Khoa Vo

RELATIONAL CONDITIONALS WITH PSEUDO-FUNCTIONS

Techniques in miniKanren

Bachelor's thesis

Degree programme in Information Technology

2019

XAMK

South-Eastern Finland University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Author	**Degree**	**Time**
Khoa Vo	Bachelor of Engineering	April 2019
Thesis title	**Pages**	**Appendices**
Relational conditionals with pseudo-functions	37	0

Commissioned by

Self initiative

Supervisor

Timo Hynninen

Abstract

Relational programming promises the ability to run programs backwards. Using miniKanren, it is very easy to convert pure functions into relations. And because of its embedded nature, programmers can enjoy the declarative power of miniKanren without leaving the comfort of their favorite programming languages. Even though this paper chooses to discuss the Scheme implementation of the language, the techniques presented here is general for all implementations.

Some miniKanren users prefer to first write a prototype in a functional manner in Scheme, and later transform that prototype into its relational counterpart. Despite being an easy task, this transformation frequently increases the code size. As programs grow longer, there are more chances for human errors to creep in.

One reason for this size increase (and why some people want to write programs functionally in the first place) has to do with the symbiosis between Boolean-returning functions and conditionals. Branching has long been a powerful programming tool, yet they have no adequate counterpart in miniKanren. `conda` and `condu` cannot be used relationally as they are logically impure. And even though new users are guided to convert all `cond` expressions to `conde`, `conde` clauses do not automatically deny preceding tests, a job which is then delegated to the one performing the conversion. As a result, many programs lose their compactness and structural clarity after the conversion.

This thesis aims to provide frameworks and techniques for writing conditionals in miniKanren as well as a way to analyze the correctness of pre-existing miniKanren programs using conditionals. As for the former, the goal constructor `condo` along with test combinators `conjt` and `disjt` are introduced. Concerning the latter, a more involved method of static analysis and rewriting is used, resulting in a new language variant called staticKanren.

Keywords

programming, logic, Scheme, miniKanren

CONTENTS

1 INTRODUCTION

Many problems, especially logic puzzles such as ”knights and knaves” (Smullyan 1978) put heavy emphasis on enumeration and search. Programs to solve these problems can involve different techniques, generalized under the term **backtracking**. Logic programming offer to make backtracking implicit; programmers only need to declare the rules, and the computer will automatically generate all states consistent within those rules. Logic languages often give elegant solutions to a wide class of problems, some can be also used in general computation.

Logic programming dates back to 1972, when Prolog (**pro**grammation en **log**ique) was first invented by Alain Colmerauer and Phillipe Roussel (Kowalski 1988). At the time of its birth, logic programming was considered by some to be the future, especially in Japan. In the 1970’s, the Japanese International Trade and Industry (MITI) wanted to take over the computer industry with a new Fifth Generation Computing System Project (named ICOT), aiming to replace conventional algorithmic computing with constraint-based programming techniques. The project soon failed, however, because these languages were not fast enough to compete with other mainstream object-oriented languages such as Java and C#. Carl Hewitt summed up the situation by the comment ”*Computation is not subsumed by deduction.*” (Hewitt 2008.)

Despite its name, Prolog was not just about logic and its users generally do not shy away from its extra-logical features. These features on one hand can greatly enhance performance, but they can also make programs less declarative, incomplete and even unsound. On the other hand, **relational programming** is a discipline of logic programming in which these features are left out. This guarantees that all answers are returned even when all arguments are logic variables. Additionally, the same set of answers are returned regardless of the order in which rules are applied. The design philosophy of **miniKanren** puts great emphasis on this discipline[1]. (Byrd 2009.)

[1]”Kanren” is literally Japanese for ”relation”.

Technically speaking, miniKanren is not a language with its own compiler or interpreter. It is instead a family of languages embedded in a great variety of other host languages. This paper uses the Scheme implementation since it is the canonical version mentioned in almost every research paper written on the language. Also, Scheme's macro makes the syntax less clunky and more natural to read and write, which helps keep the paper short.

This thesis aims to provide frameworks and techniques for writing conditionals in miniKanren as well as a way to analyze the correctness of many pre-existing miniKanren programs using conditionals.

The layout of this paper is as follow: Section 2 gives a brief introduction to the syntax and semantics of miniKanren. Section 3 defines conditionals using pseudo-functions and demonstrates their use in miniKanren. Section 4 introduces staticKanren, its implementation and its applications related to conditionals.

2 PRELIMINARIES

This section gives a brief overview to miniKanren. A complete introduction to the language is presented in Friedman et al. (2018). Byrd (2009) gives an excellent dissertation on the implementation details. For a more focused and concise treatment of relational programming in Scheme, please refer to microKanren (Hemann & Friedman 2013). Additionally, the website `http://minikanren.org/` keeps up with the latest implementations of the language and other related resources.

As mentioned earlier miniKanren is not a standalone programming language but an embedded structure inside of a language (in our case, Scheme). The language components from the low-level perspective include:

- **Logic variables**: These are very different from naming variables. Unbound naming variables always mean that an error has occurred. In contrast, unbound logic variables are perfectly valid objects. From here on, variable means logic variable unless stated otherwise.

- **Streams**: Streams (Abelson et al. 1996) are simply lazy lists that will yield values only when we ask for them. Because of this laziness, we can represent an infinite number of answers with finite computational resources.

- **States** (also **packages**): States are the most important ingredient, representing all logical assertions in the program at the moment.

- **Goal**: A goal is a function mapping a state to a stream of states. Goals should only be created by **goal constructors**, as will be discussed soon.

- **Relation**: Relations are simply Scheme functions that return goals.

- **Program**: A miniKanren program consists a goal wrapped inside of a `run` form which, with the help of zero or more relation definitions, outputs a list of answers, which can be thought of as normalized and pretty-printed states.

However, we can also analyze the language from a high-level point of view in terms of its primitives, its means of combination, and its means of abstraction (Abelson et al. 1996, p. 359). miniKanren's primitives are **logical constraints** and a special form to create and bind new logic variables. Its two means of combination are conjunction and disjunction. Finally, its only means of abstraction is relation.

`run` is the interface converting goals to programs which produce answers. To do this, `run` catches the stream of states produced by the goal and turns it into a list of specified length. In the process, it also converts these states into a more readable format – a process called **reification** – which also highlights the values of **query variables**. We will not describe precisely what the format produced by reification is, but it can easily be intuitively grasped as we introduce more examples. Meanwhile, readers need only know that `run` has the following syntax:

```
(run <answer-count> (<query-var-1> <query-var-2> ...) <goal>)
```

Additionally, users can use this form to get back all answers:

```
(run* (<query-var-1> <query-var-2> ...) <goal>)
```

The most important goal constructor (primitive constraint) is ==. Simply put, == unifies two terms, declaring them to be equal. More precisely, == receives a state and returns a stream. This stream can either contain one state where the two terms become their most general unifier, or be empty when they cannot be unified.

Actually, == is the only constraint in the what is called the ”core” of miniKanren. Even with only this constraint, the language is already Turing-complete. However, none but the simplest programs can be written conveniently using ==.

Below are some example usage of ==.

```
(run* (q)
  (== 'x 'x))
```

⇒ (_.0)

```
(run* (q)
  (== '(x y) '(x y)))
```

⇒ (_.0)

```
(run* (q)
  (== 'x 'y))
```

⇒ ()

In the first and second example, the two arguments to == are already equal, hence the program returns one successful answer. In the third example, the two terms `x` and `y` can never be unified and the program returns the empty list indicating that there is no possible answer. In these cases, == is only used to compare terms. == becomes a lot more interesting when variables are involved.

To make logic variables, we must use the goal constructor (fresh variable creator) `fresh`. `fresh` is perhaps the most complex goal constructor in miniKanren, partly because it ifs also a special syntactic form. Operationally, `fresh` creates one or more fresh logic variables. Syntactically, `fresh` also bind these newly created variables to Scheme identifiers within its body.

Let us take a look at some examples.

```
(run* (q)
  (fresh (v)
    (== v 'x)
    (== q v)))
```

⇒ (x)

```
(run* (q)
  (fresh (v u)
    (== v 'x)
    (== u v)
    (== q u)))
```

⇒ (x)

The first program introduces a new logic variable `v`, which is then unified with both the symbol `x` and the query variable `q` (the order does not matter). The only answer returned is the value of `q`, which is the symbol `x`.

In the second program, two variables are introduced. Since `u` is unified with `v` and the rest of the program is the same, the answer is also the same.

We cannot refer to a logic variable not introduced by, or not in the body of `fresh`. For example, this call fails.

```
(run* (q)
  (fresh (v)
    (== q v)  ; Perfectly fine
    (== q u)  ; Error: variable u is unbound
  )
  (== q v)  ; Error: variable v is unbound
)
```

The ”variables” mentioned in the error messages are naming variables, not logic variables. Technically speaking, logic variables need not have names. When talking about ”the variable `v`”, we actually mean ”the logic variable bound to the naming variable `v`”. This last example showcases lexical scope created by `fresh`.

```
(run* (q)
  (fresh (v)
    (== v 'x)
    (== q v)  ; v bound to the symbol x
    (fresh (v)
      (== v 'y)
      (== q v)  ; v bound to the symbol y
      )))
```

⇒ ()

This call fails because `q` was unified to two logic variables having two different values; they are just happen to be bound to the same naming variable `v`. The first `v` is introduced by the outer `fresh`, the second `v` is introduced by the inner `fresh` and **shadows** the outer `v`. Although shadowing seems to be a very confusing thing in this example, it is actually very useful especially for large programs.

For our purpose, it is important to demystify a little bit the answer format returned by miniKanren (more precisely, by `run`). First, unknown variables declared to be the same (e.g. by `==`) always have the same index, for instance:

```
(run* (p q r) (== p q))
```

⇒ ((_.0 _.0 _.1))

Second, if a variable is tied to a value then the value will be used instead of the form `_.n`. Furthermore, if a variable is sure to be a pair then that fact is also made known in the answer, for example:

```
(run* (q) (fresh (x y) (== `(,x ,y) q)))
```

```
⇒ ((_.0 _.1))
```

A simple principle to keep in mind is that *information must be preserved during reification*. The only lost information, however, are the names of the logic variables. The reason is due to the name conflict phenomenon mentioned earlier; that is, if we were to represent unknown variables by their names then there would be no way of telling whether two variables were really declared to have the same value or they just happen to bear the same name. As we shall see later, there is a simple way to recover names of unbound variables.

Next, we take a look at the goal constructor (goal disjunction combinator) `conde`. If one needs to point out exactly one thing that makes logic programming seem magical, disjunction would be a good answer. Every example we have seen returns either zero or one answer. With `conde`, we can start returning two or more answers. The `conde` form consists of a number of **clauses**, each contain a number of goals, for example:

```
(run* (q)
  (conde
    [(== 'x q)]
    [(== 'y q)]))
```

```
⇒ ((x y))
```

As promised, this program actually returns two answers. We can view `conde` as creating two ”parallel universes” where `q` is unified with `x` in the first and `y` in the second.

The final goal constructor (goal conjunction combinator) of core miniKanren... does not actually exist. However, we have already encountered conjunction multiple times in previous examples. Intuitively, conjunction is implicitly used whenever two or more goals are written in series in the body of `run`, `fresh` or `conde` clauses. Each of the examples below returns `()` due to value conflict.

```
(run* (q)
  (conde
    [(== 'x q) (== 'y q)]))

(run* (q)
  (== 'x q)
  (== 'y q))

(run* (q)
  (fresh ()
    (== 'x q)
    (== 'y q)))
```

The introduction of miniKanren's core should be complete with `==`, `fresh`, `conde` and conjunction. However, we will also take a look at the most important extra constraint `=/=`. As the name suggest, `=/=` declares **disequality** of two miniKanren terms:

```
(run* (q) (=/= 5 q))

⇒ ((_.0 (=/= ((_.0 5)))))
```

The program above returns one answer, stating that while `q` is unknown, it must not be made equivalent to `5`. If we happen to do that, the program fails:

```
(run* (q) (=/= 5 q) (== q 5))

⇒ ()
```

To better understand answers containing disequalities, a more complex example is needed:

```
(run* (q p r) (=/= `(,q ,p) `(1 2)) (=/= p r))

⇒ (((_.0 _.1 _.2)(=/= ((_.0 1)(_.1 2))((_.1 _.2)))))
```

We see that each disequality constraint store, as presented in the answers, is a list of mini anti-substitution lists. Each anti-substitution list in turns contains disassociations of the form `(x v)` where `x` must be a variable. Failure happens when all disassociated terms in one anti-substitution list are equal. In plain English, the answer above says that p is different from r and that either q must be different from 1 or p must be different from 2, which is exactly what we declared in the program.

3 RELATIONAL CONDITIONALS WITH PSEUDO-FUNCTIONS

Suppose we need to write a relation called lookupo to retrieve a variable's value in an environment. The task at first is just a straightforward transformation from the Scheme function lookup.

```
(define lookup
  (lambda (x env)
    (let ([a (car env)])
      (cond
       [(eq? x (lhs a)) (rhs a)]
       [else (lookup x (cdr env))]))))

(define lookupo
  (lambda (x env t)
    (fresh (y b rest)
      (== `((,y ,b) . ,rest) env)
      (conde
       [(== y x) (== b t)]
       [(=/= y x) (lookupo x rest t)]))))
```

Later on we might need to actually handle the case where the variable is unbound instead of raising an error or fail. We update both definitions:

```
(define lookup
  (lambda (x env)
    (cond
     [(null? env) #f]
     [else
      (let ([a (car env)])
        (cond
         [(eq? x (lhs a)) a]
         [else (lookup x (cdr env))])))])))

(define lookupo
  (lambda (x env t bound?)
    (conde
     [(== '() env) (== #f bound?)]
     [(fresh (y b rest)
        (== `((,y ,b) . ,rest) env)
        (conde
         [(== y x) (== #t bound?) (== b t)]
         [(=/= y x) (lookupo x rest t bound?)]))])))
```

So far so good: `lookup` only needs to add an input indicating whether the variable is bound; meanwhile for `lookup` we have to devise a special signal, namely `#f`, for the unbound case[2]. However, a problem arises when we actually use the output signal:

```
(define case1
  (lambda (x env)
    (cond
     [(lookup x env) => rhs]
     [else #f])))

(define case1o
  (lambda (x env out)
    (conde
     [(lookupo x env out #t)]
     [(lookupo x env 'unbound #f) (== #f out)])))
```

Why can we not use `else` in the second clause? The reason is that despite the similarity in appearance, `conde` does not process its clauses from top to bottom

[2]There is technically a way to return multiple values in Scheme, but doing so would be lengthy and unconventional.

like `cond` does, so there can be no default case. As a result, programmers must always be prepared to append extra denials in lower `conde` clauses. This issue gets serious when the control flow is more complex:

```
(define case2
  (lambda (x y env)
    (cond
     [(lookup x env) => rhs]
     [(lookup y env) => rhs]
     [else #f])))

(define case2o
  (lambda (x y env out)
    (conde
     [(lookupo x env out #t)]
     [(lookupo x env '? #f) (lookupo y env out #t)]
     [(lookupo x env '? #f) (lookupo y env '? #f)
      (== #f out)])))
```

The example above is still very generous, as it is usual for relations to include four or more `conde` clauses. The number of extra denials exhibits quadratic growth, introducing opportunities for errors without any significant contribution to the meaning of the program. Fortunately, there is a simple way to deal with them using a technique inspired by Neumerkel & Kral (2016). This solution relies on **pseudo-functions**.

Pseudo-functions receive their results as arguments instead of returning them. To make sense of this concept, observe the following transformation (the ”t” at the end denotes pseudo-functions, simply because it is too late to change):

```
;; From function
(define bar (lambda (in) 'v))
;; To relation
(define baro (lambda (in out) (== out 'v)))
;; To pseudo-function...
```

```
(define bart (lambda (in) (lambda (out) (baro in out))))
;; ...which is equivalent to
(define bart (lambda (in) (lambda (out) (== out 'v))))
```

From a low-level point of view, a pseudo-function is a function which takes a single argument and return a goal. It can be seen that the act of returning values in functional programming is analogous to the act of unification in logic/relational programming. Moreover, a relation can be made to return whichever of its formal parameters, depending on the circumstances:

```
;; From relation
(define conso (lambda (a d ls) (== `(,a . ,d) ls)))
;; To pseudo-function 1
(define cart (lambda (ls) (lambda (a) (fresh (d) (conso a d
   ls)))))
;; To pseudo-function 2
(define cdrt (lambda (ls) (lambda (d) (fresh (a) (conso a d
   ls)))))
;; To pseudo-function 3
(define const (lambda (a d) (lambda (ls) (conso a d ls))))
```

I have never encountered a situation where this is actually useful, since there is generally only one parameters deemed as the ”output”. It is worth noting that even though pseudo-functions act like pure functions, they can affect more than just the output. For instance, the goal `((cart x)a)` will always instantiate `x` to a pair if it is currently unknown.

Going back to the problem of conditionals, we first define two trivial pseudo-functions `truet` and `falset`, ”returning” `#t` and `#f` respectively.

```
(define truet (lambda () (lambda (?) (== #t ?))))
(define falset (lambda () (lambda (?) (== #f ?))))
```

Next, we define a single ”primitive” pseudo-function similar to Scheme’s `eq?`[3]. This

[3]Depending on the implementation, more primitive pseudo-functions may be defined using

function relies on both == and =/= to correctly express two branches of the output.

```
(define ==t
  (lambda (x y)
    (lambda (?)
      (conde
       [(== #t ?) (== x y)]
       [(== #f ?) (=/= x y)]))))
```

Next, we provide ways of creating more complex pseudo-functions from simpler ones, starting with `negt` (negation):

```
(define negt
  (lambda (g)
    (lambda (?)
      (conde
       [(== #t ?) (g #f)]
       [(== #f ?) (g #t)]))))
```

From this, `=/=t` can be trivially derived:

```
(define =/=t
  (lambda (x y) (negt (==t x y))))
```

Similarly, `conjt` (conjunction) and `disjt` (disjunction) are defined using macro:

```
(define-syntax conjt
  (syntax-rules ()
    [(_) (truet)]
    [(_ g) g]
    [(_ g1 g2 gs ...)
     (lambda (?)
       (conde
        [(g1 #t) ((conjt g2 gs ...) ?)]
        [(== #f ?) (g1 #f)]))]))
```

additional primitive constraints.

```
(define-syntax disjt
  (syntax-rules ()
    [(_) (falset)]
    [(_ g) g]
    [(_ g1 g2 gs ...)
     (lambda (?)
       (conde
        [(== #t ?) (g1 #t)]
        [(g1 #f)  ((disjt g2 gs ...) ?)]))]))
```

This next example demonstrates conjunction:

```
(run* (t x y z)
   ((conjt (==t x y)
           (==t y z))
    t))


⇒

(((#f _.0 _.1 _.2) (=/= ((_.0 _.1))))
 (#t _.0 _.0 _.0)
 ((#f _.0 _.0 _.1) (=/= ((_.0 _.1)))))
```

Similarly, this example demonstrates disjunction:

```
(run* (t x y z)
   ((disjt (==t x y)
           (==t y z))
    t))


⇒

((#t _.0 _.0 _.1)
 ((#t _.0 _.1 _.1) (=/= ((_.0 _.1))))
 ((#f _.0 _.1 _.2) (=/= ((_.0 _.1)) ((_.1 _.2)))))
```

Finally, we can define the goal constructor `condo` – the closest relational counterpart of cond. Keep in mind that succeed stands for the ”do nothing” goal and fail stands for the ”always fail” goal.

```
(define-syntax condo
  (syntax-rules (else)
    [(_ [else]) succeed]
    [(_ [else g]) g]
    [(_ [else g1 g2 g* ...])
     (fresh () g1 g2 g* ...)]
    [(_ [test g* ...] c* ...)
     (conde
      [(test #t) g* ...]
      [(test #f) (condo c* ...)])]
    [(_) fail]))
```

condo is an interesting combinator because it uses two types of objects, pseudo-functions in the test positions and normal miniKanren goals in the bodies of each clause. The else keyword can be used to signify the default case; although it can be omitted, in which case the program fails when no clause matches. The example below demonstrates the use of condo:

```
(run* (x y z)
  (condo
   [(==t x y) succeed]
   [(==t y z) succeed]
   [else (== x z)]))
```

⇒

```
((_.0 _.0 _.1)
 ((_.0 _.1 _.1) (=/= ((_.0 _.1))))
 ((_.0 _.1 _.0) (=/= ((_.0 _.1)))))
```

We are now able to replace lookupo with the pseudo-function lookupt.

```
(define lookupt
  (lambda (x env t)
    (lambda (bound?)
      (conde
       [(== '() env)
        (== #f bound?) (== t 'unbound)]
       [(fresh (y b rest)
          (== `((,y ,b) . ,rest) env)
          (condo
           [(==t y x) (== #t bound?) (== b t)]
           [else
            ((lookupt x rest t) bound?)]))])))))
```

Now the comparison between `case2` and `case2o` does not look so bad anymore.

```
(define case2
  (lambda (x y env)
    (cond
     [(lookup x env) => rhs]
     [(lookup y env) => rhs]
     [else #f])))

(define case2o
  (lambda (x y env)
    (lambda (out)
      (condo
       [(lookupt x env out) succeed]
       [(lookupt y env out) succeed]
       [else (== #f out)]))))
```

Some readers may already be convinced to convert all of their relations to pseudo-functions. However, there are still some issues to address. Although hiding the control flow is great for readability, the downside is that there is no easy way to see what is done behind the layer of abstraction. Therefore, it is very easy for people to accidentally change the behavior of programs without knowing it. Take for example this goal using `condo`:

```
(fresh (x y)
  (condo
   [(==t 'a x) (== 'A y)]
   [(==t 'b x) (== 'B y)]
   [(==t 'c x) (== 'C y)]))
```

It is not clear that there is an issue with this code, because it looks so much like normal Scheme. The issue only shows itself when we expand the `condo`:

```
(fresh (x y)
  (conde
   [(== 'a x) (== 'A y)]
   [(=/= 'a x) (== 'b x) (== 'B y)]
   [(=/= 'a x) (=/= 'b x) (== 'c x) (== 'C y)]))
```

Clearly, the additional denials in these `conde` clause are redundant since they are already mutually exclusive. Moreover, switching from `conde` to `condo` always means changing the way miniKanren's search work.

As reality has shown that industrial miniKanren users who are conscious about their programs' behavior cannot afford `condo`. However, that also means we are once again stuck with the lengthy, error-prone ways of writing conditionals. To address this problem, this paper offers a way of verifying the correctness of existing programs with **staticKanren**.

4 STATICKANREN

This section introduces staticKanren, a language designed to analyze miniKanren programs. We first step through a few simple examples in section 4.1 to develop intuition for the detailed implementation in section 4.2. Section 4.3 and 4.4 demonstrate some applications of the language.

4.1 Introductory examples

On the surface, staticKanren was designed to look as close to miniKanren as possible; however there are still some fundamental differences. Unlike miniKanren, staticKanren cannot deal with infinite answer streams. In fact, staticKanren programs can only return all answers using `run*` as there is no point withholding computed answers. As a result, we cannot write non-trivial recursive relations in staticKanren as-is. Looking on the bright side, this gives the language a simpler implementation.

staticKanren offers only three primitives constraints: equality, disequality and **lazy** constraint. The first two retain their syntax and semantics from miniKanren, whereas the third is a new addition to emulate recursive relations. There is no reason not to include more primitive constraints, the more knowledge staticKanren has, the better answers it can return. However, adding other constraints would be a distraction from our goal. Therefore, only disequality is implemented because it plays such an important role in relational programs.

In staticKanren, answers are also interpreted as programs. For that reason, the language must treat variable names more seriously. For example (in this section, ⇒ means ”evaluating under staticKanren”):

```
(run* (q p) (== q 4))


⇒ (((4 #(p 0))() ()))
```

If the program above were run under miniKanren instead, the answer would have been `((4 _.0))`. The variable `p` remains unknown in the answer, so it gets to keep its name (let us ignore the zero on the right for now). Besides the query variables, staticKanren always returns two additional constraint stores, one for disequalities and one for fake constraints. Hence, every answer has the same shape of a three-element list. The next example demonstrates how the language deals with name shadowing:

```
(run* (q a) (fresh (a d) (== q `(,a . ,d))))
```

```
⇒ ((((#(a 1). #(d 1))#(a 0))() ()))
```

The same program under miniKanren would return `(((_.0 . _.1)_.2))` instead. There are two variables having the same name `a`. staticKanren can still distinguish these two variables by their **birthdate** (the number shown on the right). Meanwhile, miniKanren sidesteps this problem completely by introducing a new name for every unbound variables used in the answer. It is perhaps a subjective matter to debate which representation is better, but there is one at least one theoretical problem avoided by the new approach[4]:

```
(run* (q p) (== q '_.0))

⇒ (((_.0 #(p 0))() ()))
```

The same program under miniKanren would return `((_.0 _.0))`!. And because two Scheme symbols are the same iff they look the same, there is no way to distinguish `p` and `q` in that case, even from the computer's perspective. The last example demonstrates fake constraints. It is a silly program to stress that these fake constraints have no actual meaning, as the `fake` form does no more than evaluating its body and add the resulting expression to the fake constraint store.

```
(run* (q p r) (fake `(mother ,q ,p)) (fake `(father ,p ,r)))

⇒

(((#(q 0) #(p 0) #(r 0))
 ()
 ((father #(p 0) #(r 0)) (mother #(q 0) #(p 0)))))}
```

4.2 Implementation

This section presents an R6RS compliant implementation of staticKanren. We will highlight only those parts that demonstrates fundamental differences from the

[4]This example was even mentioned in one of the online miniKanren's Uncourse Hangouts organized by William Byrd.

standard miniKanren implementation describe by Byrd (2009)[5]. The full implementation can be found at `https://github.com/lackhoa/staticKanren`.

The following functions construct and dispatch variables. Variables are vectors of two elements: its name (a symbol) and its birthdate (a number). Additionally, we would like to create a total order on the set of variables. A variable is prioritized over another if its birthdate is less (i.e. it was introduced sooner), or if it has the same birthdate and a lexicographically lesser name. We call prioritized variables **seniors** and less prioritized ones **juniors**.

```
(;; name is a symbol, bd is a number
 define var (lambda (name bd) (vector name bd)))
(define var->name (lambda (var) (vector-ref var 0)))
(define var->bd (lambda (var) (vector-ref var 1)))
(define var? vector?)
(define var<?
  ;; v1 is prioritized over v2
  (lambda (v1 v2)
    (let ([n1 (symbol->string (var->name v1))] [bd1 (var->bd
       v1)]
           [n2 (symbol->string (var->name v2))] [bd2 (var->bd
              v2)])
      (or (< bd1 bd2)
          (and (= bd1 bd2) (string<? n1 n2))))))
```

Next we have definitions about constraints. Many of the functions could easily be defined automatically using record, but we use list to retain conceptual simplicity and flexibility. A constraint is a 4-tuple consisting of a substitution `S`, a date counter `C`, a disequality store `D` and a fake constraint store `F`. The counter starts from 0 and only goes upwards. Additionally, `letg@` is a special form to dispatch on states.

```
(define all-constraints '(S C D F))
```

[5]However, this does not mean that merely overriding the definitions of miniKanren by the ones in this paper would give a working staticKanren since there are many other small technical differences.

```
(define init-S '())
(define init-C 0)
(define init-D '())
(define init-F '())
(define make-c (lambda (S C D F) (list S C D F)))
(define init-c (make-c init-S init-C init-D init-F))
(define c->
  (lambda (c store)
    (rhs (assq store (map list all-constraints c)))))
(define update-S (lambda (c S) (letg@ (c : C D F) (make-c S C
   D F))))
(define update-C (lambda (c C) (letg@ (c : S D F) (make-c S C
   D F))))
(define update-D (lambda (c D) (letg@ (c : S C F) (make-c S C
   D F))))
(define update-F (lambda (c F) (letg@ (c : S C D) (make-c S C
   D F))))
```

The final piece of groundwork concerns the answer stream monad. Because there are no lazy streams, definitions are trivial to implement and we keep them here only for the sake of comparison.

```
(define mzero (lambda () '()))
(define unit (lambda (x) `(,x)))
(define choice (lambda (x y) `(,x . ,y)))
(define mplus append)
(define bind (   (c* g) (apply mplus (map g c*))))
```

`unify` takes two terms along with a substitution and returns the extended substitution where these two terms are made the same. In the case that the two terms are always different, however, `unify` returns `#f`. There is a new clause in staticKanren to handle the case where both walked terms are variables, we make sure that seniors are always on the rhs. That way, walked seniors never result in juniors.

```
(define unify
  (lambda (t1 t2 S)
    (let ([t1 (walk t1 S)]
          [t2 (walk t2 S)])
      (cond
       [(eq? t1 t2) S]
       [(and (var? t1) (var? t2))
        (or (and (var<? t2 t1)
                 (extend S t1 t2))
            (extend S t2 t1))]
       [(var? t1) (extend-check t1 t2 S)]
       [(var? t2) (extend-check t2 t1 S)]
       [(and (pair? t1) (pair? t2))
        (let ([S+ (unify (car t1) (car t2) S)])
          (and S+ (unify (cdr t1) (cdr t2) S+)))]
       [(equal? t1 t2) S]
       [else #f]))))
```

Now we are ready to define user-level functions, starting with fake constraint.

```
(define fake
  (lambda (expr)
    (lambdag@ (c : F)
      (unit (update-F c `(,expr . ,F))))))
```

Next is `fresh`, a conjunction in the body of a `letv`. `letv` extracts the date counter from its input state and assign it to the new variables; the counter is then incremented in the resulting states.

```
(define-syntax fresh
  (syntax-rules ()
    [(_ (x* ...) g g* ...)
     (letv (x* ...) (conj g g* ...))]))


(define-syntax letv
  (syntax-rules ()
    [(_ (x* ...) g)
     (lambdag@ (c : C)
       (let ([x* (var 'x* C)] ...)
         (g (update-C c (+ C 1)))))]))
```

The fascinating part is that reification can be made even simpler due to the fact that unknown variables get to keep their names. For the final result, `reify` only need to deep walk the query variables and the constraint stores in `S`. A more compact representation of the disequality store is then obtained by removing constraints that containing either no variable or irrelevant variable(s) (i.e. variable that does not occur in the final version of `q*` or `F`).

```
(define reify
  ;; This will return a c with clausal S
  (lambda (c q*)
    (letg@ (c : S D F)
      (let ([t (walk* q* S)]
            [D (walk* D S)]
            [F (walk* F S)])
        (let ([R (get-vars `(,t ,F))])
          (let ([D (rem-subsumed (purify-D D R))])
            `(,t ,D ,F)))))))
```

For closure, helpers of `reify` are given below. It is worth noting that we treat constraint stores as normal miniKanren terms, which explains why the code is so short. Additionally, `case-term` is a macro form which helps dispatch on term. There are three cases for a term: variable, pair and atom. The cases are treated in that order.

```
(define get-vars
  (lambda (t)
    (case-term t
      [v `(,v)]
      [(a d) (append (get-vars a) (get-vars d))]
      [a '()])))

(define purify-D
  (lambda (D R)
    (filter (lambda (d)
              (not (or (constant? d)
                       (has-iv? d R))))
            D)))

(define constant?
  (lambda (t)
    (case-term t
      [v #f]
      [(a d) (and (constant? a) (constant? d))]
      [atom #t])))

(define has-iv?
  (lambda (t R)
    (let has-iv? ([t t])
      (case-term t
        [v (not (memq v R))]
        [(a d) (or (has-iv? a) (has-iv? d))]
        [atom #f]))))
```

4.3 Converting answer clausal form into substitution

Let us begin exploring the practical value of staticKanren with a simple problem of returning substitutions instead of walked query variables in the answers. We achieve this with the new `run*su` form. Answers produced by `run*su` looks like this instead of `((_.0 _.0 _.0))`.

```
(run*su (q p r) (== q p) (== p r))

⇒ ((((#(r 0)#(p 0))(#(q 0)#(p 0)))() ()))
```

We see that the answer looks exactly like the program, which in turns highlights a fundamental fact about states. miniKanren states are just accumulators of logical assertions given throughout execution of the program. By returning states as answers, we can summarize the content of the program. As a matter of fact, answers may be simpler than programs, as the following case shows.

```
(pp (run*su (q p) (== q p) (== p q)))


⇒ ((((#(q 0)#(p 0)))() ()))
```

Since `==` is commutative, the assertion `(== p q)` is redundant and it is removed from the final answer. It might look as if a complex inference mechanism is being used, but in fact this feature only requires a very simple change to `run*`. Unification has already done much of the work building up a clean substitution, we only need to retrieve it back after it has been cleaned up even more by `reify`. Without further hand-waving, here is `run*su`.

```
(define-syntax run*su
  ;; run* with substitutions
  (syntax-rules ()
    [(_ (q q* ...) g g* ...)
     (let ([q (var 'q init-C)] [q* (var 'q* init-C)] ...)
       (let ([qs `(,q ,q* ...)])
         (let ([c* ((conj g g* ... (finalize qs))
                    (update-C init-c (+ init-C 1)))])
           (map (lambda (c) (su c qs)) c*))))]))


(define su
  (lambda (c qs)
    `(,(unify (car c) qs init-S)
      .
      ,(cdr c))))
```

This much of development already lets us to obtain the ”negative” side of the `lookupt` function defined in section 3. However, we first need to redefine `lookupt` so that the recursive call is fake.

```
(define lookupt
  (lambda (x env t)
    (lambda (bound?)
      (conde
       [Unchanged ...]
       [(fresh (y b rest)
          (== `((,y . ,b) . ,rest) env)
          (condo
           [Unchanged ...]
           [else
            (;; The only place that is changed
             fake `((lookupt ,x ,rest ,t) ,bound?))]))]))))

(run*su (x env) ((lookupt x env 'unbound) #f))
```

⇒

```
(((#(x 0) ())
  ()
  ())
 ((#(x 0)
   ((#(y 1) . #(b 1)) . #(rest 1)))
  (((#(y 1) #(x 0))))
  (((lookupt #(x 0) #(rest 1) unbound) #f))))
```

This response is akin to the goal:

```
(conde
 [(== '() x)]
 [(== `((,y . ,b) . ,rest) x)
  (=/= y x)
  ((lookupt x rest 'unbound) #f)])
```

The second clause is particularly interesting: it states that if the environment is non-empty and the look-up is to fail, then the variable `x` must not match the association's lhs and the look-up must also fail for the rest of the environment.

4.4 Analyzing common unification pattern with anti-unification

There are still cases where the answers returned do not look at all similar to the program that generated them. Take `membero` for example:

```
(define membero
  (lambda (x ees)
    (fresh (e es)
      (== ees `(,e . ,es))
      (condo
       [(==t x e) succeed]
       [else (fake `(membero ,x ,es))]))))
```

Rewritten through `run*su`, this gives.

```
((((#(ees 0) (#(x 0) . #(es 1))))
  ()
  ())
 (((#(ees 0) (#(e 1) . #(es 1))))
  (((#(e 1) #(x 0))))
  ((membero #(x 0) #(es 1)))))
```

Which is equivalent to this program.

```
(define membero
  (lambda (x ees)
    (conde
     [(fresh (es) (== `(,x . ,es) ees))]
     [(fresh (e es)
        (== `(,e . ,es) ees)
        (=/= e x)
        (membero x es))])))
```

Human readers would agree that this version does not look at all natural: it does not show the common structure of `ees` in both clauses. To make this notion of commonness more clear, we refer to **anti-unification**, which as the name suggests is a dual of unification. Given a collection of terms, anti-unification returns their least general generalization. This process describes quite well humans' ability to recognize pattern over symbolic formulae, which is exactly what we need for the purpose. For example,

```
(let ([t* '((1 * 2 = 2 + 1)
            (4 * 3 = 3 + 4))])
  (anti-unify t*))


⇒ (#(au0 0.5)* #(au1 0.5)= #(au1 0.5)+ #(au0 0.5))
```

Anti-unification has captured the common pattern from these two formulae. `au0` and `au1` are normal staticKanren variables introduced in the process, which from now on we shall refer to as **pattern variables**. All pattern variables have the birthdate `0.5`, the reason for which will become clear when the program is complete.

The implementation of `anti-unify` is adapted from Østvold (2004). The biggest difference is that this version also recognizes identical variables as being the same terms and reuse that variable in the pattern.

```
(define anti-unify
  (lambda (t*)
    (let-values
        ([(res _iS)
          (let au ([t* t*] [iS '()])
            (cond
             [;; rule 7: eq? deal with variables as well
              ;; hence, it would not introduce useless new vars
              (for-all (eqp? (car t*)) (cdr t*))
              (values (car t*) iS)]
             [;; rule 8
              (for-all pair? t*)
```

```
               (let-values ([(a iS+) (au (map car t*) iS)])
                 (let-values ([(d iS++)
                               (au (map cdr t*) iS+)])
                   (values `(,a . ,d) iS++)))]
              [;; rule 9
               (find (lambda (s) (teq? (lhs s) t*)) iS)
               =>
               (lambda (s) (values (rhs s) iS))]
              [;; rule 10
               else
               (let ([new-var
                      (var (au-name (length iS)) AU-BD)])
                 (values new-var (extend iS t* new-var)))]))])
      res)))

(define eqp? (lambda (u) (lambda (v) (eq? u v))))

(define AU-BD (+ init-C 0.5))

(define au-name
  (lambda (n)
    (string->symbol (string-append "au" (number->string n)))))
```

Anti-unification is the heart of our solution, but the way to victory is not laid out just yet. It is not clear how to convert the obtained pattern into a usable program. We start by making a small step just like in section 4.3.

```
(define-syntax run*au
  ;; run* with anti-unification analysis
  (syntax-rules ()
    [(_ (q q* ...) g g* ...)
     (let ([q (var 'q init-C)] [q* (var 'q* init-C)] ...)
       (let ([qs `(,q ,q* ...)])
         (let ([c* ((conj g g* ... (finalize qs))
                    (update-C init-c (+ init-C 1)))])
```

```
            (au-extract c* qs))))]))
```

The main work is now delegated to `au-extract`, which takes the answers and the query variables as input. First, `au-extract` takes the answer terms `t*` and extracts a common pattern `au`. Second, it unifies the queries variables with `au` in the empty environment, resulting in the substitution `uS`. The next obvious step is to unify `au` back into each of the answer term in `t*` to obtain the substitutions `S*`. Finally, we clean up the substitutions with `purify-S` and walk the other constraints in those substitutions.

```
(define au-extract
  (lambda (c* qs)
    (let ([t* (map car c*)]
          [D* (map cadr c*)]
          [F* (map caddr c*)])
      (let ([au (anti-unify t*)])
        (let ([auS (unify qs au init-S)])
          (let ([S* (map (lambda (t) (prefix-unify au t auS))
                         t*)])
            `(,(purify-S auS init-C)
              ,(map au-helper S* D* F*))))))))

(define prefix-unify
  (lambda (t1 t2 S) (prefix-S (unify t1 t2 S) S)))

(define au-helper
  (lambda (S D F)
    (let ([S (purify-S S AU-BD)]
          [D (walk* D S)]
          [F (walk* F S)])
      `(,S ,D ,F))))
```

The only question left is how to clean up the substitutions. We have already done this before with the help of `walk*`, but this time things are not so convenient because we also want to retain the intermediate associations with the anti-unification patterns. There is a first principle which seems to work well: An association is redundant iff it is a rename (an association whose rhs is a variable) whose lhs has not yet been introduced yet. This is because we can just use the rhs without having to ever introduce the lhs. Conveniently, `purify-S` can keep track of already introduced variables using their birthdates.

```
(define purify-S
  (lambda (S date)
    (filter (lambda (s) (<= (var->bd (lhs s)) date))
            S)))
```

This should also explain why pattern variables are born on `0.5`: they are introduced after query variables, but before any other variable introduced by the user. With the complete implementation, we can start using `run*au` to rewrite programs. The call below will return `lookupo`, obtained by forcing `lookupt` to return `#t`. Notice that the recursive call of `lookupt` has been replaced by a fake one just like in the previous section.

```
(run*au (x env t) ((lookupt x env t) #t))
```

⇒

```
(((#(env 0)
   ((#(au0 0.5) . #(au1 0.5)) . #(rest 1))))
 ((((#(au1 0.5) (val . #(t 0)))
    (#(au0 0.5) #(x 0)))
   ()
   ())
  (((#(t 0)
     (closure
      #(lam-expr 2)
      ((#(x 0) rec . #(lam-expr 2)) . #(rest 1))))
    (#(au1 0.5) (rec . #(lam-expr 2)))
    (#(au0 0.5) #(x 0)))
   ()
   ())
  (()
   (((#(au0 0.5) #(x 0))))
   (((lookupt #(x 0) #(rest 1) #(t 0)) #t)))))
```

This response is akin to the goal:

```
(fresh (au0 au1)
  (== `((,au0 . ,au1) . ,rest) env)
  (conde
   [(== au0 x) (== `(val . ,t) au1)]
   [(== au0 x)
    (== au1 `(rec . ,lam-expr))
    (== `(closure
          ,lam-expr
          ((,x . (rec . ,lam-expr)) . ,rest))
       t)]
   [(=/= au0 x)
    ((lookupt x rest t) #t)]))
```

Anti-unification identifies that across all clauses, the environment is always a non-empty list with the first element being an association. It also correctly handles how each clause has to do afterwards to obtain the original semantics. Especially the third clause is a clever assertion: it says that if the association's lhs (`au0`) is different from the variable `x` and the look-up is to succeed, then the look-up of `x` in the tail of the environment has to succeed.

5 CONCLUSION

This paper presented the following contributions:

1. A technique of using pseudo-functions to express conditionals in miniKanren.

2. The implementation a miniKanren variant to help extract certain modes of programs.

3. A way of reformatting staticKanren's answers to a more human-oriented style.

The goal of this thesis is to explore techniques of avoiding code duplication of positive and negative relations. We learned along the way the important fact that while conditionals can be easily expressed with `condo`, the behavior of the program is almost impossible to preserve. The best solution to this problem suggested by this paper is to specify the two-way algorithm, run it through staticKanren then manually check and modify the generated result if needed for the final program. While this solution is not ideal due to the additional framework in staticKanren, I believe that it is a necessary price to pay for the mechanization of negation while retaining utmost user flexibility. An alternative approach no explored here is to use constructive negation (Chan 1989), although it seems significantly more complex to implement.

There is an interesting use case of staticKanren in Scheme program analysis. With the help of `condo` it is very easy to convert any Scheme program into staticKanren and extract its various modes. Additionally, pseudo-functions might also be useful for expressing other useful functional programming features such as composition.

I thank my thesis supervisor Timo Hynninen for the advice at the start of the process. Additionally, I am grateful for William Byrd for spending time answering my questions. The topic idea of this thesis is largely thanks to him.

REFERENCES

Abelson, H., Sussman, G. & Sussman, J. 1996. Structure and interpretation of computer programs. Electrical engineering and computer science series. MIT Press.

Byrd, W. 2009. “Relational Programming in miniKanren: techniques, applications, and implementations”. PhD thesis. Indiana University.

Chan, D. 1989. “An extension of constructive negation and its application in coroutining”. *Logic Programming: Proceedings of the North American Conference 1989*. Vol. 1. Mit Press, 477–493.

Friedman, D., Byrd, W., Kiselyov, O. & J., H. 2018. The reasoned Schemer. 2nd ed. MIT Press.

Hemann, J. & Friedman, D. 2013. “muKanren: a minimal functional core for relational programming”. *2013 workshop on Scheme and functional programming*.

Hewitt, C. 2008. Development of logic programming: what went wrong, what was done about it, and what it might meant for the future. Tech. rep. AAAI workshop.

Kowalski, R. A. 1988. The early years of logic programming. *Communications of the ACM* 31.1, 38–44.

Neumerkel, U. & Kral, S. 2016. Indexing dif/2. *CoRR* abs/1607.01590.

Østvold, B. 2004. A functional reconstruction of anti-unification. Tech. rep. Norwegian Computing Center.

Smullyan, R. 1978. What is the name of this book?: the riddle of Dracula & other logical puzzles. Prentice Hall.