages use solutions that are more complicated than is necessary (see Chapter 6).

Although defunctionalization has been previously applied to arrive at a first-order miniKanren representation [Rosenblatt et al., 2019], it has been only done with the interpreting language as Racket. In other words, for user-defined relations one must still piggyback onto Racket's `lambda` and perform application to use them. But for metaKanren, the interpreting language miniKanren has neither of these features, demanding a total defunctionalization (unlike defunctionalizing partially, and piggybacking onto host level features).

We break down our metaKanren relational interpreter into three relational sub-interpreters, one each for goal expressions, term expressions, and programs. Each one plays a specific well-defined role in correspondence with metaKanren's grammar and they call one another when needed, which makes the entire interpreter as a whole easier to understand and modify as required. Most of the code is included in the text, except for tedious parts which are in Appendix A.[2]

4.3.1 Logic variables

We represent logic variables as the `var` tag consed with a Peano numeral. For example, the very first logic variable is `(var)` (the value of the expression `(cons 'var '())`), which would have been `(vector 0)` in μKanren. We thus define relations for handling variables accordingly:

```
; Succeed if p is a peano numeral, fail otherwise
(define (peanoo p)
  (conde
    [(== '() p)]
    [(fresh (p1)
       (== `(,p1) p)
       (peanoo p1))]))

; Succeed if x is a logic variable, fail otherwise
(define (var?o x)
```

[2]Code for the entire metaKanren interpreter is available here https://github.com/iambrj/ms-thesis/blob/master/artifact/metaKanren.scm

```
  (fresh (val)
    (== `(var . ,val) x)
    (peanoo val)))

; Succeed if x and y are the same logic variable, fail otherwise
(define (var=?o x y)
  (fresh (val)
    (== `(var . ,val) x)
    (== `(var . ,val) y)
    (peanoo val)))

; Succeed if x and y are not the same logic variable, fail otherwise
(define (var=/=o x y)
  (fresh (val1 val2)
    (== `(var . ,val1) x)
    (== `(var . ,val2) y)
    (=/= val1 val2)
    (peanoo val1)
    (peanoo val2)))
```

4.3.2 Term expression interpreter

Term expressions are those that occur as arguments to == (see Section 5.1). The goal interpreter calls the term expression interpreter when it encounters a == goal expression to evaluate both the arguments. Terms may contain lexical variables (represented as symbols in the interpreting language), thus the term interpreter takes an environment (represented as an association list) for evaluating terms so that it may look up lexical variables in the environment. For atomic values (such as booleans and numbers) and quoted terms, the interpreter simply unifies them with the output logic variable as they are self evaluating. For lists, it evaluates the `car` and `cdr` of the list recursively.

```
(define (eval-texpro expr env val)
  (conde
    [(== expr val)
```

```
 (conde
   [(== '() expr)]
   [(booleano expr)]
   [(numbero expr)])]
[(== `(quote ,val) expr)
 (conde
   [(== '() val)]
   [(fresh (a d)
      (== `(,a . ,d) val))]
   [(booleano val)]
   [(numbero val)]
   [(symbolo val)])
 (not-in-envo 'quote env)
 (absento 'var val)
 (absento 'closr val)]
[(symbolo expr)
 (lookupo expr env val)]
[(fresh (e1 e2 v1 v2)
   (== `(cons ,e1 ,e2) expr)
   (== `(,v1 . ,v2) val)
   (not-in-envo 'cons env)
   (eval-texpro e1 env v1)
   (eval-texpro e2 env v2))]))
```

`not-in-envo` is a helper relation which only succeeds if a symbol does not occur in the environment, and `booleano` only succeeds if its argument is either `#t` or `#f`. Their implementations are provided in Appendix A.

4.3.3 States

Just as in μKanren, a state in metaKanren is represented as a list with two elements – a substitution and a counter (represented as a Peano numeral). We use association lists for substitutions and Peano numerals for counters. For example, the initial state is `(() ())`, which would have been `(() . 0)`

in μKanren. Goals are applied to states to generate streams. The goal interpreter (discussed in Section 4.3.6) performs the application.

The value of a logic variable (if any) is searched in the substitution of a state while performing unification via the `walk` operation. Terms other than logic variables `walk` to themselves.

```
; Unify the first cons pair of the substitution whose car is v with
; out
(define (assp-subo v s out)
  (fresh (h t h-a h-d)
    (== `(,h . ,t) s)
    (== `(,h-a . ,h-d) h)
    (var?o v)
    (var?o h-a)
    (conde
      [(== h-a v) (== h out)]
      [(=/= h-a v) (assp-subo v t out)])))

; Succeed if the substitution has no cons pair whose car is v, fail
; otherwise
(define (not-assp-subo v s)
  (fresh ()
    (var?o v)
    (conde
      [(== '() s)]
      [(fresh (t h-a h-d)
        (== `((,h-a . ,h-d) . ,t) s)
        (=/= h-a v)
        (var?o h-a)
        (not-assp-subo v t))])))

; Unify the value of the term u in the substitution s with v
(define (walko u s v)
```

```
  (conde
    [(== u v)
     (conde
       [(== '() u)]
       [(symbolo u)]
       [(numbero u)]
       [(booleano u)]
       [(fresh (a d)
          (== `(,a . ,d) u)
          (=/= a 'var))])]
    [(var?o u)
     (conde
       [(== u v) (not-assp-subo u s)]
       [(fresh (pr-d)
          (assp-subo u s `(,u . ,pr-d))
          (walko pr-d s v))])]))
```

Substitutions are extended by adding the logic variable and its value. We perform a check before extending to ensure there are no cycles in our substitutions.

```
(define (ext-so u v s s1)
  (fresh (occurs?)
    (occurso u v s occurs?)
    (conde
      [(== #t occurs?) (== #f s1)]
      [(== #f occurs?) (== `((,u . ,v) . ,s) s1)])))


(define (occurso x v-unwalked s occurs?)
  (fresh (v)
    (walko v-unwalked s v)
    (conde
      [(== '() v) (== #f occurs?)]
      [(symbolo v) (== #f occurs?)]
```

```
[(numbero v) (== #f occurs?)]
[(booleano v) (== #f occurs?)]
[(var?o v) (var=?o x v) (== #t occurs?)]
[(var?o v) (var=/=o x v) (== #f occurs?)]
[(fresh (av dv occurs-av? occurs-dv?)
   (== `(,av . ,dv) v)
   (=/= 'var av)
   (occurso x av s occurs-av?)
   (occurso x dv s occurs-dv?)
   (conde
     [(== #f occurs-av?) (== #f occurs-dv?) (== #f occurs?)]
     [(== #f occurs-av?) (== #t occurs-dv?) (== #t occurs?)]
     [(== #t occurs-av?) (== #f occurs-dv?) (== #t occurs?)]
     [(== #t occurs-av?) (== #t occurs-dv?) (== #t occurs?)]))]))))
```

4.3.4 Unification

Translating μKanren's `unify` into `unifyo`, its first-order relational counterpart is a straightforward but laborious process due to the number of cases involved. In metaKanren, each of the two terms in a unification is one of the following:

1. A logic variable.
2. An Arabic numeral.
3. A symbol.
4. A Boolean value.
5. The empty list.
6. A non-empty list of any of these.

Therefore, there are a total of 36 + 5 (when the types match, but the two terms are not equal) = 41 cases. Here we show just enough cases to shed light on how `unifyo` should behave, and defer the entire definition of `unifyo` to Appendix A.

```
; s1 is the extended substitution if unification of u-unwalked and
```

```
; v-unwalked in the substitution s succeeds, #f otherwise
(define (unifyo u-unwalked v-unwalked s s1)
  (fresh (u v)
    (walko u-unwalked s u)
    (walko v-unwalked s v)
    (conde
      [(var?o u) (var?o v) (var=?o u v) (== s s1)]
      ; Variables may extend the substitution
      [(var?o u) (var?o v) (var=/=o u v) (ext-so u v s s1)]
      ; Types other than number in appendix
      [(var?o u) (numbero v) (ext-so u v s s1)]
      ; Types and values of terms are equal
      [(numbero u) (numbero v) (== u v) (== s s1)]
      ; Types are equal but values are not, unification fails
      [(numbero u) (numbero v) (=/= u v) (== #f s1)]
      ; Types are not equal, unification fails
      [(numbero u) (symbolo v) (== #f s1)])))
```

4.3.5 Streams & search

μKanren uses procedures to represent immature streams and evaluates them to pull states out of the stream, which is not possible to do with metaKanren. Instead, we use lists with various tags and have `pullo`, a first-order relational translation of μKanren's `pull`, extract values from the stream (discussed below). The tag of the stream represents the action that is delayed so that it may be performed when pulling states from the stream. The structures of various streams are shown in Table 4.1. Details of various streams follow.

1. **Empty stream/mzero** : This is generated when a goal fails. For example, `(== 1 2)` generates `mzero` (irrespective of the state to which it is applied). We represent `mzero` using the empty list.

2. **Mature** : This is generated when a goal succeeds with at least one answer (possibly followed by a delayed stream). For example, `(conde [(== 1 1)] [(== 2 2)])` generates `(state state)`, where `state` is the state to which the goal is applied, since both branches of the disjunction succeed. Similarly, `(conde [(== 1 1)] [(delay (== 2 2))])` generates

Stream	Structure
Empty/mzero	`()`
Mature	`(state . stream)`
Delayed eval	`(delayed eval goal-expr state env)`
Delayed mplus	`(delayed mplus stream stream)`
Delayed bind	`(delayed bind stream goal-expr env)`

Table 4.1 Structures of various streams

```
((state delayed mplus (delayed eval (== 2 2) state env) env))
```

where `env` is the lexical environment to which the goal is applied. Note that the delayed goal generates a delayed stream that follows the result of the first goal.

3. **Delayed eval** : This is generated when a goal expression evaluation is delayed. For example, `(delay (== 1 1))` generates `(delayed eval (== 1 1) state env)`. It saves the goal expression, the state, and the lexical environment so that the evaluation may be performed later when pulling states.

4. **Delayed mplus**: This is generated when one of the goals in a disjunction generates a delayed stream. A `delayed mplus` stream saves both of its argument streams so that the `mplus` can be performed later when pulling states.

   For example, `(conde [(delay (== 1 1))] [(== 2 2)])` generates the stream

```
(delayed mplus (delayed eval (== 1 1) state env)
               state)
```

5. **Delayed bind**: This is generated when the first goal in a conjunction generates a delayed stream. As with `delayed mplus`, a `delayed-bind` stream saves all of its arguments so that the `bind` may be performed later when pulling states.

   For example, `(fresh () (delay (== 1 1)) (== 2 2))` generates the stream

```
(delayed bind (delayed eval (== 1 1) state env)
              (== 2 2)
              env)
```

We employ an interleaving search strategy similar to μKanren encoded using `mpluso` and `bindo`, the first-order relational counterparts to μKanren's `mplus` and `bind`. For empty and mature streams,

`mpluso` and `bindo` have a nearly identical implementation to μKanren. However, for any of the delayed streams, they simply delay it further by saving the appropriate parameters.

```
(define mzero '())

(define (mpluso $1 $2 $)
  (conde
    [(== '() $1) (== $2 $)]
    [(fresh ($1-a $1-d $1-d-$2)
       (== `(,$1-a . ,$1-d) $1)
       (== `(,$1-a . ,$1-d-$2) $)
       (=/= 'delayed $1-a)
       (mpluso $1-d $2 $1-d-$2))]
    [(fresh (d)
       (== `(delayed . ,d) $1)
       (== `(delayed mplus ,$1 ,$2) $))]))

(define (bindo $ ge env $1)
  (conde
    [(== '() $) (== mzero $1)]
    [(fresh ($1-a $1-d v-a v-d)
       (== `(,$1-a . ,$1-d) $)
       (=/= 'delayed $1-a)
       (eval-gexpro ge $1-a env v-a)
       (bindo $1-d ge env v-d)
       (mpluso v-a v-d $1))]
    [(fresh (d)
       (== `(delayed . ,d) $)
       (== `(delayed bind ,$ ,ge ,env) $1))]))
```

`pullo` is the key component in interleaving the search. If the input stream is either empty or a mature stream, it simply unifies it with the output stream. However, when it is passed a delayed stream, it performs the action appropriate for the delay tag. For example, if it were passed a `delayed eval`

stream, it will call the goal expression interpreter (see Section 4.3.6) on the saved parameters and recursively pull on the resulting stream.

```
(define (pullo $ $1)
  (conde
    [(== '() $) (== '() $1)]
    [(fresh (a d)
       (== `(,a . ,d) $)
       (== $ $1)
       (=/= 'delayed a))]
    [(fresh (th $-th)
       (== `(delayed . ,th) $)
       (eval-thunko $ $-th)
       (pullo $-th $1))]))

(define (eval-thunko th $)
  (conde
   [(fresh (gexpr st env)
      (== `(delayed eval ,gexpr ,st ,env) th)
      (eval-gexpro gexpr st env $))]
   [(fresh ($1 $2 $1e)
      (== `(delayed mplus ,$1 ,$2) th)
      (eval-thunko $1 $1e)
      (mpluso $2 $1e $))]
   [(fresh ($1 gexpr env $1e)
      (== `(delayed bind ,$1 ,gexpr ,env) th)
      (eval-thunko $1 $1e)
      (bindo $1e gexpr env $))]))
```

4.3.6 Goal interpreter

μKanren uses procedures to represent goals, which is again not possible to do with metaKanren since miniKanren does not have procedures. Instead, we use a list of quoted datum to represent a goal. For

example, we represent the goal `(== 2 3)` as the list of datum `(== 2 3)`. This also has the advantage of making goals inspectable and thus helps in debugging metaKanren programs. As in μKanren, goals are applied to a state and produce a stream of states, and the goal interpreter performs this application. However, the metaKanren interpreter also has to pass around a lexical environment to manage bindings introduced via `letrec-rel` and `fresh`.

Delayed goals produce a `delayed eval` stream with the parameters saved, so that the application may be performed later when required. Disjunctions and conjunctions are translated into the appropriate calls to `mpluso` and `bindo` respectively. The `fresh` goal is handled by extended the lexical environment by binding the interpreting logic variable to the current counter, and recursively evaluating the body in the extended environment with an incremented counter in the state. The == goal is handled by calling the term interpreter (see Section 4.3.2) to evaluate the two terms, and then unifying them (see Section 4.3.4).

To handle user-defined relations, the goal interpreter extends the environment by binding the identifier of the user-defined relation to a closure with the parameters, the right hand side and the enclosing environment and recursively evaluates the body of the `letrec-rel`. To apply a user-defined relation, it looks up the identifier in the environment to fetch a closure, and then recursively evaluates the body after extending the environment with the values for the parameters. The implementation of user-defined relations is tangential to the relational components of metaKanren, we refer the reader to **?** for a thorough exposition of implementing such features.

```
(define (eval-gexpro expr st env $)
  (conde
    [(fresh (ge)
       (== `(delay ,ge) expr)
       ; Maturation happens when pulling
       (== `(delayed eval ,ge ,st ,env) $))]
    [(fresh (te1 te2 v1 v2 S C S1)
       (== `(== ,te1 ,te2) expr)
       (== `(,S ,C) st)
       (eval-texpro te1 env v1)
       (eval-texpro te2 env v2)
       (conde
         [(== #f S1) (== '() $)]
         [(=/= #f S1) (== `((,S1 ,C)) $)])
       (unifyo v1 v2 S S1))]
```

```
    [(fresh (x* ge+)
       (== `(fresh ,x* . ,ge+) expr)
       (=/= '() ge+)
       (eval-fresho x* ge+ st env $))]
    [(fresh (ge+)
       (== `(conde . ,ge+) expr)
       (=/= '() ge+)
       (eval-condeo ge+ st env $))]
    [(fresh (b* c* ge+ renv)
       (== `(letrec-rel ,b* . ,ge+) expr)
       (ext-reco b* '() env renv)
       (eval-conjno ge+ st renv $))]
    [(fresh (id args params ge+ env1 ext-env vargs)
       (== `(,id . ,args) expr)
       (lookupo id env `(closr ,params ,env1 . ,ge+))
       (eval-argso args env vargs)
       (exto params vargs env1 ext-env)
       (eval-conjno ge+ st ext-env $))]))

(define (eval-condeo conjn-ge* st env $)
  (conde
    [(== '() conjn-ge*) (== '() $)]
    [(fresh (conjn-ge-a conjn-ge-d $-a $-d)
       (== `(,conjn-ge-a . ,conjn-ge-d) conjn-ge*)
       (eval-conjno conjn-ge-a st env $-a)
       (eval-condeo conjn-ge-d st env $-d)
       (mpluso $-a $-d $))]))

(define (eval-conjno ge+ st env $)
  (fresh (ge-a ge-d ge-a-$)
    (== `(,ge-a . ,ge-d) ge+)
```

```
    ; XXX : sensitive conjunction order
    ; Loops on forward run if order changed
    (eval-gexpro ge-a st env ge-a-$)
    (conde
      [(== '() ge-d)
       (== ge-a-$ $)]
      [(=/= '() ge-d)
       (fresh (ge-d-gexpr)
         (== `(fresh () . ,ge-d) ge-d-gexpr)
         (bindo ge-a-$ ge-d-gexpr env $))]))))

(define (eval-fresho x* ge+ st env $)
  (conde
    [(== '() x*)
     (eval-conjno ge+ st env $)]
    [(fresh (x-a x-d S C env1)
       (== `(,x-a . ,x-d) x*)
       (== `(,S ,C) st)
       (exto `(,x-a) `((var . ,C)) env env1)
       (eval-fresho x-d ge+ `(,S (,C)) env1 $))]))
```

`eval-args`, `exto` and `lookupo` are helper relations to evaluate the arguments during an application of a user-defined goal, extend a lexical environment and search for a binding in an environment respectively. These are provided in Appendix A.

4.3.7 Recovering reification

There is a close one-to-one correspondence between the macros and functions in μKanren and the relations in metaKanren used to implement the user interface.

`take-no` and `take-allo` try to extract the first n and all states from a stream respectively, by using `pullo` (see Section 4.3.5) to extract one state at a time.

```
(define (take-allo $ s/c*)
  (fresh ($1)
```

```
    (pullo $ $1)
    (conde
      [(== '() $1) (== '() s/c*)]
      [(fresh (a d-s/c* $d)
         (== `(,a . ,$d) $1)
         (== `(,a . ,d-s/c*) s/c*)
         (take-allo $d d-s/c*))])))

(define (take-no n $ s/c*)
  (conde
    [(== '() n) (== '() s/c*)]
    [(=/= '() n)
     (fresh ($1)
       (pullo $ $1)
       (conde
         [(== '() $1) (== '() s/c*)]
         [(fresh (n-1 d-s/c* a d)
            (== `(,a . ,d) $1)
            (== `(,n-1) n)
            (== `(,a . ,d-s/c*) s/c*)
            (take-no n-1 d d-s/c*))]))]))
```

`reifyo` refies each state in a list of states by passing the state's substitution to `reify-state/1st-var`, which reifies it with respect to the first logic variable.

```
(define (reifyo s/c* out)
    (conde
      [(== '() s/c*) (== '() out)]
      [(fresh (a d va vd)
         (== `(,a . ,d) s/c*)
         (== `(,va . ,vd) out)
         (reify-state/1st-varo a va)
         (reifyo d vd))]))
```

```
(define (reify-state/1st-varo s/c out)
  (fresh (s c v u)
    (== `(,s . ,c) s/c)
    (walk*o `(var . ()) s v)
    (reify-so v '() u)
    (walk*o v u out)))
```

`reify-so` is a helper relation to reify fresh logic variables in a substitution. One must be careful about the representation chosen for the reification of fresh logic variables as a careless representation may introduce cycles in the substitution. We use the symbol `'_.` consed with the Peano numeral for the length of the substitution. For example, the first fresh logic variable would be reified as `(_.)` (which would have been `'_.0` in μKanren). The helper relation `lengtho`, used to calculate the length of the substitution as a Peano numeral, and `walk*o` which is the same as `walko` except recurs when it comes across a list, are in Appendix A.

```
(define (reify-so v-unwalked s s1)
  (fresh (v)
    (walko v-unwalked s v)
    (conde
      [(var?o v)
       (fresh (len)
         (lengtho s len)
         (== `((,v . (_. . ,len)) . ,s) s1))]
      [(== s s1)
       (conde
         [(numbero v)]
         [(symbolo v)]
         [(booleano v)]
         [(== '() v)])]
      [(fresh (a d sa)
         (=/= 'var a)
         (== `(,a . ,d) v)
         (conde
```

```
          [(== '_. a)
           (== s s1)]
          [(=/= '_. a)
           (reify-so a s sa)
           (reify-so d sa s1)]))])))
```

4.3.8 Program interpreter

The program interpreter calls the goal interpreter by wrapping a `fresh` corresponding to the logic variable in the `run` query, pulls as many states as need from the resulting stream, and then reifies these pulled states. See Section 4.2 for examples that use `eval-programo`.

```
(define empty-s '())
(define peano-zero '())
(define init-env '())

(define (eval-programo expr out)
  (conde
    [(fresh (lvar gexpr $ s/c*)
       (symbolo lvar)
       (== `(run* (,lvar) ,gexpr) expr)
       (eval-gexpro `(fresh (,lvar) ,gexpr)
                    `(,empty-s . ,peano-zero)
                    init-env
                    $)
       (take-allo $ s/c*)
       (reifyo s/c* out))]
    [(fresh (n lvar gexpr $ s/c*)
       (symbolo lvar)
       (== `(run ,n (,lvar) ,gexpr) expr)
       (eval-gexpro `(fresh (,lvar) ,gexpr)
                    `(,empty-s . ,peano-zero)
                    init-env
```

```
            $)
(takeo n $ s/c*)
(reifyo s/c* out))]))
```

Chapter 5

Adding Type Constraints

In this chapter, we show how to extend metaKanren to handle the `numbero` and `symbolo` type constraints. These constraints are particularly useful for implementing relational interpreters. Furthermore, our relational interpreter for metaKanren itself uses these constraints. Thus, to make our interpreter metacircular, we have to implement these constraints. This chapter is based on Joshi and Byrd [2022].

5.1 Implemeting Type Constraints

The grammar for metaKanren goal expressions extended with these type constraints is as follows.

⟨*goal-expr*⟩ ::= `(==` ⟨*term-expr*⟩ ⟨*term-expr*⟩`)`
| `(symbolo` ⟨*term-expr*⟩`)`
| `(numbero` ⟨*term-expr*⟩`)`
| `(fresh (`⟨*id*⟩ ...`)` ⟨*goal-expr*⟩ ...`)`
| `(conde (`⟨*goal-expr*⟩ ...`)` ...`)`
| `(letrec-rel ([`⟨*id*⟩ `(` ⟨*id*⟩ ...`)` ⟨*goal-expr*⟩ ...`]` ...`)` ⟨*goal-expr*⟩ ...`)`
| `(delay` ⟨*goal-expr*⟩`)`

The rest of the non-terminals remain unchanged.
Implementing type constraints can be broken down into four steps.

1. Extend the state to include a type constraint store.

2. Define `numbero` and `symbolo` goal constructors that update the state.

3. Now we have two interactions to a deal with — between type and == constraints, and between type and disequality constraints. We need to:

(a) verify after each == constraint that no type constraints are violated, and possibly simplify the type constraint store.

(b) implement subsumption of disequality constraints by type constraints;

4. Update the reifier to display information from the type store.

5.1.1 Extending the State

Firstly, we extend the state to hold a type constraint store.

```
(define empty-T '())
(define empty-state `(,empty-S ,peano-zero ,empty-T))
```

The type constraint store is a list of constraints, where each constraint consists of two components.

1. The logic variable on which the constraint exists (in our implementation type constraints on constant terms such as numbers and symbols get immediately resolved and are never added to the constraint store).

2. A tag naming the constraint, which will be useful when multiple constraints are applied on the same variable and displaying the final answer.

Thus, for example if we were to apply the `numbero` constraint on a logic variable `x`, the generated constraint would be:

```
(x . num)
```

Here, `x` would be replaced by the actual logic variable it is represented by, and the symbol `num` is the tag used to represent the `numbero` constraint.

With the above understanding of the type constraint store, we can extend the goal expression interpreter `eval-gexpro` to handle `symbolo` and `numbero` constraints.

5.1.2 Defining the Constraints

Here we only elaborate on the `numbero` case, as the `symbolo` case follows by symmetry. To evaluate `numbero`, we first evaluate the term expression `te` it is applied to. Then we call the interpreter `eval-numbero` on the evaluated value of the term expression, `v`.

```
(define (eval-gexpro expr st env out)
  (conde
    [(fresh (te v)
       (== `(numbero ,te) expr)
```

```
      (eval-texpro te env v)
      (eval-numbero v st env out))]
   [(fresh (te v)
      (== `(symbolo ,te) expr)
      (eval-texpro te env v)
      (eval-symbolo v st env out))]
   ...)) ; Rest of the cases unchanged
```

Implementing `eval-numbero` can be broken down into four cases, corresponding to the value of the term expression `v`.

1. If `v` is a `cons` pair, `eval-numbero` fails.
2. If `v` is a constant, `eval-numbero` succeeds only if `v` is a numerical constant. There are no changes to the constraint store in this case.
3. If `v` is a variable, then there are three cases. If `v` has no constraints on it so far, `eval-numbero` extends the constraint store to add a `num` constraint on `v`. If `v` already has a `num` constraint on it, `eval-numbero` does not modify the constraint store. Otherwise, if `v` has a conflicting `sym` constraint on it, `eval-numbero` fails.

The following code implements the above.

```
(define (eval-numbero v-unwalked st env out)
  (fresh (v S C T)
    (== `(,S ,C ,T) st)
    (walko v-unwalked S v)
    (conde
     [(== '() out)
      (conde
        [(== '() v)]
        [(booleano v)]
        [(symbolo v)]
        [(fresh (a d)
           (== (cons a d) v)
           (=/= 'var a))])]
     [(numbero v)
```

```
     (== `(,st) out)]
    [(fresh (p ext)
       (== `(,S ,C ,T) st)
       (== (cons 'var p) v)
       (ext-To v T 'num ext)
       (conde
         [(== #f ext) (== '() out)]
         [(== '() ext) (== `(,st) out)]
         [(== `(,v . num) ext) (== `((,S ,C (,ext . ,T))) out)]))]))))
```

Here, the auxiliary relation `ext-To` handles the variable case. `ext-To` goes through each constraint in the constraint store to check for three of the variable cases.

```
(define (ext-To x T tag ext)
  (conde
    [(== '() T) (== `(,x . ,tag) ext)]
    [(fresh (v vtag d)
       (== `((,v . ,vtag) . ,d) T)
       (conde
         [(== v x)
          (conde
            [(== tag vtag) (== '() ext)]
            [(=/= tag vtag)
             (== #f ext)])]
         [(=/= v x)
          (ext-To x d tag ext)]))]))
```

Now that we have defined the type constraints, we next have to deal with the interplay between the type constraints and the == constraint.

5.1.3 Unification and Type Constraints

Not only do we have to check that unifications do not violate type constraints, but also possibly simplify type constraint store. For example, consider the following program fragment:

```
(fresh (x)
```

```
  (symbolo x)
  (== 5 x))
```

The unification here breaks the `symbolo` constraint on `x`. Furthermore, unifications may also simplify the constraint store. For instance:

```
(fresh (x)
  (numbero x)
  (== 10 x))
```

After unification, we may discard the `numbero` constraint.

To implement these two, we go through each type constraint and check for both the cases — whether the new unification broke the type constraint or whether the type constraint may be discarded. If neither, we return false to indicate the constraint is retained as is.

```
(define (reform-To T S Tˆ)
  (conde
    [(== '() T) (== '() Tˆ)]
    [(fresh (u-unwalked u tag Td Tdˆ)
       (== (cons (cons u-unwalked tag) Td) T)
       (walko u-unwalked S u)
       (reform-To Td S Tdˆ)
       (conde
         [(var?o u)
          (fresh (ext)
            (ext-To u Tdˆ tag ext)
            (conde
              [(== '() ext) (== Tdˆ Tˆ)]
              [(fresh (a d)
                 (== (cons a d) ext)
                 (== `(,ext . ,Tdˆ) Tˆ))]
              [(== #f ext) (== #f Tˆ)]))]
         [(== 'num tag)
          (conde
            [(== Tdˆ Tˆ)
```

```
     (numbero u)]
    [(conde
       [(== '() u)]
       [(booleano u)]
       [(symbolo u)]
       [(fresh (a d)
          (== `(,a . ,d) u)
          (=/= a 'var))])
     (== #f T^)])]
[(== 'sym tag)
 (conde
   [(== Td^ T^)
    (symbolo u)]
   [(conde
      [(== '() u)]
      [(booleano u)]
      [(numbero u)]
      [(fresh (a d)
         (== `(,a . ,d) u)
         (=/= a 'var))])
    (== #f T^)])]
[(== #f Td^) (== #f T^)]))]))
```

5.1.4 Reification

We filter out constraints not relevant to the final answer during reification. For reifying type constraints, we require the following properties.

1. Every semantically equivalent program should produce the same answer, irrespective of the constraint order in the program.

2. Logic variables with the same constraint must be grouped together.

3. Irrelevant and redundant constraints must not be reified in the final answer.

To understand the first property, consider the following programs.

```
(run 1 (x) (fresh (a b)
             (== (cons a b) x)
             (numbero a)
             (numbero b)))
⟹ (((_.0 . _.1) (num _.0 _.1)))
```

Now consider the following semantically equivalent program, where the order of the `numbero` constraint is reversed.

```
(run 1 (x) (fresh (a b)
             (== (cons a b) x)
             (numbero b)
             (numbero a)))
⟹ (((_.0 . _.1) (num _.0 _.1)))
```

Observe that both the programs produce literally the same answer. We insist upon this property because of two reasons. Firstly, this property enables a convenient specification of synthesis problems. If the order of constraints in the output was dependent on the order in which they were written, to specify a synthesis problem we would have to predict the constraint order in the answer for the program we wish to synthesize. The other reason we insist on this property is to cut down on the search space – otherwise the interpreting miniKanren will synthesize vacuous metaKanren programs by merely changing the constraint order. The same reasons apply for insisting the second property.

For the third property, observe that any type constraint on a fresh logic variable not in the final answer may be discarded as the type constraint can always be satisfied by assigning the fresh logic variable a value satisfying the constraint. We use the auxiliary `purify-To` relation for this.

```
(define (purify-To T r T^)
  (conde
    [(== '() T) (== '() T^)]
    [(fresh (u-unwalked tag u Ta Td purified-Td)
       (== (cons Ta Td) T)
       (== (cons u-unwalked tag) Ta)
       (walko u-unwalked r u)
       (purify-To Td r purified-Td)
       (conde
```

```
      [(var?o u) (== purified-Td T^)]
      [(fresh (p)
         (== (cons '_. p) u)
         (peanoo p))
       (== (cons Ta purified-Td) T^)]))]))
```

We next incorporate the auxiliary `purify-To` into the reifier.

```
(define (reify-state/1st-varo st out)
  (fresh (S C T v reified-v reified-S impure-T)
    (== `(,S ,C ,impure-T) st)
    (walk*o `(var . ()) S v)
    (build-reify-S v '() reified-S)
    (walk*o v reified-S reified-v)
    (purify-To impure-T reified-S T)
    (prettifyo reified-v reified-S T out)))
```

To implement grouping, we repeatedly group all elements having the same constraint tag as the first element in the constraint store until there are no more elements in the constraint store.

```
(define (group-tag tag T tagged-T)
  (conde
    [(== '() T) (== '() tagged-T)]
    [(fresh (u-tag u Td tagged-Td)
       (== `(((var . ,u) . ,u-tag) . ,Td) T)
       (conde
         [(== u-tag tag) (== `(,u . ,tagged-Td) tagged-T)]
         [(=/= u-tag tag) (== tagged-Td tagged-T)])
       (group-tag tag Td tagged-Td))]))
```

Finally, we fill the last piece of the reifier – the auxiliary `prettifyo` relation. This relation implements the first property discussed above by lexicographically sorting the lists of variables with constraints on them. The insertion sort relation used is provided in Appendix A.

```
(define (prettifyo v reified-S reified-T out)
  (fresh (symT numT peano-symT peano-numT unreified-symT
```

```
    unreified-numT unsorted-symT unsorted-numT)
    (group-tag 'num reified-T unsorted-numT)
    (group-tag 'sym reified-T unsorted-symT)
    (insertion-sort-peano-list unsorted-symT unreified-symT)
    (insertion-sort-peano-list unsorted-numT unreified-numT)
    (walk*o unreified-symT reified-S peano-symT)
    (walk*o unreified-numT reified-S peano-numT)
    (insert-_. peano-numT numT)
    (insert-_. peano-symT symT)
    (conde
      [(== '() symT) (== '() numT) (== `(,v) out)]
      [(=/= '() symT) (== '() numT) (== `(,v (sym . ,symT)) out)]
      [(== '() symT) (=/= '() numT) (== `(,v (num . ,numT)) out)]
      [(=/= '() symT) (=/= '() numT)
       (== `(,v (num . ,numT) (sym . ,symT)) out)])))
```

5.2 Program Synthesis Examples

Let's start with a simple example, with the relational interpreter running in the forward direction.

```
(run* (out)
  (eval-programo `(run* (q)
                    (fresh (x)
                      (== x q)
                      (numbero x)))
                 out))
⟹ ((((_.) (num (_.)))))
```

Once again, it must be noted that we have a `run*` running inside a `run*`, which is only possible with a deep embedding.

Now let us look a more complicated example. Consider the following Scheme procedure, which flattens a nested list.

```
(define flatten
```

```
(lambda (xs)
  (cond [(null? xs) '()]
        [(pair? (car xs))
         (append (flatten (car xs)) (flatten (cdr xs)))]
        [else
         (cons (car xs) (flatten (cdr xs)))])))
```

To see the procedure in action, consider the following example.

```
(flatten '((a) b (c d)))
⟹ '(a b c d)
```

Assuming the only atomic values are symbols, the relational version of the above program would be as follows.

```
(define flatteno
  (lambda (xs xs^)
    (conde [(== '() xs) (== '() xs^)]
           [(fresh (a aa ad d a^ d^)
              (== (cons a d) xs)
              (== (cons aa ad) a)
              (flatteno a a^)
              (flatteno d d^)
              (appendo a^ d^ xs^))]
          [(fresh (a d d^)
             (== (cons a d) xs)
             (symbolo a)
             (flatteno d d^)
             (== (cons a d^) xs^))])))
```

To see the relation in action, consider the previous example again.

```
(run* (out) (flatteno '((a) b ((c d))) out))
⟹ ((a b c d))
```

Now let us try to synthesize the type constraint via an input output example. Firstly, we have to rewrite the above program to follow metaKanren's syntax. The above example would then look as follows.

```
(run 1 (val) (eval-programo
               `(run* (out)
                      (letrec-rel ([appendo (l s out)
                                            (conde
                                              [(== '() l) (== s out)]
                                              [(fresh (a d res)
                                                 (== (cons a d) l)
                                                 (== (cons a res) out)
                                                 (delay (appendo d s res)))])])
                                  (letrec-rel ([flatteno (xs xsˆ)
                                                         (conde
                                                           [(== '() xs) (== '() xsˆ)]
                                                           [(fresh (a aa ad d aˆ dˆ)
                                                              (== (cons a d) xs)
                                                              (== (cons aa ad) a)
                                                              (delay (flatteno a aˆ))
                                                              (delay (flatteno d dˆ))
                                                              (delay (appendo aˆ dˆ xsˆ)))]
                                                           [(fresh (a d dˆ)
                                                              (== (cons a d) xs)
                                                              (symbolo a)
                                                              (delay (flatteno d dˆ))
                                                              (== (cons a dˆ) xsˆ))])])
                                    (flatteno '((a) b ((c d))) out))))
               val))
⟹ ((((a b c d))))
```

Now let us try to synthesize the argument the `symbolo` constraint.

```
(run 1 (synth) (eval-programo
                 `(run* (out)
                        (letrec-rel ([appendo (l s out)
```

```
                                    (conde
                                      [(== '() l) (== s out)]
                                      [(fresh (a d res)
                                         (== (cons a d) l)
                                         (== (cons a res) out)
                                         (delay (appendo d s res)))])])
                            (letrec-rel ([flatteno (xs xsˆ)
                                          (conde
                                            [(== '() xs) (== '() xsˆ)]
                                            [(fresh (a aa ad d aˆ dˆ)
                                               (== (cons a d) xs)
                                               (== (cons aa ad) a)
                                               (delay (flatteno a aˆ))
                                               (delay (flatteno d dˆ))
                                               (delay (appendo aˆ dˆ xsˆ)))]
                                            [(fresh (a d dˆ)
                                               (== (cons a d) xs)
                                               (symbolo ,synth)
                                               (delay (flatteno d dˆ))
                                               (== (cons a dˆ) xsˆ))])])
                             (flatteno '((a) b ((c d))) out))))
                  '(((a b c d)))))
⟹ (a)
```

In the above program, we are trying to synthesize the argument to the `symbolo` constraint, given the input output example.

Chapter 6

Related Work

We know of two other deep relational interpreters for miniKanren languages [Hemann, 2014, Ballantyne, 2016]. Hemann [2014] implements μKanren extended to handle user-defined and delayed goals using a deep relational interpreter. They have a more complicated implementation as they make use of the purely relational arithmetic system described in Kiselyov et al. [2008] to implement logic variables. Furthermore, they do not implement a reifier or a `run N` or `run*` interface so, for instance, they cannot directly run any of the examples presented in this dissertation.

On the other hand, Ballantyne [2016] uses non-relational features to implement logic variables, making it impossible to make the interpreter metacircular by extending the interpreted language without breaking relationality. Furthermore, they do not implement a `run*` for the interpreted language which makes it difficult to express failing programs as discussed in Chapter 3. It is also worth noting that their interpreter is continuation passing rather than stream passing like ours.

Additionally, another attempt at a first-order implementation of miniKanren is by Rosenblatt et al. [2019]. The motivation here was to have a representation of miniKanren that is transparent and supports debugging, as opposed to most miniKanren implementations which tend to use non-transparent objects (e.g. closures) to implement goals.

Chapter 7

Conclusions and Future Work

There are two directions of future work evident from our experiments with metaKanren. We elaborate on both of these directions in this chapter.

7.1 Towards full metacircularity

As of now, metaKanren uses two constraints that it does not implement, namely the disequality constraint `=/=` and `absento`. To make our relational interpreter fully metacircular, both of these must be implemented. As promised in the introduction, we will now describe precisely the problem with implementing these constraints. The techniques used to implement the `symbolo` and `numbero` type constraints (see Chapter 5) cannot be extended to implement the `=/=` and `absento` constraints. This is because of an important property we require of our metaKanren implementation – semantically equivalent programs should produce literally the same answers upon termination irrespective of the order in which constraints occur in the program. For example, consider the following miniKanren program.

```
(run 1 (x) (=/= 'bat x) (=/= 'cat x))
⟹ ((_.0 (=/= ((_.0 bat)) ((_.0 cat)))))
```

Now consider the following semantically equivalent program, where the order of the `=/=` constraint is reversed.

```
(run 1 (x) (=/= 'cat x) (=/= 'bat x))
⟹ ((_.0 (=/= ((_.0 bat)) ((_.0 cat)))))
```

Recall from Section 5.1.4 that for type constraints, we implement this by sorting the peano numerals of the variables with type constraints. However, in the case of disequality (and `absento`) constraints, the previous strategy cannot be used as we would need a lexicographical comparison for arbitrary strings to perform sorting on strings. To understand why, consider the above disequality example again. Here, the reified disequality store is `(=/= ((_.0 bat)) ((_.0 cat)))`. To decide if `(_.0 bat)`

constraint or `(_.0 cat)` constraint comes first, we would have to lexicographically compare the symbols `bat` and `cat`. But such a comparison is not available in miniKanren[1].

7.2 Optimizing metaKanren for synthesis

As mentioned in synthesis example 4.2.2, the current interpreter takes about a minute to synthesize something as simple as the arguments to a recursive call. This means that to perform synthesis of any interesting miniKanren programs, a significant improvement in performance is needed. Three particular strategies that can be employed to improve metaKanren's performance are:

1. **Staging**: The current tower of interpreters is shown in Table 7.1. The interpretive overhead can be removed by transforming the interpreter into a single-pass compiler by using staging techniques Amin and Rompf [2017].

metaKanren runs on ...
miniKanren runs on ...
Scheme

Table 7.1 Tower of interpreters

2. **Barliman techniques**: Several techniques have been developed and incorporated into the Barliman smart editor [Byrd and Rosenblatt, 2016] to perform real-time program synthesis of programs as complex as `append` in a relational Scheme interpreter. Some of these techniques can be useful to speed up the metaKanren interpreter as well.

3. **Guiding the search**: We have used faster-miniKanren Ballantyne [2022] to run all our metaKanren experiments. Using a guided search strategy, such as the one described by Zhang et al. [2018], can also improve performance.

[1]While such comparison could be done by integrating miniKanren with an external solver or miniKanren extended to handle more constraint solving , recall that in Chapter 1 we specify the exact version referred to by "miniKanren". We refer to this specific version of miniKanren in this claim. Furthermore, adding such constraint solving would also make reaching full metacircularity harder.

Appendix A

Auxiliary Code

This appendix contains (helper) relations/procedures used but not (entirely) provided.

```
(define (peano n)
  (if (zero? n) '() `(,(peano (- n 1)))))

(define (booleano b)
  (conde
    [(== #t b)]
    [(== #f b)]))

; u, v <- {logic var, number, symbol, boolean, empty list, non-empty list}
; Total 36 + 5 (types match, but terms do not) = 41 cases
(define (unifyo u-unwalked v-unwalked s s1)
  (fresh (u v)
    (walko u-unwalked s u)
    (walko v-unwalked s v)
    (conde
      [(var?o u) (var?o v) (var=?o u v) (== s s1)]
      [(var?o u) (var?o v) (var=/=o u v) (ext-so u v s s1)]
      [(var?o u) (numbero v) (ext-so u v s s1)]
      [(var?o u) (symbolo v) (ext-so u v s s1)]
      [(var?o u) (booleano v) (ext-so u v s s1)]
      [(var?o u) (== '() v) (ext-so u v s s1)]
```

```
[(var?o u)
 (fresh (a d)
   (== `(,a . ,d) v)
   (=/= 'var a))
 (ext-so u v s s1)]
[(numbero u) (var?o v) (ext-so v u s s1)]
[(numbero u) (numbero v) (== u v) (== s s1)]
[(numbero u) (numbero v) (=/= u v) (== #f s1)]
[(numbero u) (symbolo v) (== #f s1)]
[(numbero u) (booleano v) (== #f s1)]
[(numbero u) (== '() v) (== #f s1)]
[(numbero u)
 (fresh (a d)
   (== `(,a . ,d) v)
   (=/= 'var a))
 (== #f s1)]
[(symbolo u) (var?o v) (ext-so v u s s1)]
[(symbolo u) (numbero v) (== #f s1)]
[(symbolo u) (symbolo v) (== u v) (== s s1)]
[(symbolo u) (symbolo v) (=/= u v) (== #f s1)]
[(symbolo u) (booleano v) (== #f s1)]
[(symbolo u) (== '() v) (== #f s1)]
[(symbolo u)
 (fresh (a d)
   (== `(,a . ,d) v)
   (=/= 'var a))
 (== #f s1)]
[(booleano u) (var?o v) (ext-so v u s s1)]
[(booleano u) (numbero v) (== #f s1)]
[(booleano u) (symbolo v) (== #f s1)]
[(booleano u) (booleano v) (== u v) (== s s1)]
```

```
[(booleano u) (booleano v) (=/= u v) (== #f s1)]
[(booleano u) (== '() v) (== #f s1)]
[(booleano u)
 (fresh (a d)
   (== `(,a . ,d) v)
   (=/= 'var a))
 (== #f s1)]
[(== '() u) (var?o v) (ext-so v u s s1)]
[(== '() u) (numbero v) (== #f s1)]
[(== '() u) (symbolo v) (== #f s1)]
[(== '() u) (booleano v) (== #f s1)]
[(== '() u) (== '() v) (== s s1)]
[(== '() u)
 (fresh (a d)
   (== `(,a . ,d) v)
   (=/= 'var a))
 (== #f s1)]
[(var?o v)
 (fresh (a d)
   (== `(,a . ,d) u)
   (=/= 'var a))
 (ext-so v u s s1)]
[(numbero v)
 (fresh (a d)
   (== `(,a . ,d) u)
   (=/= 'var a))
 (== #f s1)]
[(symbolo v)
 (fresh (a d)
   (== `(,a . ,d) u)
   (=/= 'var a))
```

```
       (== #f s1)]
      [(booleano v)
       (fresh (a d)
         (== `(,a . ,d) u)
         (=/= 'var a))
       (== #f s1)]
      [(== '() v)
       (fresh (a d)
         (== `(,a . ,d) u)
         (=/= 'var a))
       (== #f s1)]
      [(fresh (u-a u-d v-a v-d s-a)
         (== `(,u-a . ,u-d) u)
         (== `(,v-a . ,v-d) v)
         (=/= 'var u-a)
         (=/= 'var v-a)
         (conde
           [(== s-a #f) (== #f s1) (unifyo u-a v-a s s-a)]
           [(=/= s-a #f)
            (unifyo u-a v-a s s-a)
            (unifyo u-d v-d s-a s1)]))])))

(define (exto params args env env1)
  (conde
    [(== params '())
     (== args '())
     (== env env1)]
    [(fresh (x-a x-d v-a v-d)
       (== `(,x-a . ,x-d) params)
       (== `(,v-a . ,v-d) args)
       (exto x-d v-d `((,x-a . ,v-a) . ,env) env1))]))
```

```
(define (lookupo x env v)
  (conde
    [(fresh (y u env1)
      (== `((,y . ,u) . ,env1) env)
      (=/= y 'rec)
      (conde
        [(== x y) (== v u)]
        [(=/= x y) (lookupo x env1 v)]))]
    [(fresh (id params geb env1)
        (== `((rec (,id ,params ,geb)) . ,env1) env)
        (conde
          [(== id x)
           (== `(closr ,params ,geb ,env) v)]
          [(=/= id x)
           (lookupo x env1 v)]))]))

(define (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y v env1)
       (== `((,y . ,v) . ,env1) env)
       (=/= x y)
       (not-in-envo x env1))]))

(define (eval-args args env vals)
  (conde
    [(== args '())
     (== vals '())]
    [(fresh (a d va vd)
       (== `(,a . ,d) args)
```

```
       (== `(,va . ,vd) vals)
       (eval-texpro a env va)
       (eval-args d env vd))]))

(define (walk*o unwalked-v s u)
  (fresh (v)
    (walko unwalked-v s v)
    (conde
      [(== v u)
       (conde
         [(var?o v)]
         [(numbero v)]
         [(symbolo v)]
         [(booleano v)]
         [(== '() v)])]
      [(fresh (a d walk*-a walk*-d)
         (== `(,a . ,d) v)
         (=/= a 'var)
         (conde
           [(== '_. a)
            (== u v)]
           [(=/= '_. a)
            (== `(,walk*-a . ,walk*-d) u)
            (walk*o a s walk*-a)
            (walk*o d s walk*-d)]))])))

(define (lengtho l len)
  (conde
    [(== '() l) (== '() len)]
    [(fresh (a d len-d)
       (== `(,a . ,d) l)
```

```
       (== `(,len-d) len)
       (lengtho d len-d))]))

(define (insertion-sort-peano-list L sorted-L)
  (conde
    [(== '() L) (== '() sorted-L)]
    [(fresh (a d sorted-d)
       (== (cons a d) L)
       (insertion-sort-peano-list d sorted-d)
       (insert-into sorted-d a sorted-L))]))

(define (insert-into sorted-L u inserted-L)
  (conde
    [(== '() sorted-L) (== `(,u) inserted-L)]
    [(fresh (a d <?)
       (== (cons a d) sorted-L)
       (peano< a u <?)
       (conde
         [(== #t <?)
          (fresh (inserted-d)
            (insert-into d u inserted-d)
            (== (cons a inserted-d) inserted-L))]
         [(== #f <?)
          (== (cons u sorted-L) inserted-L)]))]))

(define (peano< p1 p2 <?)
  (fresh ()
    (peanoo p1)
    (peanoo p2)
    (conde
      [(== p1 p2) (== #f <?)]
```

```
[(== '() p1) (=/= '() p2) (== #t <?)]
[(=/= '() p1) (== '() p2) (== #f <?)]
[(fresh (d1 d2)
   (== `(,d1) p1)
   (== `(,d2) p2)
   (peano< d1 d2 <?))])))
```

Related Publications

1. Bharathi Ramana Joshi, and William E. Byrd. “metaKanren: Towards a Metacircular Relational Interpreter” *Relational Programming Workshop. 2021*

2. Bharathi Ramana Joshi, and William E. Byrd. “An Annotated Implementation of miniKanren With Constraints” *Relational Programming Workshop. 2022*

Bibliography

Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158140. URL https://doi.org/10.1145/3158140.

Michael Ballantyne. meta-minikanren. https://github.com/michaelballantyne/meta-minikanren, 2016.

Michael Ballantyne. meta-minikanren. https://github.com/michaelballantyne/faster-minikanren, 2022.

William E. Byrd. *Relational Programming in Minikanren: Techniques, Applications, and Implementations*. PhD thesis, USA, 2009. AAI3380156.

William E. Byrd and Greg Rosenblatt. Barliman. https://github.com/webyrd/Barliman, 2016.

William E. Byrd, Eric Holk, and Daniel P. Friedman. Minikanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, page 8–29, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450318952. doi: 10.1145/2661103.2661105. URL https://doi.org/10.1145/2661103.2661105.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017. doi: 10.1145/3110252. URL https://doi.org/10.1145/3110252.

D.P. Friedman, W.E. Byrd, O. Kiselyov, J. Hemann, and D. Bibby. *The Reasoned Schemer, second edition*. MIT Press, 2018. ISBN 9780262535519. URL https://books.google.co.in/books?id=ur34DwAAQBAJ.

Jason Hemann Daniel P Friedman. μkanren: A minimal functional core for relational programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming. http://webyrd. net/scheme-2013/papers/HemannMuKanren2013. pdf*, 2013.

Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). *SIGPLAN Not.*, 49(9):339–347, aug 2014. ISSN 0362-1340. doi: 10.1145/2692915.2628138. URL https://doi.org/10.1145/2692915.2628138.

Jason Hemann. micro-in-mini. https://github.com/jasonhemann/micro-in-mini/, 2014.

Jason Hemann and Daniel P. Friedman. Some novel minikanren synthesis tasks. In *Proceedings of the miniKanren and Relational Programming Workshop*, miniKanren '20. Association for Computing Machinery, 2020.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 111–119, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912152. doi: 10.1145/41625.41635. URL https://doi.org/10.1145/41625.41635.

Bharathi Ramana Joshi and William E. Byrd. An annotated implementation of minikanren with constraints. 2022.

Oleg Kiselyov, William Byrd, Daniel Friedman, and Chung-chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). pages 64–80, 12 2008. ISBN 978-3-540-78968-0. doi: 10.1007/978-3-540-78969-7_7.

Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. First-order minikanren representation: Great for tooling and search. *1 Towards a miniKanren with fair search strategies by Lu, Ma & Friedman 1 2 First-order miniKanren representation by Rosenblatt, Zhang, Byrd & Might 16 3 Relational Interpreters for Search Problems by Lozov, Verbitskaia & Boulytchev 43 4 Constructive Negation for miniKanren by Moiseenko 58 5 Certified Semantics for miniKanren by Rozplokhas, Vyatkin & Boulytchev 80*, page 16, 2019.

Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/67d16d00201083a2b118dd5128dd6f59-Paper.pdf.