metaKanren: Towards a Metacircular Relational Interpreter

BHARATHI RAMANA JOSHI, IIIT Hyderabad, India
WILLIAM E. BYRD, University of Alabama at Birmingham, USA

We motivate the need for a deep relational interpreter for a miniKanren language, and then present metaKanren, a μKanren-like language for which we have implemented a deep relational interpreter in miniKanren. We demonstrate this relational interpreter running "backwards" to synthesize metaKanren code and then provide a commented tour of our relational interpreter for metaKanren. Finally, we show how metaKanren can serve as a useful starting point towards a metacircular interpreter for a miniKanren language, and miniKanren program synthesis.

Additional Key Words and Phrases: metaKanren, miniKanren, metacircularity, relational programming, interpreters, program synthesis, relational interpreters

ACM Reference Format:
Bharathi Ramana Joshi and William E. Byrd. 2021. metaKanren: Towards a Metacircular Relational Interpreter. 1, 1 (August 2021), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Metacircular interpreters, interpreters written in the language being interpreted itself, elucidate on the semantics and demonstrate non-trivial usage of the language [Reynolds 1972]. Relational interpreters [Byrd et al. 2012], interpreters written in a relational language like miniKanren [Byrd 2010], have been used to solve several interesting problems [Byrd et al. 2017] and by their very nature are capable of being run "backwards" to synthesize parts of the interpreted language. We unify these two fascinating ideas, and thus ask: can we have a relational metacircular interpreter for miniKanren, which would enable miniKanren program synthesis?

1.1 A First Attempt: Shallow Embedding

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

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

Authors' addresses: Bharathi Ramana Joshi, IIIT Hyderabad, India, bharathi.joshi@research.iiit.ac.in; William E. Byrd, University of Alabama at Birmingham, USA, webyrd@uab.edu.

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

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

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

⟨*value*⟩ ::= Any valid Racket symbol

⟨*id*⟩ ::= Any valid Racket symbol

Term expressions evaluate to terms, whose grammar is:

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

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

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

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

The corresponding shallow relational self-interpreter[1] implemented using faster-miniKanren's Racket port [Ballantyne 2021] is:[2]

```
#lang racket

(require "../faster-miniKanren/mk.rkt")

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

; Evaluate goal expression in an environment
(define (eval-gexpro expr env)
    (conde
      [(fresh (e1 e2 t)
         (== `(== ,e1 ,e2) expr)
         ; Evaluation of both terms should unify to the same term t
```

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

[2]This code is available here https://github.com/iambrj/metaKanren/blob/master/shallow.rkt.

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

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

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

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

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

⇒

```
(cat)
```

[3]Code for all examples in the shallow interpreter is available here https://github.com/iambrj/metaKanren/blob/master/shallow-examples.rkt

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

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

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

⇒

```
('cat z)
```

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

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

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

⇒

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

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

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

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

1.2 metaKanren

In this paper we present metaKanren, a μKanren-like language for which we have implemented a deep relational interpreter in miniKanren. We believe this is a tiny, but right, step in the endeavor to synthesize miniKanren programs. Concretely, our contributions are:

- We present the metaKanren language (Section 2), which is μKanren extended to support user-defined relations and delaying goals.
- We demonstrate a deep relational interpreter for metaKanren performing program synthesis by running it "backwards" (Section 3).
- We show how defunctionalization of μKanren can be used to implement a stream-passing relational interpreter for metaKanren (Section 4).
- We show how metaKanren can serve as a useful starting point towards a metacircular interpreter for miniKanren and synthesizing miniKanren programs (Section 5).
- We show how the metaKanren interpreter is different from other current deep relational self-interpreters (Section 6).

2 THE METAKANREN LANGUAGE

The metaKanren language is μKanren with three new constructs and a different representation for query numbers. We direct readers unfamiliar with μKanren to Hemann and Friedman [2013], familiar readers may skip the grammar and go to Section 2.1 where we discuss these differences.

```
⟨mmk-program⟩  ::= (run* ⟨id⟩ ⟨goal-expr⟩)
                 |  (run ⟨peano⟩ ⟨id⟩ ⟨goal-expr⟩)
⟨goal-expr⟩  ::= (disj ⟨goal-expr⟩ ⟨goal-expr⟩)
               |  (conj ⟨goal-expr⟩ ⟨goal-expr⟩)
               |  (fresh (⟨id⟩) ⟨goal-expr⟩)
               |  (== ⟨term-expr⟩ ⟨term-expr⟩)
               |  (letrec-rel ((⟨id⟩ (⟨id⟩ ...) ⟨goal-expr⟩)) ⟨goal-expr⟩)
               |  (call-rel ⟨id⟩ ⟨term-expr⟩ ...)
               |  (delay ⟨goal-expr⟩)
⟨term-expr⟩  ::= (quote ⟨value⟩)
               |  ⟨id⟩
               |  (cons ⟨term-expr⟩ ⟨term-expr⟩)
⟨value⟩  ::= ⟨number⟩
          |  ⟨boolean⟩
          |  ⟨symbol⟩ [other than 'var']
          |  ()
          |  (⟨value⟩ . ⟨value⟩)
⟨boolean⟩  ::= #t | #f
⟨number⟩ ::= Any number satisfying numbero

⟨lexical-var⟩ ::= Any valid Racket symbol

⟨logic-var⟩ ::= (var . ⟨peano⟩)

⟨peano⟩ ::= () | (⟨peano⟩)
```

In addition to this we have a separate term language, consisting of values logic variables can be unified with.

```
⟨term⟩  ::= ⟨logic-var⟩
          |  ⟨number⟩
          |  ⟨boolean⟩
          |  ⟨symbol⟩ [other than ’var’]
          |  ()
          |  (⟨term⟩ . ⟨term⟩)
```

2.1 Differences

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

```
(run 3 (x)
  (letrec ((fives-and-sixes
             (lambda (x)
               (disj (== x 5)
                     (disj (== x 6)
                           (lambda (s/c)
                             (lambda ()
                               ((fives-and-sixes x) s/c)))))))))
    (fives-and-sixes x)))
```

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

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

   We refer to natural numbers in the above representation as **Peano numerals**. Thus, the first change we make is replacing 3 by the Peano numeral `((()))`:
   ```
   (run ((())) (x)
     (letrec ((fives-and-sixes
                (lambda (x)
                  (disj (== x 5)
                        (disj (== x 6)
                              (lambda (s/c)
                                (lambda ()
                                  ((fives-and-sixes x) s/c)))))))))
       (fives-or-sixes x)))
   ```
(2) Next, we note that the relation uses the interpreting language’s `letrec` and `lambda` forms to define a recursive relation and procedure application to use this relation. Once again, doing this would not be possible with miniKanren as the interpreting language because it

does not have any of `letrec`, `lambda`, or procedure application. Therefore, we introduce a new language construct, `letrec-rel` (see syntax 2), to define a recursive relation and another, `call-rel` (see syntax 2), to use the defined recursive relation in metaKanren. Thus, by rewriting the program using these, we end up with:

```
(run ((((())))) (z)
  (letrec-rel ((fives-or-sixes (x)
                 (disj (== x 5)
                       (disj (== x 6)
                             (lambda () (call-rel fives-or-sixes x))))))
    (call-rel fives-or-sixes z)))
```

(3) Finally, we note that a `lambda` is still being used to delay goal evaluation in the recursive call to `fives-or-sixes`. For the same reason as above, we introduce a new construct `delay` to delay goal evaluation in metaKanren.
With these changes, we finally arrive at the following valid metaKanren program:

```
(run ((((())))) (z)
  (letrec-rel ((fives-or-sixes (x)
                 (disj (== x 5)
                       (disj (== x 6)
                             (delay (call-rel fives-or-sixes x))))))
    (call-rel fives-or-sixes z)))
```

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

3 METAKANREN SYNTHESIS EXAMPLES

Our interpreter is the two-argument relation `eval-programo`, where the first argument is a quoted `run`/`run*` expression and the second argument the list of answers. We use the Racket procedure peano (provided in Appendix A) to convert Arabic numerals to Peano numerals (see 1) to improve the queries' readability. Section 4 is a commented tour of the interpreter itself, which is not a prerequisite to understand the examples here.

Of course our relational interpreter is capable of running metaKanren programs "forwards", i.e. evaluate a `run`/`run*` expression to generate a list of values for the query's logic variable satisfying the given constraints. For example, consider the following run of appendo:[4]

```
(run* (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((appendo (l1 l2 l)
                      (disj
                        (conj (=='() l1) (== l2 l))
                        (fresh (a)
                          (fresh (d)
                            (fresh (l3)
                              (conj (== (cons a d) l1)
                                    (conj (== (cons a l3) l)
                                          (delay (call-rel appendo d
```

[4]Code for all examples in this section is available here https://github.com/iambrj/metaKanren/blob/master/paper-examples.rkt

```
                                                                          l2
                                                                          l3))))))))))
         (call-rel appendo '(1 2) '(3 4) z)))
    x))
⇒
(((1 2 3 4)))
```

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

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

3.1 Synthesis with nested run*

Here we try to make the metaKanren interpreter synthesize conj by providing the expected output and placing a logic variable in the inner query's goal expression:

```
(run* (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((appendo (l1 l2 l)
                      (disj
                        (conj (=='() l1) (== l2 l))
                        (fresh (a)
                          (fresh (d)
                            (fresh (l3)
                              (,x (== (cons a d) l1)
                                  (conj (== (cons a l3) l)
                                        (delay (call-rel appendo d
                                                                 l2
                                                                 l3))))))))))
         (call-rel appendo '(1 2) '(3 4) '(1 2 3 4))))
    '((_.))))
⇒
(conj)
```

And metaKanren indeed synthesizes it! The nested run*s prove that conj is the *only* possible metaKanren fragment that x can be.

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

```
(run* (x)
  (eval-programo
    `(run ,(peano 1) (z)
       (letrec-rel ((appendo (l1 l2 l)
                      (disj
                        (conj (=='() l1) (== l2 l))
                        (fresh (a)
                          (fresh (d)
                            (fresh (l3)
                              (,x (== (cons a d) l1)
                                  (conj (== (cons a l3) l)
                                        (delay (call-rel appendo d
                                                                 l2
                                                                 l3))))))))))
         (call-rel appendo '(1 2) '(3 4) '(1 2 3 4))))
    '((_.))))
⇒
(disj conj)
```

We get disj in addition to conj. Although this answer may seem puzzling, it is in fact a correct answer for a run 1 inner query, because the run 1 semantics dictate that *only one of the answers* should be (_.) — which it is, when both the branches of the disjunction succeed! In other words, we also get back a disj because when both the branches succeed, disj has the same semantics as conj. But when we have run* for the inner query, the semantics dictate that the *only answer* should be (_.). In this case, disj is not a valid answer for x, because the inner query has more answers than just a single (_.) when x is disj.

This demonstrates the additional expressivity a deep embedding brings to the table that a shallow embedding lacks (see Section 1.1).

3.2 Synthesizing parts of a user-defined relation

Instead of replacing the goal constructor conj with a logic variable, we get another interesting example if we were to replace the arguments to appendo in the recursive call in its definition with logic variables instead (i.e. provide a partially defined appendo with expected output, and synthesize the left out parts).

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

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

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

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

```
(run 1 (e1 e2)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== ,e1 ,e2)))
         (call-rel five 5)))
    '((_.)))
⇒
'(((_.0 _.0) (num _.0)))
```

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

```
(run 1 (x)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== 7 7)))
         (call-rel five 'elephant)))
    x)
⇒
'(((_.))
```

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

```
(run 3 (e1 e2)
  (eval-programo
    `(run* (z)
       (letrec-rel ((five (f)
                      (== ,e1 ,e2)))
         (call-rel five 'elephant)))
    '((_.)))
⇒
'(((_.0 _.0) (num _.0)) (() ()) (5 f))
```

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

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

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

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

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

And we get what we expect!

In addition to external grounding, here we also:

(1) provide two more examples as to how appendo should behave to avoid overfitting.
(2) provide additional hints in the form of symbolo constraints to make the query return faster. Despite these extra hints, it takes the current naive interpreter a few minutes to synthesize the arguments. Therefore, improving the interpreter's performance is crucial for any serious synthesis tasks (see Section 5).

3.3 Counting answers

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

```
(run* (count)
  (eval-programo
    `(run ,count (z)
       (disj (== z 1)
             (== z 2)))
    '(1 2)))
⇒
((((())) (((_.0)) (=/= ((_.0 ()))))))
```

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

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

```
(run* (count answers)
  (eval-programo`(run ,count (z)
                   (disj (== z 1)
                         (== z 2)))
    answers))
```

⇒

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

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

4 THE METAKANREN INTERPRETER

The problems to be solved to implement metaKanren are:

(1) Choosing a representation for logic variables that can be manipulated relationally.
(2) Defunctionalizing goals.
(3) Defunctionalizing streams.
(4) Defunctionalizing user-defined relations.

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

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

This section is a commented tour of our interpreter for metaKanren. We assume familiarity with μKanren and relational interpreters. We direct readers unfamiliar with μKanren to Hemann and Friedman [2013] and relational interpreters to Byrd et al. [2012].

We break down our metaKanren relational interpreter into three relational sub-interpreters, one each for goal expressions, term expressions, and programs. Each one plays a specific well-defined role in correspondence with metaKanren's grammar and they call one another when needed, which makes the entire interpreter as a whole easier to understand and modify as required. Most of the code is included in the text, except for tedious parts which are in Appendix A.[5]

4.1 Logic variables

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

```
; Succeed if p is a peano numeral, fail otherwise
(define (peanoo p)
```

[5]Code for the entire metaKanren interpreter is available here https://github.com/iambrj/metaKanren/blob/master/metaKanren.rkt

```
  (conde
    [(== '() p)]
    [(fresh (p1)
       (== `(,p1) p)
       (peanoo p1))]))

; Succeed if x is a logic variable, fail otherwise
(define (var?o x)
  (fresh (val)
    (== `(var . ,val) x)
    (peanoo val)))

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

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

4.2 Term interpreter

Term expressions are those that occur as arguments to == (see Section 2). The goal interpreter calls the term expression interpreter when it encounters a == goal to evaluate both the arguments. Terms may contain lexical variables (represented as symbols in the interpreting language), thus the term interpreter takes an environment (represented as an association list) for evaluating terms so that it may look up lexical variables in the environment. For atomic values (such as booleans and numbers) and quoted terms, the interpreter simply unifies them with the output logic variable as they are self evaluating. For lists, it evaluates the `car` and `cdr` of the list recursively.

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

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

4.3 States

Just as in μKanren, a state in metaKanren is represented as a cons pair of a substitution and a counter (represented as a Peano numeral). We use association lists for substitutions and Peano numerals for counters. For example, the initial state is (()) (the value of the expression (cons '() '())), which would have been (() . 0) in μKanren. Goals are applied to states to generate streams. The goal interpreter (discussed in Section 4.6) performs the application.

The value of a logic variable (if any) is searched in the substitution of a state while performing unification via the walk operation. Searching for terms other than a logic variable result in the term itself.

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

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

; Unify the value of the term u in the substitution s with v
(define (walko u s v)
  (conde
    [(== u v)
     (conde
       [(symbolo u) (== u v)]
       [(numbero u) (== u v)]
       [(booleano u) (== u v)]
       [(== '() u) (== u v)])]
    [(fresh (a d)
       (== `(,a . ,d) u)
       (=/= a 'var)
       (== u v))]
    [(var?o u)
     (conde
       [(== u v) (not-assp-subo u s)]
       [(fresh (pr-d)
          (assp-subo u s `(,u . ,pr-d))
          (walko pr-d s v))])]))
```

Substitutions are extended by adding the logic variable and its value.

```
(define (ext-so u v s s1)
  (== `((,u . ,v) . ,s) s1))
```

4.4 Unification

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

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

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

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

4.5 Streams & search

μKanren uses procedures to represent immature streams and evaluates them to pull states out of the stream, which is not possible to do with metaKanren. Instead, we use lists with various tags and have `pullo`, a first-order relational translation of μKanren's `pull`, extract values from the stream (discussed below). The tag of the stream represents the action that is delayed so that it may be performed when pulling states from the stream. The structures of various streams are shown in Table 1. Details of various streams follow.

1. **Empty stream/mzero** : This is generated when a goal fails. For example, `(== 1 2)` generates `mzero` (irrespective of the state to which it is applied). We represent `mzero` using the empty list.
2. **Mature** : This is generated when a goal succeeds with at least one answer (possibly followed by a delayed stream). For example, `(disj (== 1 1) (== 2 2))` generates `(state state)`, where `state` is the state to which the goal is applied, since both branches of the disjunction succeed. Similarly, `(disj (== 1 1) (delay (== 2 2)))` generates
   ```
   (state (delayed eval (== 2 2) state env))
   ```
   where `env` is the lexical environment to which the goal is applied. Note that the delayed goal generates a delayed stream that follows the result of the first goal.
3. **Delayed eval** : This is generated when a goal expression evaluation is delayed. For example, `(delay (== 1 1))` generates `(delayed eval (== 1 1) state env)`. It saves the goal expression, the state, and the lexical environment so that the evaluation may be performed later when pulling states.
4. **Delayed mplus**: This is generated when one of the goals in a disjunction generates a delayed stream. A `delayed mplus` stream saves both of its argument streams so that the `mplus` can be performed later when pulling states.
   For example, `(disj (delay (== 1 1)) (== 2 2))` generates the stream
   ```
   (delayed mplus (delayed eval (== 1 1) state env)
                  state)
   ```
5. **Delayed bind**: This is generated when the first goal in a conjunction generates a delayed stream. As with `delayed mplus`, a `delayed-bind` stream saves all of its arguments so that the `bind` may be performed later when pulling states.
   For example, `(conj (delay (== 1 1)) (== 2 2))` generates the stream
   ```
   (delayed bind (delayed eval (== 1 1) state env)
                 (== 2 2)
                 env)
   ```

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

Table 1. Structures of various streams

We employ an interleaving search strategy similar to μKanren encoded using `mpluso` and `bindo`, the first-order relational counterparts to μKanren's `mplus` and `bind`. For empty and mature streams, `mpluso` and `bindo` have a nearly identical implementation to μKanren. However, for any of the delayed streams, they simply delay it further by saving the appropriate parameters.

```
(define mzero '())

(define (mpluso $1 $2 $)
  (conde
    [(== '() $1) (== $2 $)]
    [(fresh (a d r1)
       (== `(,a . ,d) $1)
       (=/= 'delayed a)
       (== `(,a . ,r1) $)
       (mpluso d $2 r1))]
    [(fresh (d)
       (== `(delayed . ,d) $1)
       (== `(delayed mplus ,$1 ,$2) $))]))

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

`pullo` is the key component in interleaving the search. If the input stream is either empty or a mature stream, it simply unifies it with the output stream. However, when it is passed a delayed stream, it performs the action appropriate for the delay tag. For example, if it were passed a `delayed eval` stream, it will call the goal expression interpreter (see Section 4.6) on the saved parameters and recursively pull on the resulting stream.

```
(define (pullo $ $1)
  (conde
    [(== '() $) (== '() $1)]
    [(fresh (a d)
       (== `(,a . ,d) $)
       (== $ $1)
       (=/= 'delayed a))]
    [(fresh (ge s/c env $2)
       (== `(delayed eval ,ge ,s/c ,env) $)
       (eval-gexpro ge s/c env $2)
       (pullo $2 $1))]
    [(fresh ($a $b $a1 $2)
       (== `(delayed mplus ,$a ,$b) $)
       (pullo $a $a1)
       ; Note the interleaving
       (mpluso $b $a1 $2)
       (pullo $2 $1))]
    [(fresh (saved-ge saved-env saved-$ saved-$1 $2)
       (== `(delayed bind ,saved-$ ,saved-ge ,saved-env) $)
       (pullo saved-$ saved-$1)
       (bindo saved-$1 saved-ge saved-env $2)
       (pullo $2 $1))]))
```

4.6 Goal interpreter

μKanren uses procedures to represent goals, which is again not possible to do with metaKanren since miniKanren does not have procedures. Instead, we use a list of quoted datum to represent a goal. For example, we represent the goal `(== 2 3)` as `(== 2 3)`. This also has the advantage of making goals inspectable and thus helps in debugging metaKanren programs. As in μKanren, goals are applied to a state and produce a stream of states, and the goal interpreter performs this application. However, the metaKanren interpreter also has to pass around a lexical environment to manage bindings introduced via `letrec-rel` and `fresh`.

Delayed goals produce a `delayed eval` stream with the parameters saved, so that the application may be performed later when required. Disjunctions and conjunctions are translated into the appropriate calls to `mpluso` and `bindo` respectively. The `fresh` goal is handled by extended the lexical environment by binding the interpreting logic variable to the current counter, and recursively evaluating the body in the extended environment with an incremented counter in the state. The == goal is handled by calling the term interpreter (see Section 4.2) to evaluate the two terms, and then unifying them (see Section 4.4).

To handle user-defined relations, the goal interpreter extends the environment by binding the identifier of the user-defined relation to a closure with the parameters, the right hand side and the enclosing environment and recursively evaluates the body of the `letrec-rel`. To apply a user-defined relation, it looks up the identifier in the environment to fetch a closure, and then recursively evaluates the body after extending the environment with the values for the parameters. The implementation of user-defined relations is tangential to the relational components of metaKanren, we refer the reader to Byrd et al. [2012] for a thorough exposition of implementing such features.

```
(define (eval-gexpro expr s/c env $)
  (conde
    [(fresh (ge)
       (== `(delay ,ge) expr)
       (== `(delayed eval ,ge ,s/c ,env) $))]
    [(fresh (ge1 ge2 ge1-$ ge2-$)
       (== `(disj ,ge1 ,ge2) expr)
       (eval-gexpro ge1 s/c env ge1-$)
       (eval-gexpro ge2 s/c env ge2-$)
       (mpluso ge1-$ ge2-$ $))]
    [(fresh (ge1 ge2 ge1-$)
       (== `(conj ,ge1 ,ge2) expr)
       (eval-gexpro ge1 s/c env ge1-$)
       (bindo ge1-$ ge2 env $))]
    [(fresh (x ge s c env1)
       (== `(fresh (,x) ,ge) expr)
       (== `(,s . ,c) s/c)
       (exto `(,x) `((var . ,c)) env env1)
       (eval-gexpro ge `(,s . (,c)) env1 $))]
    [(fresh (te1 te2 v1 v2 s c s1)
       (== `(== ,te1 ,te2) expr)
       (== `(,s . ,c) s/c)
       (eval-texpro te1 env v1)
       (eval-texpro te2 env v2)
       (conde
         [(== #f s1) (== '() $)]
         [(=/= #f s1) (== `((,s1 . ,c)) $)])
       (unifyo v1 v2 s s1))]
    [(fresh (id params geb ge e1)
       (== `(letrec-rel ((,id ,params ,geb)) ,ge) expr)
       (== `((rec (closr ,id ,params ,geb)) . ,env) e1)
       (eval-gexpro ge s/c e1 $))]
    [(fresh (id args params geb env1 ext-env vargs)
       (== `(call-rel ,id . ,args) expr)
       (lookupo id env `(closr ,params ,geb ,env1))
       (eval-args args env vargs)
       (exto params vargs env1 ext-env)
       (eval-gexpro geb s/c ext-env $))]))
```

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

4.7 Recovering reification

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

take-no and take-allo try to extract the first n and all states from a stream respectively, by using pullo (see Section 4.5) to extract one state at a time.

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

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

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

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

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

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

```
(define (reify-so v-unwalked s s1)
  (fresh (v)
    (walko v-unwalked s v)
    (conde
      [(var?o v)
```

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

4.8 Program interpreter

The program interpreter calls the goal interpreter by wrapping a `fresh` corresponding to the logic variable in the `run` query, pulls as many states as need from the resulting stream, and then reifies these pulled states. See Section 3 for examples that use `eval-programo`.

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

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

5 FUTURE WORK

There are two directions of future work, corresponding to two of our initial goals (metacircularity and program synthesis), evident from our experiments with metaKanren.

5.1 Towards metacircularity

As of now, metaKanren uses four constraints that it does not implement, namely `=/=`, `absento`, and data type constraints `numbero` & `symbolo`. The framework described in Hemann and Friedman [2017] can be used for implementing these. Furthermore, `letrec-rel` must also be extended to handle mutual recursion, since some of the relations used to implement metaKanren are themselves mutually recursive. Thus, to make it metacircular, all of these must be implemented.

Finally, metaKanren is implemented using macros most miniKanren languages provide to facilitate a more convenient syntax (e.g. `conde`). A relational parser must be implemented to play the role of these macros.

5.2 Towards miniKanren synthesis

As mentioned in synthesis example 3.2, the current naive interpreter takes a couple of minutes to synthesize something as simple as the arguments to a recursive call. This means that to perform synthesis of any interesting miniKanren programs, a staggering boost in performance is inescapable. Three particular strategies that can be employed to improve metaKanren's performance are:

(1) **Staging**: The current tower of interpreters is shown in Table 2. The interpretive overhead can be removed by transforming the interpreter into a single-pass compiler by using staging techniques [Amin and Rompf 2017].

metaKanren runs on …
miniKanren runs on …
Racket

Table 2. Tower of interpreters

(2) **Barliman techniques**: Several techniques have been developed and incorporated into the Barliman smart editor [Ballantyne et al. 2021] to perform real-time program synthesis of programs as complex as `append` in a relational Scheme interpreter. Some of these techniques can be useful to speed up the metaKanren interpreter as well.

(3) **Guiding the search**: We have used faster-miniKanren [Ballantyne 2021] to run all our metaKanren experiments. Using a guided search strategy, such as the one described in Zhang et al. [2018], can also improve performance.

6 RELATED WORK

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

On the other hand, Ballantyne [2016] uses non-relational features to implement logic variables, making it impossible to make the interpreter metacircular by simply implementing more features. Furthermore, they do not implement a `run*` for the interpreted language which limits the expressivity during synthesis as discussed in Section 1.1.

7 ACKNOWLEDGEMENTS

We thank Talia Ringer for starting the SIGPLAN mentoring initiative (without which this collaboration would not have happened), Siddharth Bhat for suggesting the answer counting example, Gregory Rosenblatt for suggesting improvements to the syntax, and the anonymous reviewers for their valuable feedback.

Research reported in this publication was supported by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

Nada Amin and Tiark Rompf. 2017. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

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

Michael Ballantyne. 2021. faster-miniKanren. https://github.com/michaelballantyne/faster-miniKanren.

Michael Ballantyne, William E Byrd, Greg Rosenblatt, Kanae Tsushima, and Rob Zinkov. 2021. Barliman. https://github.com/webyrd/Barliman/.

William E Byrd. 2010. Relational programming in miniKanren: techniques, applications, and implementations. (2010).

William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–26.

William E Byrd, Eric Holk, and Daniel P Friedman. 2012. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming.* 8–29.

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming.* 339–347.

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

Jason Hemann and Daniel P Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. (2013).

Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. In *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016 (EPTCS, Vol. 234),* Sibylle Schwarz and Janis Voigtländer (Eds.). 135–149. https://doi.org/10.4204/EPTCS.234.10

Joxan Jaffar and J-L Lassez. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 111–119.

Oleg Kiselyov, William Byrd, Daniel Friedman, and Chung-chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). 64–80. https://doi.org/10.1007/978-3-540-78969-7_7

John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2.* 717–740.

Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-order miniKanren representation: Great for tooling and search. In *miniKanren and Relational Programming Workshop.* 16.

Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural Guided Constraint Logic Programming for Program Synthesis. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/67d16d00201083a2b118dd5128dd6f59-Paper.pdf

A FEW MISCELLANEOUS RELATIONS

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

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

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

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

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

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

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

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

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

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