Khoa Vo

RELATIONAL CONDITIONALS WITH PSEUDO-FUNCTIONS

Techniques in miniKanren

Bachelor's thesis

Degree programme in Information Technology

2019

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	45	0
Commissioned by		
Self initiative		
Supervisor		
Timo Hynninen		

Abstract

Relational programming promises the ability to run programs backwards. Using miniKanren, it is 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 thesis 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 offers to make backtracking implicit; programmers only need to declare the rules, and the computer will automatically generate all states consistent within these rules. Logic languages often give elegant solutions to a wide class of problems, some can also be 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. It is a well-known fact that while these features on one hand can enhance performance, they can also make programs more difficult to reason about. 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 and the same set of answers is returned regardless of the order in which rules are applied. The design philosophy of **miniKanren** puts a lot of 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 thesis 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 present text short.

This thesis aims to provide techniques for writing conditionals in miniKanren. I also propose a method for analyzing the correctness of pre-existing programs as well as automatically generating new ones using conditionals. The main component is a new mini-language named **staticKanren** whose implementation is available at `https://github.com/lackhoa/staticKanren`.

The layout of my thesis 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.

Readers are recommended to use a text editor with parentheses highlighting for the code. Code listings and likewise programming keywords are written in `monospace` using only ASCII characters. **Boldface** indicates new concepts and emphasis is achieved with *italic*.

There will be some basic examples of relation given throughout the text (usually in the form of comments) to give non-miniKanren users an easier time. However, it is recommended to get familiar with miniKanren first before reading this thesis. Although there is a brief introduction to miniKanren in section 2, it is by no means comprehensive or complete.

There are many stylistic choices in Scheme programming that this thesis adopts:

- The shorthand form for function definition is not used. E.g. we write `(define foo (lambda (x) x))` instead of `(define (foo x) x)`.

- The functions `cons` and `list` are never used and lists are always created with quasi-quotation.

- Finally, associations will always be proper lists instead of pairs. E.g. `([x 1] [y (a b)])` is used instead of `([x 1] [y a b])`.

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, but for readers who are either interested in a more fundamental treatment of logic programming, or just find Scheme macros difficult to comprehend, the implementation in Hemann & Friedman (2013) is perhaps more suitable. Alas, this thesis assume familiarity with Scheme, to which the introduction can be found in Abelson et al. (1996) and Dybvig (2009).

2.1 Basic concepts of miniKanren

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

- **Logic variables**: It is important to distinguish these 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 are simply lazy lists that will yield values only when we ask for them (Abelson et al. 1996). This laziness can be used to represent an infinite number of answers with finite computational resources.

- **States** (also **packages**): State is the most important object in miniKanren. A state is what keeps track of all logical assertions made by the program.

- **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 below.

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

- **Program**: Every miniKanren program consists of a goal wrapped inside of the `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.

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. 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**. This thesis will not describe precisely what the format produced by reification is, but it can be intuitively grasped when more examples are introduced. 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 the following form to get back all answers regardless of how many there are:

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

2.2 Core primitives: equality and fresh variable creation

The most important goal constructor (primitive constraint) is ==. Simply put, == unifies (declares equality of) two terms. 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 ==.

The examples below introduce some uses of ==.

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

⇒ `(_.0)`

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

⇒ `(_.0)`

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

⇒ `()`

In the first and second examples, 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 is also a special syntactic form. Operationally, `fresh` creates one or more fresh logic variables. Syntactically, `fresh` also bind these newly created variables to identifiers within its body.

Let us take a look at some examples involving `fresh`:

```
(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.

Unlike Prolog, we cannot refer to a logic variable not introduced by, or not in the body of `fresh`. For example, the call below fails.

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

```
)
```

The term ”variable” in the error messages refers to naming variables, not logic variables. Technically speaking, logic variables need not even have names. When talking about ”the variable `v`”, we actually mean ”the logic variable bound to the naming variable `v`”. The last example below 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 confusing thing in this example, it becomes useful especially when writing larger programs.

For our purpose, it is important to demystify 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, the value will be used instead of the form `_.n`. Furthermore, if a variable is sure to be a pair, that fact is also made known in

the answer, as the following example shows:

```
(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, is the names of logic variables. The reason for that is due to the name conflict phenomenon mentioned earlier; that is, if we were to simply represent unknown variables by their names, 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, staticKanren uses a simple technique to recover names of unbound variables using birthdate tags.

2.3 Goal combinators: disjunction and conjunction

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 above 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 containing 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 the previous examples. Intuitively, conjunction is implicit whenever two or more goals are written in series in the bodies 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)))
```

2.4 Disequality

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. A small example is given below:

```
(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 that happens to be the case, the program fails, as shown in the following case:

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

⇒ ()

To better understand answers containing disequalities, a more complex example is needed, such as the following:

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

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

The above answer shows see that each disequality constraint store, as presented in the answers, is a list of mini anti-substitution lists. Each anti-substitution list, in turn, 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

3.1 The problem with conde

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

```
(define lookup
  ;; (lookup  x  ([x 1] [y 2] [z 3]))
  ;; => 1
  (lambda (x env)
    (let ([a (car env)])
      (cond
       [(eq? x (lhs a)) (rhs a)]
       [else (lookup x (cdr env))]))))
```

```
(define lookupo
  ;; (run* (x t) (lookupo x  ([x 1] [y 2] [z 3]) t))
  ;; => ((x 1) (y 2) (z 3))
  (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 as follows:

```
(define lookup
  ;; (lookup  x  ([x 1] [y 2]))
  ;; => (x 1)
  ;; (lookup  z  ([x 1] [y 2]))
  ;; => #f
  (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
  ;; (run* (x t bound) (lookupo x  ([x 1] [y 2]) t
   bound?))
  ;; => ((x 1 #t)
  ;;     (y 2 #t)
  ;;     ((_.0 _.1 #f) (=/= ((_.0 x)) ((_.0 y)))))
```

```
  (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 an additional relation argument indicating whether the variable is bound[2]. Meanwhile, we have to devise a special signal for `lookup`, namely `#f`, to indicate the unbound case[3]. However, a problem arises when we actually use the output in `cond`. `case1` and `case1o` demonstrate essentially the same usage of `lookup(o)` (the main point is that these cases are analogous; it does not matter what they actually mean):

```
(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

[2]We could also write another relation that succeeds precisely when lookup fails, but that approach would quickly lead to a dead end, as will be seen later.

[3]There is technically a way to return multiple values in Scheme but it is not too common, particularly due to the verbosity of the syntax.

similarity in appearance, `conde` does not process its clauses from top to bottom like `cond` does, hence 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, such as in `case2(o)` below:

```
(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)])))
```

This time there are three extra denials instead of just the one of `case1o`. The example above is still 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. The next section shows a simple way to deal with this using a technique first implemented in Prolog by Neumerkel & Kral (2016).

3.2 Pseudo-function and condo

miniKanren code looks primitive because all goals are actions manipulating implicit *states*. For more complex combinations to work, there must be another way

for relations to ”communicate”. **Pseudo-functions** achieve this communication by signaling their computed results. However, pseudo-functions receive their results as arguments instead of returning them. To make sense of this concept, the following transformation should be observed (the ”t” at the end of names denote pseudo-functions, 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 returns a goal. The act of *returning* a result is then mimicked by the goal's act of *unifying* that single argument with the result. As an aside, a relation can be made to return whichever of its formal parameters, depending on the circumstance. Examples of this can be seen below:

```
;; From relation
(define conso
  ;; (run* (a d ls) (conso x y))
  ;; => ((_.0 _.1 (_.0 . _.1)))
  (lambda (a d ls)
    (== `(,a . ,d) ls)))
;; To pseudo-function 1
(define cart
  ;; (run* (ls a) ((cart ls) a))
  ;; => (((_.0 . _.1) _.0))
  (lambda (ls)
    (lambda (a)
      (fresh (d) (conso a d ls)))))
```

```
;; To pseudo-function 2
(define cdrt
  ;; (run* (ls d) ((cdrt ls) d))
  ;; => (((_.0 . _.1) _.1))
  (lambda (ls)
    (lambda (d) (fresh (a) (conso a d ls)))))
;; To pseudo-function 3
(define const
  ;; (run* (a d ls) ((const a d) ls))
  ;; => ((_.0 _.1 (_.0 . _.1)))
  (lambda (a d)
    (lambda (ls)
      (conso a d ls))))
```

I have never encountered a situation where this is actually useful as there is generally only one parameter deemed ”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.

Getting back to the problem of conditionals, we first define below two trivial pseudo-functions `truet` and `falset` whose results are `#t` and `#f` respectively.

```
(define truet
  ;; (run* (?) ((truet) ?))
  ;; => (#t)
  (lambda ()
    (lambda (?)
      (== #t ?))))

(define falset
  ;; (run* (?) ((falset) ?))
  ;; => (#f)
```

```
  (lambda ()
    (lambda (?)
      (== #f ?))))
```

Next, we define a single ”primitive” pseudo-function similar to Scheme’s `eq?`[4]. This function relies on both == and =/= to express two branches of the output.

```
(define ==t
  ;; (run* (x y ?) ((==t x y) ?))
  ;; => ((_.0 _.0 #t)
  ;;     ((_.0 _.1 #f) (=/= ((_.0 _.1)))))
  (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) below:

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

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

```
(define =/=t
  ;; (run* (x y ?) ((=/=t x y) ?))
  ;; (((_.0 _.1 #t) (=/= ((_.0 _.1))))
  ;;  (_.0 _.0 #f))
```

[4]Depending on the implementation, more primitives may be defined using additional primitive constraints.

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

Next, we define `conjt` (conjunction) and `disjt` (disjunction) using macro in the following ways:

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

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

The two following examples demonstrate conjunction and disjunction:

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

$\Rightarrow$

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

(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`. We should 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`, as seen below.

```
(define lookupt
  ;; (run* (x t bound?) ((lookupt x  ([x 1]) t) bound?))
  ;; => ((x 1 #t)
  ;;     ((_.0 _.1 #f) (=/= ((_.0 x)))))
  (lambda (x env t)
    (lambda (bound?)
      (conde
       [(== '() env)
        (== #f bound?)]
       [(fresh (y b rest)
          (== `((,y ,b) . ,rest) env)
          (condo
           [(==t y x) (== #t bound?) (== b t)]
           [else
            ((lookupt x rest t) bound?)]))])))))
```

Now, the following 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)]))))
```

Before converting all of your relations to pseudo-functions, there are still some issues to discuss. `condo` works it magic by hiding the control flow. And despite the enhanced readability that this technique brings, there is no simple way to see what is actually going on behind the layer of abstraction. As a result, it is easy to accidentally change the behavior of programs. Take for example the following goal using `condo`:

```
(fresh (x y)
  (condo
   [(==t 'a x) (== 1 y)]
   [(==t 'b x) (== 2 y)]
   [(==t 'c x) (== 3 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` as follows:

```
(fresh (x y)
```

```
(conde
 [(== 'a x) (== 1 y)]
 [(=/= 'a x) (== 'b x) (== 2 y)]
 [(=/= 'a x) (=/= 'b x) (== 'c x) (== 3 y)]))
```

Clearly, the additional denials in these `conde` clauses are redundant since they are already mutually exclusive. Moreover, switching from `conde` to `condo` always means changing the way miniKanren's search work, which might be devastating to some existing programs. As a result, we are once again stuck with lengthy, buggy `conde` clauses whose meaning cannot be easily grasped. The next section addresses this issue by offering a way to verify 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. Sections 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 already-computed answers. On one hand, this is unfortunate since we cannot write non-trivial recursive relations as-is. But on the other hand, this also gives the language a simpler implementation.

staticKanren offers only three primitive constraints: equality, disequality and **fake**.

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, hence we only include disequality because it plays such an important role in many relations.

The core idea of staticKanren is the observation that relational programs' answers are themselves relational programs. And because answers are interpreted as programs, the language must treat variable names more seriously, as seen below:

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

```
⇒ (((4 #(p 0)) () ()))
```

It should be noted that ⇒ by default means "evaluating under staticKanren" in this section. If the above program 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. Remember that terms of the form `#(...)` are vectors, thus the variable bound to `p` is the vector `#(p 0)`[5] (let us ignore the number on the right for now). Besides the walked query, staticKanren always returns two additional constraint stores, one for disequalities and one for fake constraints. Therefore, every answer has the normal shape of a three-element list, which is good for computers and may not be intuitive for humans. The next example demonstrates how the language deals with name conflict:

```
(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).

[5] Variables in miniKanren have always been vectors, but we cannot not see that because variables are always reified to symbols at the end.

Meanwhile, miniKanren sidesteps this problem completely by introducing a new name for every unbound variable appearing in the answer. It is perhaps a subjective matter to debate which representation is better, but there is at least one theoretical problem avoided by the new approach[6]:

```
(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 below demonstrates fake constraints:

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

This is a meaningless program to stress that these fake constraints have no actual meaning. The `fake` form does no more than evaluating its body and adding the resulting expression to the fake constraint store.

```
⇒
(((#(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. The discussion will highlight only those parts containing interesting differences from the simple miniKanren implementation in Byrd (2009)[7].

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

[7] The full implementation is available 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 have a total order on variables defined by `var>?`. A variable is greater than another if its birthdate is less (i.e. it was introduced sooner), or if it has the same birthdate and a lexicographically lesser name. Greater variables are called **seniors** and lesser ones are called **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 concerning states. A state 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. `letg@` is a special form to extract constraints from states.

```
(define-syntax letg@
  ;; let with state inspection
  (syntax-rules (:)
    [(_ (c : s* ...) e)
     (let ([s* (c-> c 's*)] ...) e)]))
(define all-constraints '(S C D F))
(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 laziness, these definitions are trivial 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 (lambda (c* g) (apply mplus (map g c*))))
```

The first interesting function is `unify`, which takes two terms along with a substitution and returns the extended substitution where these two terms are made the same. In the case that they are always different, however, `unify` returns `#f`. There is a new clause in staticKanren to handle the case where both walked terms are variables, which makes sure that the senior is 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))
        (cond
         [(var>? t2 t1) (extend t1 t2 S)]
         [else (extend t2 t1 S)])]
       [(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]))))
```

The new clause (to compare variables) adds a little runtime overhead, but that does not matter in this case because this function simply is not used at runtime. There is a method to achieve the same effect without modifying `unify` by tweaking the reification process instead, which should eliminate the runtime overhead. However, doing it in this way keeps the substitution list clean and easy to work with. This cleanliness is helpful in section 4.4 where we have to modify the substitution list after reification.

Figures 1 and 2 show both miniKanren's and staticKanren's representations of the same variable-only substitution. Each vertex corresponds to a variable and each edge from vertex `x` to vertex `y` corresponds to an association with lhs `x` and rhs `y`.

Now we are ready to define user-level functions, starting with the fake goal constructor. It should be remembered that `lambdag@` is a `lambda` form with the ability to extract various constraints from its argument, which is always a state.

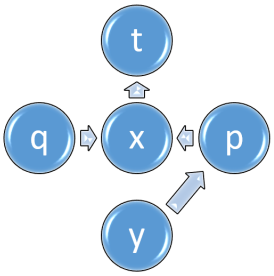

Figure 1: miniKanren's substitution

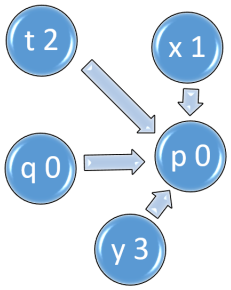

Figure 2: staticKanren's substitution

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

The job of `fresh` is broken down to a conjunction (`conj`) which strings goals together and a `letv` which creates variables. `letv` extracts the date counter from its input state and assigns it to the new variables. The counter is then incremented before getting fed to the goal in its body.

```
(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)))))]))
```

Other goal constructors do not need to be modified. Surprisingly, reification can be made simpler despite the fact that unknown variables get to keep their names. To obtain the answers, `reify` only needs to deep walk the query and the constraint stores in `c`. A more compact representation of disequality is then obtained

by removing constraints containing either no variable or irrelevant variables (i.e. variables absent from both deep walked `q*` and deep walked `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)))))))
```

Some helpers of `reify` are given below. It is worth noting that miniKanren data are also miniKanren terms, which explains why the code can be so short. The function performing subsumption check of disequality stores, `rem-subsumed`, remains unchanged.

```
(define-syntax case-term
  ;; A type dispatcher for mk terms
  (syntax-rules ()
    [(_ e [v e1] [(a d) e2] [atom e3])
     (let ([term e])
       (cond
        [(var? term) (let ([v term]) e1)]
        [(pair? term)
          (let ([a (car term)] [d (cdr term)]) e2)]
        [else (let ([atom term]) e3)]))]))

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

```
                        (has-iv? d R))))
            D)))

(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]))))
```

Finally, the definition of `run*` is as follows:

```
(define-syntax run*
  (syntax-rules ()
    [(_ (q q* ...) g g* ...)
     ((fresh (q q* ...)
        g g* ...
        (finalize `(,q ,q* ...)))
      init-c)]))

(define finalize
  (lambda (qs)
    (lambdag@ (final-c)
      (unit (reify final-c qs)))))
```

This concludes the core implementation of staticKanren.

4.3 Returning substitutions in answers

Let us begin exploring the practical value of staticKanren with a simple improvement: returning substitutions instead of walked queries in the answers. We achieve

this with the new `run*su` form. For instance, the answers in the next call would be `((#(p 0) #(p 0) #(p 0)) () ())` if we replaced `run*su` by `run*`.

```
(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 once again highlights the fact that miniKanren states are just accumulations of logical assertions throughout the execution of programs, albeit normalized. By looking at the answers, we can summarize the content of the program that produced them. 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 involved, but in fact this feature only requires a simple addition to `run*`. We can retrieve a substitution simply by unifying the query with the walked version of itself, which is what would be returned by `run*`. The definition `run*su` is given below:

```
(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)
    (let ([t (car c)])
      `(,(unify t qs init-S)
        .
        ,(cdr c)))))
```

This much of development already lets us to obtain the ”negative” side of the `lookupt` function defined earlier in Section 3. Before running it in staticKanren, however, we need to first redefine `lookupt`, so that the recursive call is fake as follows.

```
(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 lookup is to fail, the variable `x` must not match the association's lhs and the lookup must also fail for the rest of the environment.

4.4 Obtaining common unification with anti-unification

There are still cases where rewritten programs do not look at all similar to the original. Take `membero` for example:

```
(define membero
  ;; (run* (x) (membero x  (a b c)))
  ;; (without fake goal) => (a b c)
  (lambda (x ees)
    (fresh (e es)
      (== ees `(,e . ,es))
      (condo
       [(==t x e) succeed]
       [else (fake `(membero ,x ,es))]))))
```

Rewriting through `run*su` gives us the following:

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

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

The answer above 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 the version above does not look at all natural: it does not show the common structure of `ees` in both clauses. To make this notion of commonness clear, we introduce **anti-unification**, 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 patterns over symbolic formulae, which is exactly what we need for this purpose. The following is an example of anti-unification:

```
(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 shall be referred to as **pattern variables**. All pattern variables have the birthdate `0.5`, the reason for which will become clear when the program is com-

plete.

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

```
(define anti-unify
  (lambda (t*)
    (let-values
        ([(res _iS)
          (let au ([t* t*] [iS '()])
            (cond
             [;; rule 7: eq? deals with variables too
              (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 t* new-var
  iS)))]))])
      res)))
```

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

(define teq?
  ;; Compares two mk terms
  (lambda (t1 t2)
    (or (eq? t1 t2)
        (and (pair? t1) (pair? t2)
             (teq? (car t1) (car t2))
             (teq? (cdr t1) (cdr t2))))))

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

Even though anti-unification is the heart of our solution, there is still a lot of work to do. It is not clear how to convert the obtained pattern into a usable program. We start with a similar definition to 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 as input. First, `au-extract` extracts a common pattern `au` from the walked query `t*`. Second, it unifies the query with the pattern in the empty environment,

resulting in the substitution `uS`. The next obvious step is to unify `au` back into each of the walked queries in `t*` to obtain the substitutions `S*`. Finally, we perform the optional step of cleaning up the substitutions with `purify-S` and deep walk the other constraints, obtaining a working program. This is shown below:

```
(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))))

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

The only question left is how to clean up the substitutions. We have already done a similar thing during reification with the help of `walk*`. However, this time things are not so convenient since we also want to retain the intermediate associations

given by anti-unification. There is a basic principle that works well: an association is redundant iff its lhs (a variable) was not introduced prior to a certain date. The intuitive reason is that we can just use the rhs in the first place without ever introducing the lhs.

Conveniently, `purify-S` can keep track of already introduced variables using their birthdates. Associations whose lhs was born later than a certain date (called the **cutoff date**) are filtered out of the substitution list. We need only look at the lhs because it is always the junior variable in the case that both sides are variables. The definition of `purify-S` is as follows:

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

Looking back, we see that `purify-S` is called twice in `au-extract` and `au-helper`. The first call is to clean up the common substitution involving query variables and pattern variables, in which case the cutoff date only allows query variables (note that `init-C` is also the birthdate of all query variables). In the second call (in `au-helper`) which involves case-specific substitutions, we also include pattern variables since they have already been introduced. This explains why pattern variables are born on `0.5`: they are junior to query variables, but senior to all variables introduced by the ”user” (i.e. variables coming from `fresh`). Figure 3 illustrates the idea.

The implementation is now complete and we can start using `run*au` to rewrite programs. The call below will return `lookupo`, obtained by forcing `lookupt` to ”return” `#t`. Notice that this `lookupt` is the same modified version in section 4.3.

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

$\Rightarrow$

```
(((#(env 0)
```

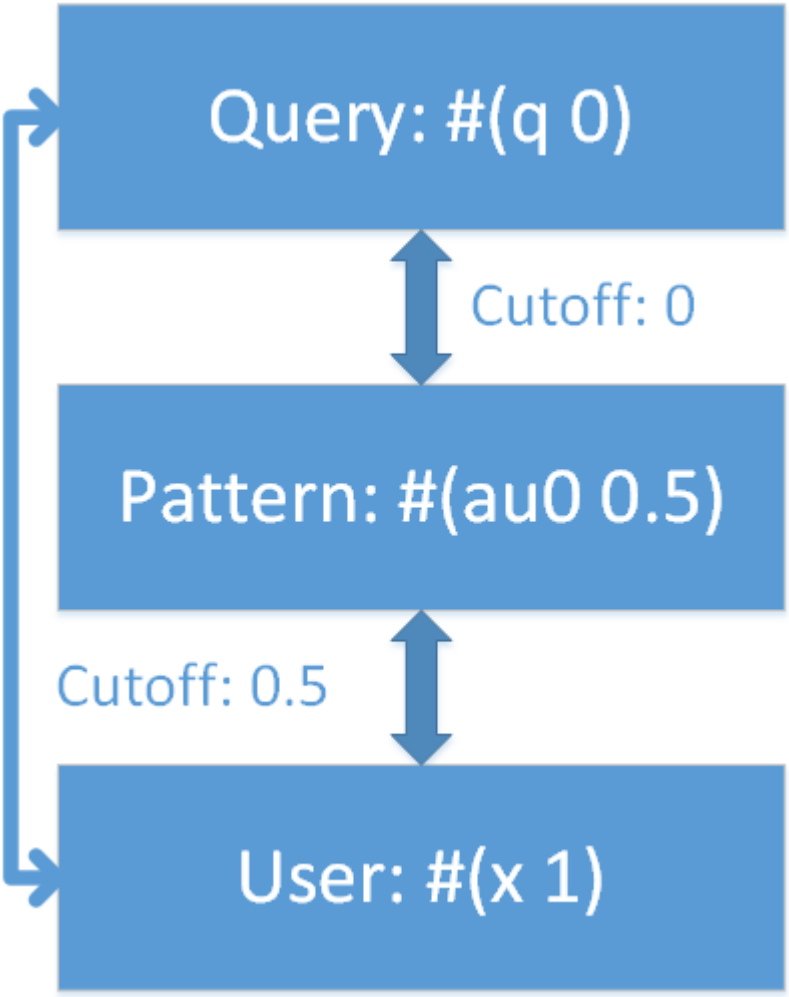

Figure 3: The interaction between query, pattern and ”user” variables

```
   ((#(au0 0.5) #(au1 0.5)) . #(rest 1))))
 ((((#(au1 0.5) #(t 0))
    (#(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) (== t au1)]
   [(=/= au0 x)
    ((lookupt x rest t) #t)]))
```

Anti-unification identifies that across all clauses, the environment `env` is always a non-empty list with the first element being an association. It also correctly handles what 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 lookup is to succeed, the lookup of `x` in the tail of the environment has to succeed.

To aid in understanding what the role of `purify-S` is in the example above, here is the same answer but with both calls to `purify-S` removed:

```
(((#(env 0)
   ((#(au0 0.5) #(au1 0.5)) . #(rest 1))))
 ((((#(au1 0.5) #(t 0))
    (#(au0 0.5) #(x 0)))
   ()
   ())
  (((#(b 1) #(au1 0.5))
    (#(y 1) #(au0 0.5)))
   (((#(au0 0.5) #(x 0))))
   (((lookupt #(x 0) #(rest 1) #(t 0))
     #t)))))
```

This is akin to the goal:

```
(fresh (au0 au1)
  (== `((,au0 ,au1) . ,rest) env)
  (conde
   [(== au0 x) (== au1 t)]
```

```
[(== b au1) (== y au0)
 (=/= au0 x)
 ((lookupt x rest t) #t)]))
```

The two ==’s involving `b` and `y` are not necessary and were removed by the second call to `purify-S` in the original answer due to the fact that `b` and `y` were born after `AU-BD` (0.5).

5 ACKNOWLEDGEMENTS

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

6 CONCLUSION

The goal of this thesis was to find a way to write conditionals in miniKanren. This resulted in the following contributions:

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

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

3. A way of reformatting staticKanren’s answers to a more human-like style.

We learned along the way the important fact that while conditionals can be easily expressed with `condo`, the behavior of existing programs are almost impossible to preserve. The best solution, as presented in this thesis, is to specify a more compact, Scheme-like version with `condo`, run it through staticKanren for normal-

ization and then manually modify the generated result, if needed, for the final version.

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 user flexibility. An alternative approach is to use constructive negation (Chan 1989) but due to the workload required it is not suitable for a Bachelor thesis. Besides, creating yet another version of miniKanren would dilute the already diluted attention of newcomers since there are so many versions available at the moment.

There is another interesting use case of staticKanren to be explored in future work. With the help of `condo`, it is simple to rewrite any Scheme program to staticKanren and extract its various modes for verification and analysis. Additionally, the idea of pseudo-functions might be useful for exploring many other functional programming features, the most obvious instance of which being function composition.

REFERENCES

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

Dybvig, R. 2009. The Scheme programming language. Cambridge: MIT Press.

Friedman, D., Byrd, W., Kiselyov, O. & J., H. 2018. The reasoned Schemer. 2nd ed. Cambridge: 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. Upper Saddle River: Prentice Hall.