From Variadic Functions to Variadic Relations

A miniKanren Perspective

William E. Byrd and Daniel P. Friedman

Department of Computer Science, Indiana University, Bloomington, IN 47408

{webyrd,dfried}@cs.indiana.edu

Abstract

We present an implementation of miniKanren, an embedding of logic programming in R^{5}RS Scheme that comprises three logic operators. We describe these operators, and use them to define $plus^o$, a relation that adds two numbers. We then define $plus^{*o}$, which adds zero or more numbers; $plus^{*o}$ takes exactly two arguments, the first of which is a list of numbers to be added or a logical variable representing such a list. We call such a relation *pseudo-variadic*. Combining Scheme's var-args facility with pseudo-variadic helper relations leads to *variadic* relations, which take a variable number of arguments. We focus on pseudo-variadic relations, which we demonstrate are more flexible than their variadic equivalents.

We show how to define $plus^{*o}$ in terms of $plus^o$ using $foldr^o$ and $foldl^o$, higher-order relational abstractions derived from Haskell's *foldr* and *foldl* functions. These higher-order abstractions demonstrate the benefit of embedding relational operators in a functional language. We define many other pseudo-variadic relations using $foldr^o$ and $foldl^o$, consider the limitations of these abstractions, and explore their effect on the divergence behavior of the relations they define. We also consider *double-pseudo-variadic* relations, a generalization of pseudo-variadic relations that take as their first argument a list of lists or a logical variable representing a list of lists.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords miniKanren, variadic, pseudo-variadic, double-pseudo-variadic, Scheme, logic programming, relations

1. Introduction

Scheme's var-args mechanism makes it easy to define *variadic functions*, which take a variable number of arguments. miniKanren, an embedding of logic programming in Scheme, makes it easy to define *variadic relations* using that same mechanism. A variadic relation takes a variable number of arguments, but can be defined in terms of a *pseudo-variadic* helper relation that takes only two arguments, the first of which must be a list. A fresh (uninstantiated) logical variable passed as the first argument to a pseudo-variadic relation represents arbitrarily many arguments in the equivalent variadic relation—because of this flexibility, we focus on pseudo-variadic relations instead of their variadic brethren.

Certain variadic functions can be defined in terms of binary functions using the *foldr* or *foldl* abstractions [8]; certain variadic relations can be defined in terms of ternary relations using $foldr^o$ or $foldl^o$, the relational equivalents of *foldr* and *foldl*, respectively. $foldr^o$, $foldl^o$, and other miniKanren relations can be derived from their corresponding function definitions—we have omitted these derivations, most of which are trivial. The ease with which we can define higher-order relational abstractions such as $foldr^o$ demonstrates the benefits of using Scheme as the host language for miniKanren.

We also consider *double-pseudo-variadic* relations, which are a generalization of pseudo-variadic relations. A double-pseudo-variadic relation takes two arguments, the first of which is either a list of lists or a logical variable representing a list of lists. (Unless we explicitly state that a variable is a lexical variable, it is assumed to be a logical variable.) As with pseudo-variadic relations, certain double-pseudo-variadic relations can be defined using higher-order relational abstractions.

miniKanren is a descendant of the language presented in *The Reasoned Schemer* [6], which was itself inspired by Prolog. Not surprisingly, the techniques we present will be familiar to most experienced Prolog programmers.

This paper has four additional sections and an appendix. Section 2 gives a brief overview of miniKanren, and presents a simple unary arithmetic system in miniKanren using Peano numerals; these arithmetic relations are used in several of the examples. Readers unfamiliar with logic programming should carefully study this material and the miniKanren implementation in the appendix before reading section 3. (For a gentle introduction to logic programming, we recommend Clocksin [3].) Section 3 is the heart of the paper; it introduces pseudo-variadic relations, and shows how some pseudo-variadic relations can be defined using the $foldr^o$ or $foldl^o$ relational abstractions. Section 4 discusses double-pseudo-variadic relations and the $foldr^{*o}$ and $foldl^{*o}$ relational abstractions. In section 5 we conclude. The appendix presents an R^{5}RS-compliant [9] implementation of miniKanren.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

2. miniKanren Overview

This section is divided into two parts. Part one introduces the three miniKanren operators and demonstrates their behavior. Part two defines several arithmetic relations that are used in later examples, and allows the reader to become familiar with fixed-arity relations before considering the more complicated pseudo-variadic relations.

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. Quotes, quasiquotes, and unquotes are suppressed, and quoted or quasiquoted lists appear with bold parentheses—for example **()** and **(**x **.** *x***)** are entered as `'()` and `` `(x . ,x) ``, respectively. By our convention, names of relations end with a superscript *o*—for example *plus*^{o}, which is entered as `pluso`. miniKanren's relational operators do not follow this convention: ≡ (entered as `==`), **cond**^{e} (entered as `conde`), and **fresh**. Similarly, (**run**$^{\mathbf{5}}$ (*q*) *body*) and (**run*** (*q*) *body*) are entered as `(run 5 (q) body)` and `(run #f (q) body)`, respectively.

2.1 Introduction to miniKanren

miniKanren, like Schelog [15], is an embedding of logic programming in Scheme. miniKanren extends Scheme with three operators: ≡, **cond**^{e}, and **fresh**. There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

fresh, which syntactically looks like **lambda**, introduces new variables into its scope; ≡ unifies two values. Thus

```
(fresh (x y z) (≡ x z) (≡ 3 y))
```

would associate *x* with *z* and *y* with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

```
(run^1 (q) (fresh (x y z) (≡ x z) (≡ 3 y))) ⇒ (_0)
```

The value returned is a list containing the single value $\_{0}$; we say that $\_{0}$ is the *reified value* of the fresh variable *q*. *q* also remains fresh in

```
(run^1 (q) (fresh (x y) (≡ x q) (≡ 3 y))) ⇒ (_0)
```

We can get back other values, of course.

```
(run^1 (y)          (run^1 (q)         (run^1 (y)
  (fresh (x z)        (fresh (x z)       (fresh (x y)
    (≡ x z)             (≡ x z)            (≡ 4 x)
    (≡ 3 y)))           (≡ 3 z)            (≡ x y))
                        (≡ q x)))        (≡ 3 y))
```

Each of these examples returns **(**3**)**; in the rightmost example, the *y* introduced by **fresh** is different from the *y* introduced by **run**. **run** can also return the empty list, indicating that there are no values.

```
(run^1 (x) (≡ 4 3)) ⇒ ()
```

We use **cond**^{e} to get several values—syntactically, **cond**^{e} looks like **cond** but without ⇒ or **else**. For example,

```
(run^2 (q)
  (fresh (x y z)
    (cond^e
      ((≡ (x y z x) q))
      ((≡ (z y x z) q))))) ⇒
((_0 _1 _2 _0) (_0 _1 _2 _0))
```

Although the two **cond**^{e}-clauses are different, the values returned are identical. This is because distinct reified fresh variables are assigned distinct numbers, increasing from left to right—the numbering starts over again from zero within each value, which is why the reified value of *x* is $\_{0}$ in the first value but $\_{2}$ in the second value.

Here is a simpler example using **cond**^{e}.

```
(run^5 (q)
  (fresh (x y z)
    (cond^e
      ((≡ a x) (≡ 1 y) (≡ d z))
      ((≡ 2 y) (≡ b x) (≡ e z))
      ((≡ f z) (≡ c x) (≡ 3 y)))
    (≡ (x y z) q))) ⇒
((a 1 d) (b 2 e) (c 3 f))
```

The superscript 5 denotes the maximum length of the resultant list. If the superscript $*$ is used, then there is no maximum imposed. This can easily lead to infinite loops:

```
(run^* (q)
  (let loop ()
    (cond^e
      ((≡ #f q))
      ((≡ #t q))
      ((loop)))))
```

Had the $*$ been replaced by a non-negative integer *n*, then a list of *n* alternating #f's and #t's would be returned. The **cond**^{e} succeeds while associating *q* with #f, which accounts for the first value. When getting the second value, the second **cond**^{e}-clause is tried, and the association made between *q* and #f is forgotten—we say that *q* has been *refreshed.* In the third **cond**^{e}-clause, *q* is refreshed once again.

We now look at several interesting examples that rely on *any*^{o}.

```
(define any^o
  (lambda (g)
    (cond^e
      (g)
      ((any^o g)))))
```

any^o tries *g* an unbounded number of times.

Here is the first example using *any*^{o}.

```
(run^* (q)
  (cond^e
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

This example does not terminate, because the call to *any*^{o} succeeds an unbounded number of times. If $*$ is replaced by 5, then instead we get **(**#t #f #f #f #f**)**. (The user should not be concerned with the order in which values are returned.)

Now consider

```
(run^10 (q)
  (any^o
    (cond^e
      ((≡ 1 q))
      ((≡ 2 q))
      ((≡ 3 q))))) ⇒
(1 2 3 1 2 3 1 2 3 1)
```

Here the values 1, 2, and 3 are interleaved; our use of *any*^{o} ensures that this sequence will be repeated indefinitely.

Here is *always*^{o},

```
(define always^o (any^o (≡ #f #f)))
```

along with two **run** expressions that use it.

```
(run^1 (x)                  (run^5 (x)
  (≡ #t x)                    (cond^e
  always^o                      ((≡ #t x))
  (≡ #f x))                     ((≡ #f x)))
                              always^o
                              (≡ #f x))
```

The left-hand expression diverges—this is because *always*^{o} succeeds an unbounded number of times, and because (≡ #f x) fails each of those times.

The right-hand expression returns a list of five #f's. This is because both **cond**^{e}-clauses are tried, and both succeed. However, only the second **cond**^{e}-clause contributes to the values returned in this example. Nothing changes if we swap the two **cond**^{e}-clauses. If we change the last expression to (≡ #t x), we instead get a list of five #t's.

Even if some **cond**^{e}-clauses loop indefinitely, other **cond**^{e}-clauses can contribute to the values returned by a **run** expression. (We are not concerned with Scheme expressions looping indefinitely, however.) For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns (1 2 3); replacing **run**^{3} with **run**^{4} causes divergence, however, since there are only three values, and since *never*^{o} loops indefinitely.

2.2 Peano Arithmetic

The arithmetic examples in this paper use *Peano representation* of numbers (technically, *Peano numerals*). The advantage of this representation is that we can use ≡ both to construct and to match against numbers.

The Peano representation of zero is z, while the immediate successor to a Peano number n is represented as (s n). For example, one is the immediate successor of zero—the Peano representation of one is therefore (s z). Two is the immediate successor of one, so the Peano representation of two is (s (s z)).

Typographically, we indicate a Peano number using corner brackets—for example, ⌜3⌝ for (s (s (s z))). We represent (s x) as ⌜x+1⌝, (s (s x)) as ⌜x+2⌝, and so forth, where x is a variable or a reified variable (that is, a symbol).

Here is *plus*^{o}, which adds two Peano numbers.

```
(define plus^o
  (lambda (n m sum)
    (cond^e
      ((≡ ⌜0⌝ n) (≡ m sum))
      ((fresh (x y)
         (≡ ⌜x+1⌝ n)
         (≡ ⌜y+1⌝ sum)
         (plus^o x m y))))))
```

plus^o allows us to find all pairs of numbers that sum to six.

```
(run* (q)
  (fresh (n m)
    (plus^o n m ⌜6⌝)
    (≡ (n m) q))) ⇒

((⌜0⌝ ⌜6⌝)
 (⌜1⌝ ⌜5⌝)
 (⌜2⌝ ⌜4⌝)
 (⌜3⌝ ⌜3⌝)
 (⌜4⌝ ⌜2⌝)
 (⌜5⌝ ⌜1⌝)
 (⌜6⌝ ⌜0⌝))
```

Let us define *minus*^{o} using *plus*^{o}, and use it to find ten pairs of numbers whose difference is six.

```
(define minus^o
  (lambda (n m k)
    (plus^o m k n)))


(run^10 (q)
  (fresh (n m)
    (minus^o n m ⌜6⌝)
    (≡ (n m) q))) ⇒

((⌜6⌝ ⌜0⌝)
 (⌜7⌝ ⌜1⌝)
 (⌜8⌝ ⌜2⌝)
 (⌜9⌝ ⌜3⌝)
 (⌜10⌝ ⌜4⌝)
 (⌜11⌝ ⌜5⌝)
 (⌜12⌝ ⌜6⌝)
 (⌜13⌝ ⌜7⌝)
 (⌜14⌝ ⌜8⌝)
 (⌜15⌝ ⌜9⌝))
```

We have chosen to have subtraction of a larger number from a smaller number fail, rather than be zero.

```
(run* (q) (minus^o ⌜5⌝ ⌜6⌝ q)) ⇒ ()
```

We will also need *even*^{o} and *positive*^{o} in several examples below.

```
(define even^o
  (lambda (n)
    (cond^e
      ((≡ ⌜0⌝ n))
      ((fresh (m)
         (≡ ⌜m+2⌝ n)
         (even^o m))))))


(define positive^o
  (lambda (n)
    (fresh (m)
      (≡ ⌜m+1⌝ n))))
```

even^o and *positive*^{o} ensure that their arguments represent even and positive Peano numbers, respectively.

```
(run^4 (q) (even^o q)) ⇒ (⌜0⌝ ⌜2⌝ ⌜4⌝ ⌜6⌝)


(run* (q) (positive^o q)) ⇒ (⌜_0+1⌝)
```

The value ⌜$_0$+1⌝ shows that $n + 1$ is positive for every number n.

3. Pseudo-Variadic Relations

Just as a Scheme function can be variadic, so can a miniKanren relation. For example, it is possible to define a variadic version of $plus^{o}$ using Scheme's var-args feature. We must distinguish between the arguments whose values are to be added and the single argument representing the sum of those values. The simplest solution is to make the first argument represent the sum.

```
(define variadic-plus*o
  (lambda (out . in*)
    (plus*o in* out)))
```

```
(define plus*o
  (lambda (in* out)
    (conde
      ((≡ () in*) (≡ ⌜0⌝ out))
      ((fresh (a d res)
         (≡ (a . d) in*)
         (pluso a res out)
         (plus*o d res))))))
```

Here we use $variadic\text{-}plus^{*o}$ to find the sum of three, four, and two:

```
(run* (q) (variadic-plus*o q ⌜3⌝ ⌜4⌝ ⌜2⌝)) ⇒ (⌜9⌝)
```

Let us find the number that, when added to four, one, and two, produces nine.

```
(run* (q) (variadic-plus*o ⌜9⌝ ⌜4⌝ q ⌜1⌝ ⌜2⌝)) ⇒ (⌜2⌝)
```

$variadic\text{-}plus^{*o}$ is not as general as it could be, however. We cannot, for example, use $variadic\text{-}plus^{*o}$ to find all sequences of numbers that sum to five. This is because in^{*} must be an actual list, and cannot be a variable representing a list. The solution is simple—just use $plus^{*o}$ in place of $variadic\text{-}plus^{*o}$.

$plus^{*o}$, which does not use Scheme's var-args functionality, takes exactly two arguments. The first argument must be a list of numbers, or a variable representing a list of numbers. The second argument represents the sum of the numbers in the first argument, and can be either a number or a variable representing a number.

Variadic relations, such as $variadic\text{-}plus^{*o}$, are defined using *pseudo-variadic* helper relations, such as $plus^{*o}$. Henceforth, we focus exclusively on the more flexible pseudo-variadic relations, keeping in mind that each pseudo-variadic relation can be paired with a variadic relation.

To add three, four, and two using $plus^{*o}$, we write

```
(run* (q) (plus*o (⌜3⌝ ⌜4⌝ ⌜2⌝) q)) ⇒ (⌜9⌝)
```

Here is another way to add three, four, and two.

```
(run1 (q)
  (fresh (x y z)
    (plus*o x q)
    (≡ (⌜3⌝ . y) x)
    (≡ (⌜4⌝ . z) y)
    (≡ (⌜2⌝ . ()) z))) ⇒ (⌜9⌝)
```

Instead of passing a fully instantiated list of numbers as the first argument to $plus^{*o}$, we pass the fresh variable x—only afterwards do we instantiate x. Each call to ≡ further constrains the possible values of x, and consequently constrains the possible values of q. This technique of constraining the first argument to a pseudo-variadic relation through repeated calls to ≡ is similar to the partial application of a curried function—the $plus^{*o}$ relation can be considered both curried and (pseudo) variadic.

Replacing **run$^{\mathbf{1}}$** with **run$^{\mathbf{2}}$** causes the expression to diverge. This is because there is no second value to be found. Although ($plus^{*o}$ x q) succeeds an unbounded number of times, after each success one of the calls to ≡ fails, resulting in infinitely many failures without a single success, and therefore divergence. If the call to $plus^{*o}$ is made after the calls to ≡, the expression terminates even when using **run***.

```
(run* (q)
  (fresh (x y z)
    (≡ (⌜2⌝ . ()) z)
    (≡ (⌜3⌝ . y) x)
    (≡ (⌜4⌝ . z) y)
    (plus*o x q))) ⇒ (⌜9⌝)
```

We have also reordered the calls to ≡ to illustrate that the list associated with x need not be extended in left-to-right order—this reordering does not affect the behavior of the expression. The three calls to ≡ fully instantiate x; instead, we could have associated z with a pair whose *cdr* is a fresh variable, thereby leaving the variable x only partially instantiated. By the end of section 3, the reader should be able to predict the effect of this change on the behavior of the **run*** expression.

Here is a simpler example—we prove there is no sequence of numbers that begins with five whose sum is three.

```
(run* (q) (plus*o (⌜5⌝ . q) ⌜3⌝)) ⇒ ()
```

Returning to the problem that led us to use $plus^{*o}$, we generate lists of numbers whose sum is five.

```
(run10 (q) (plus*o q ⌜5⌝)) ⇒
((⌜5⌝)
 (⌜5⌝ ⌜0⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜5⌝)
 (⌜1⌝ ⌜4⌝)
 (⌜2⌝ ⌜3⌝)
 (⌜3⌝ ⌜2⌝)
 (⌜4⌝ ⌜1⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝))
```

There are infinitely many values, since a list can contain arbitrarily many zeros. We will consider the problem of generating all lists of positive numbers whose sum is five, but first we introduce a convenient abstraction for defining pseudo-variadic relations.

3.1 The *foldr^{o}* Abstraction

We can define certain pseudo-variadic relations using the $foldr^{o}$ relational abstraction. $foldr^{o}$ is derived from *foldr*, a standard abstraction for defining variadic functions in terms of binary ones [8].

```
(define foldr
  (lambda (f)
    (lambda (base-value)
      (letrec ((foldr (lambda (in*)
                        (cond
                          ((null? in*) base-value)
                          (else (f (car in*)
                                   (foldr (cdr in*))))))))
        foldr))))
```

Here we use *foldr*^{o} to define *plusr*^{*o}, which behaves like *plus*^{*o}. (For another approach to higher order relations such as *foldr*^{o}, see Naish [13] and O'Keefe [14].)

```
(define foldr^o
  (lambda (rel^o)
    (lambda (base-value)
      (letrec ((foldr^o (lambda (in* out)
                          (cond^e
                            ((≡ () in*) (≡ base-value out))
                            ((fresh (a d res)
                               (≡ (a . d) in*)
                               (rel^o a res out)
                               (foldr^o d res)))))))
        foldr^o))))
```

```
(define plusr*^o ((foldr^o plus^o) ⌜0⌝))
```

The first argument to *foldr* must be a binary function, whereas the first argument to *foldr*^{o} must be a ternary relation. The values of *rel*^{o} and *base-value* do not change in the recursive call to *foldr*^{o}—this allows us to pass in *rel*^{o} and *base-value* before passing in *in** and *out*. We make a distinction between the *rel*^{o} and *base-value* arguments: although *base-value* might be a variable, the value of *rel*^{o} must be a miniKanren relation, and therefore a Scheme function.

We use *positive-plusr*^{*o} to ensure that we add only positive numbers.

```
(define positive-plusr*^o
  ((foldr^o (lambda (a res out)
              (fresh ()
                (positive^o a)
                (plus^o a res out))))
   ⌜0⌝))
```

Finally, we have the sixteen lists of positive numbers whose sum is five.

```
(run* (q) (positive-plusr*^o q ⌜5⌝)) ⇒
```

```
((⌜5⌝)
 (⌜1⌝ ⌜4⌝)
 (⌜2⌝ ⌜3⌝)
 (⌜1⌝ ⌜1⌝ ⌜3⌝)
 (⌜3⌝ ⌜2⌝)
 (⌜1⌝ ⌜2⌝ ⌜2⌝)
 (⌜4⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜3⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝))
```

Let us consider another pseudo-variadic relation; *positive-even-plusr*^{*o} succeeds if its first argument represents a list of positive numbers whose sum is even.

```
(define positive-even-plusr*^o
  (lambda (in* out)
    (fresh ()
      (even^o out)
      (positive-plusr*^o in* out))))
```

Here are the first ten values returned by *positive-even-plusr*^{*o}.

```
(run^10 (q)
  (fresh (x y)
    (≡ (x y) q)
    (positive-even-plusr*^o x y))) ⇒
```

```
((() ⌜0⌝)
 ((⌜2⌝) ⌜2⌝)
 ((⌜1⌝ ⌜1⌝) ⌜2⌝)
 ((⌜4⌝) ⌜4⌝)
 ((⌜1⌝ ⌜3⌝) ⌜4⌝)
 ((⌜2⌝ ⌜2⌝) ⌜4⌝)
 ((⌜3⌝ ⌜1⌝) ⌜4⌝)
 ((⌜1⌝ ⌜1⌝ ⌜2⌝) ⌜4⌝)
 ((⌜1⌝ ⌜2⌝ ⌜1⌝) ⌜4⌝)
 ((⌜2⌝ ⌜1⌝ ⌜1⌝) ⌜4⌝))
```

Replacing **run**$^{\mathbf{10}}$ with **run*** causes divergence, since there are infinitely many values.

Let us consider another pseudo-variadic relation defined using *foldr*^{o}. Here is *append*^{o}, which appends two lists, and its pseudo-variadic variant *appendr*^{*o}.

```
(define append^o
  (lambda (l s out)
    (cond^e
      ((≡ () l) (≡ s out))
      ((fresh (a d res)
         (≡ (a . d) l)
         (≡ (a . res) out)
         (append^o d s res))))))
```

```
(define appendr*^o ((foldr^o append^o) ()))
```

Here are four examples of *appendr*^{*o}. In the first example, we use *appendr*^{*o} simply to append two lists.

```
(run* (q) (appendr*^o ((a b c) (d e)) q)) ⇒ ((a b c d e))
```

In the second example we infer for which value of q the list (a b c d . q) is equal to the concatenation of the lists (a b c), (d e), and (f g).

```
(run* (q) (appendr*^o ((a b c) (d e) (f g)) (a b c d . q)))
⇒ ((e f g))
```

The third example is more interesting—the contents of the second of three lists being appended can be inferred from the second argument to *appendr*^{*o}.

```
(run* (q) (appendr*^o ((a b c) q (g h)) (a b c d e f g h)))
⇒ ((d e f))
```

The final example shows a few of the lists of lists whose contents, when appended, are (2 3 d e).

```
(run^10 (q) (appendr*^o ((a b) (c) (1) . q) (a b c 1 2 3 d e)))
⇒
```

```
(((2 3 d e))
 ((2 3 d e) ())
 ((2 3 d e) () ())
 (() (2 3 d e))
 ((2) (3 d e))
 ((2 3) (d e))
 ((2 3 d) (e))
 ((2 3 d e) () () ())
 ((2 3 d e) () () () ())
 (() (2 3 d e) ()))
```

Replacing **run**$^{\mathbf{10}}$ with **run*** causes the last expression to diverge, since there are infinitely many values that contain the empty list—by using *pair-appendr*^{*o} instead of *appendr*^{*o}, we filter out these values.

```
(define pair^o
  (lambda (p)
    (fresh (a d)
      (≡ (a . d) p))))
```

```
(define pair-appendr^*o
  ((foldr^o (lambda (a res out)
              (fresh ()
                (pair^o a)
                (append^o a res out))))
   ()))
```

Now let us re-evaluate the previous example.

```
(run* (q)
  (pair-appendr^*o ((a b) (c) (1) . q) (a b c 1 2 3 d e))) ⇒
(((2 3 d e))
 ((2) (3 d e))
 ((2 3) (d e))
 ((2 3 d) (e))
 ((2) (3) (d e))
 ((2) (3 d) (e))
 ((2 3) (d) (e))
 ((2) (3) (d) (e)))
```

These eight values are the only ones that do not include the empty list.

3.2 The *foldl*^{o} Abstraction

Previously we defined pseudo-variadic relations with the *foldr*^{o} relational abstraction. We can also define certain pseudo-variadic relations using the *foldl*^{o} relational abstraction, which is derived from the standard *foldl* function; like *foldr*, *foldl* is used to define variadic functions in terms of binary ones.

```
(define foldl
  (lambda (f)
    (letrec
      ((foldl
         (lambda (acc)
           (lambda (in*)
             (cond
               ((null? in*) acc)
               (else ((foldl (f acc (car in*)))
                      (cdr in*))))))))
       foldl)))
```

Here we use *foldl*^{o} to define *plusl*^{*o} and *appendl*^{*o}, which are similar to *plusr*^{*o} and *appendr*^{*o}, respectively.

```
(define foldl^o
  (lambda (rel^o)
    (letrec
      ((foldl^o (lambda (acc)
                  (lambda (in* out)
                    (cond^e
                      ((≡ () in*) (≡ acc out))
                      ((fresh (a d res)
                         (≡ (a . d) in*)
                         (rel^o acc a res)
                         ((foldl^o res) d out))))))))
       foldl^o)))
```

```
(define plusl^*o ((foldl^o plus^o) ⌜0⌝))
```

```
(define appendl^*o ((foldl^o append^o) ()))
```

As we did with *foldr*^{o}, we separate the *rel*^{o} and *acc* arguments from the *in** and *out* arguments.

We have defined pseudo-variadic versions of *plus*^{o} using both *foldr*^{o} and *foldl*^{o}; these definitions differ in their divergence behavior. Consider this example, which uses *plusr*^{*o}.

```
(run1 (q) (plusr^*o (⌜4⌝ q ⌜3⌝) ⌜5⌝)) ⇒ ()
```

Replacing *plusr*^{*o} with *plusl*^{*o} causes the expression to diverge. *foldl*^{o} passes the fresh variable *res* as the third argument to the ternary relation, while *foldr*^{o} instead passes the *out* variable, which in this example is fully instantiated. This accounts for the difference in divergence behavior—the relation called by *foldr*^{o} has additional information that can lead to termination.

It is possible to use *foldl*^{o} to define *positive-plusl*^{*o}, *positive-even-plusl*^{*o}, and *pair-appendl*^{*o}. Some pseudo-variadic relations can be defined using *foldl*^{o}, but not *foldr*^{o}. For example, here is *subsetl*^{o}, which generates subsets of a given set (where sets are represented as lists).

```
(define ess^o
  (lambda (in* x out)
    (cond^e
      ((≡ () in*) (≡ ((x)) out))
      ((fresh (a d â d̂)
         (≡ (a . d) in*)
         (cond^e
           ((≡ (â . d) out) (≡ (x . a) â))
           ((≡ (a . d̂) out) (ess^o d x d̂))))))))
```

```
(define subsetl^o ((foldl^o ess^o) ()))
```

Here we use *subsetl*^{o} to find all the subsets of the set containing a, b, and c.

```
(run* (q) (subsetl^o (a b c) q)) ⇒
(((c b a)) ((b a) (c)) ((c a) (b)) ((a) (c b)) ((a) (b) (c)))
```

It is possible to infer the original set from which a given subset has been generated.

```
(run1 (q) (subsetl^o q ((a d) (c) (b)))) ⇒ ((d c b a))
```

Replacing **run**$^{\mathbf{1}}$ with **run**$^{\mathbf{2}}$ yields two values: (d c b a) and (d c a b). Unfortunately, these values are duplicates—they represent the same set. It is possible to eliminate these duplicate sets (for example, by using Prolog III-style disequality constraints [4]), but the techniques involved are not directly related to pseudo-variadic relations.

Here is *partition-suml*^{o}, whose definition is very similar to that of *subsetl*^{o}. Like *subsetl*^{o}, *partition-suml*^{o} cannot be defined using *foldr*^{o}.

```
(define pes^o
  (lambda (in* x out)
    (cond^e
      ((≡ () in*) (≡ (x) out))
      ((fresh (a d â d̂)
         (≡ (a . d) in*)
         (cond^e
           ((≡ (â . d) out) (plus^o x a â))
           ((≡ (a . d̂) out) (pes^o d x d̂))))))))
```

```
(define partition-suml^o ((foldl^o pes^o) ()))
```

partition-suml^o partitions a set of numbers, and returns another set containing the sums of the numbers in the various partitions. (This problem was posed in a July 5, 2006 post on the `comp.lang.scheme` newsgroup [5]). An example helps clarify the problem.

```
(run* (q) (partition-suml^o (⌜1⌝ ⌜2⌝ ⌜5⌝ ⌜9⌝) q)) ⇒
((⌜8⌝ ⌜9⌝)
 (⌜3⌝ ⌜5⌝ ⌜9⌝)
 (⌜17⌝)
 (⌜12⌝ ⌜5⌝)
 (⌜3⌝ ⌜14⌝)
 (⌜10⌝ ⌜2⌝ ⌜5⌝)
 (⌜1⌝ ⌜2⌝ ⌜5⌝ ⌜9⌝)
 (⌜6⌝ ⌜2⌝ ⌜9⌝)
 (⌜1⌝ ⌜11⌝ ⌜5⌝)
 (⌜1⌝ ⌜7⌝ ⌜9⌝)
 (⌜1⌝ ⌜2⌝ ⌜14⌝)
 (⌜15⌝ ⌜2⌝)
 (⌜6⌝ ⌜11⌝)
 (⌜10⌝ ⌜7⌝)
 (⌜1⌝ ⌜16⌝))
```

Consider the value (⌜15⌝ ⌜2⌝). We obtain the ⌜15⌝ by adding ⌜1⌝, ⌜5⌝, and ⌜9⌝, while the ⌜2⌝ is left unchanged.

We can infer the original set of numbers, given a specific final value.

```
(run^10 (q) (partition-suml^o q (⌜3⌝))) ⇒
((⌜3⌝)
 (⌜3⌝ ⌜0⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜0⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜2⌝ ⌜1⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝)
 (⌜1⌝ ⌜0⌝ ⌜2⌝))
```

There are infinitely many values containing zero—one way of eliminating these values is to use *positive-partition-suml*^{o}.

```
(define positive-pes^o
  (lambda (in* x out)
    (fresh ()
      (positive^o x)
      (cond^e
        ((≡ () in*) (≡ (x) out))
        ((fresh (a d â d̂)
           (≡ (a . d) in*)
           (cond^e
             ((≡ (â . d) out) (plus^o x a â))
             ((≡ (a . d̂) out) (positive-pes^o d x d̂)))))))))
```

```
(define positive-partition-suml^o ((foldl^o positive-pes^o) ()))
```

```
(run^4 (q) (positive-partition-suml^o q (⌜3⌝))) ⇒
((⌜3⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

We have eliminated values containing zeros, but we still are left with duplicate values—worse, the last value is not even a set. As before, we could use disequality constraints or other techniques to remove these undesired values. Regardless of whether we use these techniques, the previous **run** expression will diverge if we replace **run**$^{\mathbf{4}}$ with **run**$^{\mathbf{5}}$; this divergence is due to our use of *foldl*^{o}.

3.3 When *foldr*^{o} and *foldl*^{o} do not work

Consider *minusr*^{*o}, which is a pseudo-variadic *minus*^{o} defined using *plusr*^{*o}. Unlike the pseudo-variadic addition relations, *minusr*^{*o} fails when *in** is the empty list. Also, when *in** contains only a single number, that number must be zero. This is because the negation of any positive number is negative, and because Peano numbers only represent non-negative integers. These special cases prevent us from defining *minusr*^{*o} using *foldr*^{o} or *foldl*^{o}.

```
(define minusr*^o
  (lambda (in* out)
    (cond^e
      ((≡ (⌜0⌝) in*) (≡ ⌜0⌝ out))
      ((fresh (a d res)
         (≡ (a . d) in*)
         (pair^o d)
         (minus^o a res out)
         (plusr*^o d res))))))
```

Here we use *minusr*^{*o} to generate lists of numbers that, when subtracted from seven, yield three.

```
(run^14 (q) (minusr*^o (⌜7⌝ . q) ⌜3⌝)) ⇒
((⌜4⌝)
 (⌜4⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜4⌝)
 (⌜1⌝ ⌜3⌝)
 (⌜2⌝ ⌜2⌝)
 (⌜3⌝ ⌜1⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜4⌝ ⌜0⌝)
 (⌜1⌝ ⌜3⌝ ⌜0⌝)
 (⌜2⌝ ⌜2⌝ ⌜0⌝)
 (⌜3⌝ ⌜1⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝))
```

The values containing zero are not very interesting—let us filter out those values by using *positive-minusr*^{*o}.

```
(define positive-minusr*^o
  (lambda (in* out)
    (fresh (a d res)
      (≡ (a . d) in*)
      (positive^o a)
      (minus^o a res out)
      (positive-plusr*^o d res))))
```

Now we can use **run*** instead of **run**$^{\mathbf{14}}$, since there are only finitely many values.

```
(run* (q) (positive-minusr*^o (⌜7⌝ . q) ⌜3⌝)) ⇒
((⌜4⌝)
 (⌜1⌝ ⌜3⌝)
 (⌜2⌝ ⌜2⌝)
 (⌜3⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝))
```

As might be expected, we could use plusl^{*o} to define the relations minusl^{*o} and $\mathit{positive\text{-}minusl}^{*o}$.

Here is $\mathit{positive}^{*o}$, another pseudo-variadic relation that cannot be defined using foldr^{o} or foldl^{o}; this is because $\mathit{positive}^{*o}$ takes only one argument.

```
(define positive*o
  (lambda (in*)
    (conde
      ((≡ () in*))
      ((fresh (a d)
         (≡ (a . d) in*)
         (positiveo a)
         (positive*o d))))))
```

```
(run5 (q) (positive*o q)) ⇒
(()
 (⌜-0+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝ ⌜-2+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝ ⌜-2+1⌝ ⌜-3+1⌝))
```

4. Double-Pseudo-Variadic Relations

A pseudo-variadic relation takes a list, or a variable representing a list, as its first argument; a *double-pseudo-variadic* relation takes a list of lists, or a variable representing a list of lists, as its first argument. Let us define plusr^{**o}, the double-pseudo-variadic version of plusr^{*o}. We define plusr^{**o} using the foldr^{*o} relational abstraction.

```
(define foldr*o
  (lambda (relo)
    (lambda (base-value)
      (letrec ((foldr*o (lambda (in** out)
                          (conde
                            ((≡ () in**) (≡ base-value out))
                            ((fresh (dd)
                               (≡ (() . dd) in**)
                               (foldr*o dd out)))
                            ((fresh (a d dd res)
                               (≡ ((a . d) . dd) in**)
                               (relo a res out)
                               (foldr*o (d . dd) res)))))))
        foldr*o))))
```

```
(define plusr**o ((foldr*o pluso) ⌜0⌝))
```

As with plusr^{*o}, we can use plusr^{**o} to add three, four, and two.

```
(run* (q) (plusr**o ((⌜3⌝ ⌜4⌝ ⌜2⌝)) q)) ⇒ (⌜9⌝)
```

plusr^{**o} allows us to add three, four, and two in more than one way, by partitioning the list of numbers to be added into various sublists, which can include the empty list.

```
(run* (q) (plusr**o (() (⌜3⌝ ⌜4⌝) () (⌜2⌝)) q)) ⇒ (⌜9⌝)
```

In the previous section we used plusr^{*o} to generate lists of numbers whose sum is five; here we use plusr^{**o} to generate lists of numbers whose sum is three.

```
(run10 (q) (plusr**o (q) ⌜3⌝)) ⇒
((⌜3⌝)
 (⌜3⌝ ⌜0⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝ ⌜0⌝)
 (⌜1⌝ ⌜2⌝ ⌜0⌝))
```

There are infinitely many such lists, since each list can contain an arbitrary number of zeros.

As we did with plusr^{*o}, let us use plusr^{**o} to prove that there is no sequence of numbers that begins with five whose sum is three.

```
(run1 (q) (plusr**o ((⌜5⌝) q) ⌜3⌝)) ⇒ ()
```

This expression terminates because plusr^{**o} calls (plus^{o} ⌜5⌝ *res* ⌜3⌝), which immediately fails.

Swapping the positions of the fresh variable q and the list containing five yields the expression

```
(run1 (q) (plusr**o (q (⌜5⌝)) ⌜3⌝))
```

which diverges. q represents a list of numbers—since each list can contain arbitrarily many zeros, there are infinitely many such lists whose sum is less than or equal to three. For each such list, plusr^{**o} sums the numbers in the list, and then adds five to that sum; this fails, of course, since the new sum is greater than three. Since there are infinitely many lists, and therefore infinitely many failures without a single success, the expression diverges.

If we were to restrict q to a list of positive numbers, the previous expression would terminate.

```
(define positive-plusr**o
  ((foldr*o (lambda (a res out)
              (fresh ()
                (positiveo a)
                (pluso a res out))))
   ⌜0⌝))
```

```
(run1 (q) (positive-plusr**o (q (⌜5⌝)) ⌜3⌝)) ⇒ ()
```

We can now generate all the lists of positive numbers whose sum is three.

```
(run* (q) (positive-plusr**o (q) ⌜3⌝)) ⇒
((⌜3⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

The following expression returns all lists of positive numbers containing five that sum to eight.

```
(run* (q)
  (fresh (x y)
    (positive-plusr**o (x (⌜5⌝) y) ⌜8⌝)
    (appendr*o (x (⌜5⌝) y) q))) ⇒
((⌜5⌝ ⌜3⌝)
 (⌜5⌝ ⌜2⌝ ⌜1⌝)
 (⌜5⌝ ⌜1⌝ ⌜2⌝)
 (⌜5⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜5⌝ ⌜2⌝)
 (⌜1⌝ ⌜5⌝ ⌜1⌝ ⌜1⌝)
 (⌜2⌝ ⌜5⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜5⌝ ⌜1⌝)
 (⌜3⌝ ⌜5⌝)
 (⌜1⌝ ⌜2⌝ ⌜5⌝)
 (⌜2⌝ ⌜1⌝ ⌜5⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜5⌝))
```

Here is a more complicated example—we want to find all lists of numbers that sum to twenty-five and satisfy certain additional constraints. The list must begin with a list w of positive numbers, followed by the number three, followed by any single positive number x, the number four, a list y of positive numbers, the number five, and a list z of positive numbers.

```
(run* (q)
  (fresh (w x y z in**)
    (≡ (w (⌜3⌝ x ⌜4⌝) y (⌜5⌝) z) in**)
    (positive-plusr**o in** ⌜25⌝)
    (appendr*o in** q)))
```

Here is a list of the first four values.

```
((⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜12⌝)
 (⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜1⌝ ⌜11⌝)
 (⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜2⌝ ⌜10⌝)
 (⌜3⌝ ⌜2⌝ ⌜4⌝ ⌜5⌝ ⌜11⌝))
```

For the curious, the 7,806th value is

```
(⌜1⌝ ⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜11⌝)
```

and the 4,844th value is

```
(⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜1⌝ ⌜5⌝ ⌜9⌝ ⌜2⌝)
```

foldr^{*o} is not the only double-pseudo-variadic relational abstraction; here is *foldl*^{*o}, which we use to define *plusl*^{**o} and *positive-plusl*^{**o}.

```
(define foldl*o
  (lambda (relo)
    (letrec
      ((foldl*o (lambda (acc)
                  (lambda (in** out)
                    (conde
                      ((≡ () in**) (≡ acc out))
                      ((fresh (dd)
                         (≡ (() . dd) in**)
                         ((foldl*o acc) dd out)))
                      ((fresh (a d dd res)
                         (≡ ((a . d) . dd) in**)
                         (relo acc a res)
                         ((foldl*o res) (d . dd) out))))))))
       foldl*o)))

(define plusl**o ((foldl*o pluso) ⌜0⌝))

(define positive-plusl**o
  ((foldl*o (lambda (acc a res)
              (fresh ()
                (positiveo a)
                (pluso acc a res))))
   ⌜0⌝))
```

Let us revisit an example demonstrating *positive-plusr*^{**o}; we replace *positive-plusr*^{**o} with *positive-plusl*^{**o}, and **run*** with **run**$^{\mathbf{4}}$.

```
(run^4 (q) (positive-plusl**o (q) ⌜3⌝)) ⇒

((⌜3⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

We get back the same values as before, although in a different order. If we replace **run**$^{\mathbf{4}}$ with **run**$^{\mathbf{5}}$, however, the expression diverges. This should not be surprising, since we have already seen a similar difference in divergence behavior between *plusr*^{*o} and *plusl*^{*o}.

Finally, let us consider a double-pseudo-variadic relation that cannot be defined using *foldr*^{*o} or *foldl*^{*o}. Here is *minusr*^{**o}, the generalization of *minusr*^{*o}.

```
(define minusr**o
  (lambda (in** out)
    (fresh (in*)
      (appendr*o in** in*)
      (minusr*o in* out))))
```

The definition is made simple by the call to *appendr*^{*o}—why do we not use this technique when defining other double-pseudo-variadic relations? Because the call to *appendr*^{*o} can succeed an unbounded number of times when *in*** is fresh or partially instantiated, which can easily lead to divergence:

```
(run^1 (q) (minusr**o ((⌜3⌝) q) ⌜5⌝))
```

diverges because the call to *appendr*^{*o} keeps succeeding, and because after each success the call to *minusr*^{*o} fails.

Of course not every use of *minusr*^{**o} results in divergence. Let us find ten lists of numbers that contain seven and whose difference is three.

```
(run^10 (q)
  (fresh (x y)
    (≡ (x (⌜7⌝) y) q)
    (minusr**o q ⌜3⌝))) ⇒
((() (⌜7⌝) (⌜4⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜4⌝))
 (() (⌜7⌝) (⌜1⌝ ⌜3⌝))
 (() (⌜7⌝) (⌜2⌝ ⌜2⌝))
 (() (⌜7⌝) (⌜4⌝ ⌜0⌝))
 (() (⌜7⌝) (⌜3⌝ ⌜1⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜0⌝ ⌜4⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜1⌝ ⌜3⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜2⌝ ⌜2⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜4⌝ ⌜0⌝)))
```

These values give the impression that x is always associated with the empty list, which is not true. For example, when we replace **run**$^{\mathbf{10}}$ with **run**$^{\mathbf{70}}$, then the twenty-third and seventieth values are `((⌜10⌝) (⌜7⌝) ())` and `((⌜11⌝) (⌜7⌝) (⌜1⌝))`, respectively.

Let us exclude values in which sublists contain zeros, and display the concatenation of the sublists to make the results more readable.

```
(run^10 (q)
  (fresh (x y in**)
    (≡ (x (⌜7⌝) y) in**)
    (minusr**o in** ⌜3⌝)
    (appendr*o in** q)
    (positive*o q))) ⇒
((⌜7⌝ ⌜4⌝)
 (⌜7⌝ ⌜1⌝ ⌜3⌝)
 (⌜7⌝ ⌜2⌝ ⌜2⌝)
 (⌜7⌝ ⌜3⌝ ⌜1⌝)
 (⌜7⌝ ⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜7⌝ ⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜7⌝ ⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜10⌝ ⌜7⌝)
 (⌜7⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜11⌝ ⌜7⌝ ⌜1⌝))
```

Of course there are still infinitely many values, even after filtering out lists containing zeros.

Finally, let us replace the second argument to *minusr*^{**o} with a fresh variable.

```
(run¹⁰ (q)
  (fresh (x y in* in** out)
    (≡ (in* out) q)
    (≡ (x (⌜7⌝) y) in**)
    (minusr**ᵒ in** out)
    (appendr*ᵒ in** in*)
    (positive*ᵒ in*))) ⇒
```

```
(((⌜7⌝ ⌜1⌝) ⌜6⌝)
 ((⌜7⌝ ⌜2⌝) ⌜5⌝)
 ((⌜7⌝ ⌜3⌝) ⌜4⌝)
 ((⌜7⌝ ⌜4⌝) ⌜3⌝)
 ((⌜7⌝ ⌜5⌝) ⌜2⌝)
 ((⌜7⌝ ⌜6⌝) ⌜1⌝)
 ((⌜7⌝ ⌜7⌝) ⌜0⌝)
 ((⌜7⌝ ⌜1⌝ ⌜1⌝) ⌜5⌝)
 ((⌜7⌝ ⌜1⌝ ⌜2⌝) ⌜4⌝)
 ((⌜7⌝ ⌜2⌝ ⌜1⌝) ⌜4⌝))
```

When we replace **run^{10}** with **run^{30}**, the thirtieth value is

```
((⌜_.0+7⌝ ⌜7⌝) _.0)
```

This value shows that $(n + 7) - 7 = n$ for every n.

5. Conclusions

We have seen how to define both variadic and pseudo-variadic relations in miniKanren, an embedding of logic programming in Scheme. A variadic relation is defined using Scheme's var-args facility, and can take a variable number of arguments. A pseudo-variadic relation takes two arguments, the first of which is a list or a variable representing a list; the ability to pass a fresh variable as the first argument makes a pseudo-variadic relation more flexible than its variadic equivalent.

Just as certain variadic Scheme functions can be defined using the *foldr* or *foldl* abstractions, certain pseudo-variadic relations can be defined using the *foldr*^{o} or *foldl*^{o} relational abstractions. For example, pseudo-variadic versions of *plus*^{o} that add arbitrarily many numbers can be defined using either relational abstraction. Another example is *subsetl*^{o}, which is defined using *foldl*^{o} but cannot be redefined using *foldr*^{o}. Even those pseudo-variadic relations that can be defined using both relational abstractions may exhibit different divergence behavior when using one abstraction instead of the other. And some pseudo-variadic relations, such as *minusr*^{*o}, cannot be defined using either *foldr*^{o} or *foldl*^{o}.

We have also seen how to define double-pseudo-variadic relations, which are a generalization of pseudo-variadic relations. A double-pseudo-variadic relation takes two arguments, the first of which is either a list of lists or a variable representing a list of lists. As with pseudo-variadic relations, certain double-pseudo-variadic relations can be defined using relational abstractions—for example, *plusr*^{**o} is defined using *foldr*^{*o}.

The benefits of using Scheme as the host language for miniKanren are demonstrated by the ease with which we can define higher-order relational abstractions such as *foldr*^{*o}. We hope the examples we have presented inspire the reader to experiment with miniKanren, and to experience the fun of combining relational programming with Scheme.

Acknowledgments

The implementation described in the appendix was developed with Oleg Kiselyov and Chung-chieh Shan. We are also grateful to Michael Adams and Kyle Ross for their comments on our earlier proposals for pseudo-variadic relations. We thank Ronald Garcia, Oleg Kiselyov, Chung-chieh Shan, Michael Adams, and the anonymous reviewers for their detailed and insightful comments on drafts of this paper. We have found Dorai Sitaram's excellent SLaTeX package invaluable for typesetting our code. Finally, we thank Mitchell Wand for his encouragement and clever last-minute tweaking of the implementation.

References

[1] A declarative applicative logic programming system. http://kanren.sourceforge.net/.

[2] Baader, F., and Snyder, W. Unification theory. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds., vol. I. Elsevier Science, 2001, ch. 8, pp. 445–532.

[3] Clocksin, W. F. *Clause and Effect: Prolog Programming and the Working Programmer.* Springer, 1997.

[4] Colmerauer, A. An introduction to Prolog III. *Commun. ACM 33*, 7 (1990), 69–90.

[5] Edelstein, H. add-up problem. Message posted on newsgroup comp.lang.scheme on July 5, 2006 07:12 EST.

[6] Friedman, D. P., Byrd, W. E., and Kiselyov, O. *The Reasoned Schemer.* The MIT Press, Cambridge, MA, 2005.

[7] Henderson, F., Conway, T., Somogyi, Z., and Jeffery, D. The Mercury language reference manual. Tech. Rep. 96/10, University of Melbourne, 1996.

[8] Jones, S. L. P. Haskell 98: Standard prelude. *J. Funct. Program. 13*, 1 (2003), 103–124.

[9] Kelsey, R., Clinger, W., and Rees, J. Revised^{5} report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76.

[10] Kiselyov, O., Shan, C., Friedman, D. P., and Sabry, A. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005* (2005), O. Danvy and B. C. Pierce, Eds., ACM, pp. 192–203.

[11] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991), 55–92.

[12] Naish, L. Pruning in logic programming. Tech. Rep. 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.

[13] Naish, L. Higher-order logic programming in Prolog. Tech. Rep. 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, Feb. 1996.

[14] O'Keefe, R. *The Craft of Prolog.* The MIT Press, Cambridge, MA, 1990.

[15] Sitaram, D. Programming in Schelog. http://www.ccs.neu.edu/home/dorai/schelog/schelog.html.

[16] Spivey, J. M., and Seres, S. Combinators for logic programming. In *The Fun of Programming*, J. Gibbons and O. de Moor, Eds. Palgrave, 2003, ch. 9, pp. 177–200.

[17] Wadler, P. The essence of functional programming. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, Jan., 1992), ACM Press, pp. 1–14.

A. miniKanren Implementation

miniKanren evolved from Kanren [1]; its implementation comprises three kinds of operators: functions such as *unify* and *reify*, which take substitutions explicitly; goal constructors ≡, **cond**^{e}, and **fresh**, which take substitutions implicitly; and the interface operator **run**. We represent substitutions as association lists associating variables with values.

unify is based on the triangular model of substitutions (See Baader and Snyder [2], for example.). Vectors should not occur in arguments passed to *unify*, since we represent variables as vectors.

```
(define unify
  (lambda (u v s)
    (let ((u (walk u s))
          (v (walk v s)))
      (cond
        ((eq? u v) s)
        ((var? u)
         (cond
           ((var? v) (ext-s u v s))
           (else (ext-s√ u v s))))
        ((var? v) (ext-s√ v u s))
        ((and (pair? u) (pair? v))
         (let ((s (unify (car u) (car v) s)))
           (and s (unify (cdr u) (cdr v) s))))
        ((equal? u v) s)
        (else #f)))))

(define walk
  (lambda (v s)
    (cond
      ((var? v)
       (let ((a (assq v s)))
         (cond
           (a (walk (cdr a) s))
           (else v))))
      (else v))))

(define ext-s√
  (lambda (x v s)
    (cond
      ((occurs√ x v s) #f)
      (else (ext-s x v s)))))

(define occurs√
  (lambda (x v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (eq? v x))
        ((pair? v)
         (or (occurs√ x (car v) s) (occurs√ x (cdr v) s)))
        (else #f)))))

(define ext-s
  (lambda (x v s)
    (cons (x . v) s)))

(define empty-s ())

(define var vector)

(define var? vector?)
```

reify takes a substitution and an arbitrary value, perhaps containing variables. *reify* first uses *walk** to apply the substitution to a value and then methodically replaces any variables with reified names.

```
(define reify
  (letrec
    ((reify-s
       (lambda (v s)
         (let ((v (walk v s)))
           (cond
             ((var? v) (ext-s v (reify-name (length s)) s))
             ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
             (else s))))))
    (lambda (v s)
      (let ((v (walk* v s)))
        (walk* v (reify-s v empty-s))))))

(define walk*
  (lambda (w s)
    (let ((v (walk w s)))
      (cond
        ((var? v) v)
        ((pair? v) (cons (walk* (car v) s) (walk* (cdr v) s)))
        (else v)))))

(define reify-name
  (lambda (n)
    (string→symbol
      (string-append "_" "." (number→string n)))))
```

A goal g is a function that maps a substitution s to an ordered sequence of zero or more values—these values are almost always substitutions. (For clarity, we notate **lambda** as λ_G when creating such a function g.) Because the sequence of values may be infinite, we represent it not as a list but as a special kind of stream, a^∞.

Such streams contain either zero, one, or more values [10, 16]. We use (**mzero**) to represent the empty stream of values. If a is a value, then (**unit** a) represents the stream containing just a. To represent a non-empty stream we use (**choice** a f), where a is the first value in the stream, and where f is a function of zero arguments. (For clarity, we notate **lambda** as λ_F when creating such a function f.) Invoking the function f produces the remainder of the stream. (**unit** a) can be represented as (**choice** a (λ_F () (**mzero**))), but the **unit** constructor avoids the cost of building and taking apart pairs and invoking functions, since many goals return only singleton streams. To represent an incomplete stream, we create an f using (**inc** e), where e is an *expression* that evaluates to an a^∞.

```
(define-syntax mzero
  (syntax-rules ()
    ((_) #f)))

(define-syntax unit
  (syntax-rules ()
    ((_ a) a)))

(define-syntax choice
  (syntax-rules ()
    ((_ a f) (cons a f))))

(define-syntax inc
  (syntax-rules ()
    ((_ e) (λF () e))))
```

To ensure that streams produced by these four a^∞ constructors can be distinguished, we assume that a singleton a^∞ is never **#f**, a function, or a pair whose *cdr* is a function. To discriminate among these four cases, we define **case**$^\infty$.

```
(define-syntax case∞
  (syntax-rules ()
    ((_ e on-zero ((â) on-one) ((a f) on-choice) ((f̂) on-inc))
     (let ((a∞ e))
       (cond
         ((not a∞) on-zero)
         ((procedure? a∞) (let ((f̂ a∞)) on-inc))
         ((and (pair? a∞) (procedure? (cdr a∞)))
          (let ((a (car a∞)) (f (cdr a∞))) on-choice))
         (else (let ((â a∞)) on-one)))))))
```

The simplest goal constructor is $\equiv$, which returns either a singleton stream or an empty stream, depending on whether the arguments unify with the implicit substitution. As with the other goal constructors, $\equiv$ always expands to a goal, even if an argument diverges. We avoid the use of **unit** and **mzero** in the definition of $\equiv$, since *unify* returns either a substitution (a singleton stream) or **#f** (our representation of the empty stream).

```
(define-syntax ≡
  (syntax-rules ()
    ((_ u v)
     (λG (s)
       (unify u v s)))))
```

cond^e is a goal constructor that combines successive **cond**^{e}-clauses using **mplus***. To avoid unwanted divergence, we treat the **cond**^{e}-clauses as a single **inc** stream. Also, we use the same implicit substitution for each **cond**^{e}-clause. **mplus*** relies on *mplus*, which takes an a^∞ and an f and combines them (a kind of *append*). Using **inc**, however, allows an argument to *become* a stream, thus leading to a relative fairness because all of the stream values will be interleaved.

```
(define-syntax condᵉ
  (syntax-rules ()
    ((_ (g₀ g ...) (g₁ ĝ ...) ...)
     (λG (s)
       (inc
         (mplus*
           (bind* (g₀ s) g ...)
           (bind* (g₁ s) ĝ ...) ...))))))

(define-syntax mplus*
  (syntax-rules ()
    ((_ e) e)
    ((_ e₀ e ...) (mplus e₀ (λF () (mplus* e ...))))))

(define mplus
  (lambda (a∞ f)
    (case∞ a∞
      (f)
      ((a) (choice a f))
      ((a f̂) (choice a (λF () (mplus (f̂) f))))
      ((f̂) (inc (mplus (f) f̂))))))
```

If the body of **cond**^{e} were just the **mplus*** expression, then the **inc** clauses of *mplus*, *bind*, and *take* would never be reached, and there would be no interleaving of values.

fresh is a goal constructor that first lexically binds its variables (created by *var*) and then, using **bind***, combines successive goals. **bind*** is short-circuiting: since the empty stream (**mzero**) is represented by **#f**, any failed goal causes **bind*** to immediately return **#f**. **bind*** relies on *bind* [11, 17], which applies the goal g to each element in a^∞. These a^∞'s are then merged together with *mplus* yielding an a^∞. (*bind* is similar to Lisp's *mapcan*, with the arguments reversed.)

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g₀ g ...)
     (λG (s)
       (let ((x (var x)) ...)
         (bind* (g₀ s) g ...))))))

(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
    ((_ e g₀ g ...)
     (let ((a∞ e))
       (and a∞ (bind* (bind a∞ g₀) g ...))))))

(define bind
  (lambda (a∞ g)
    (case∞ a∞
      (mzero)
      ((a) (g a))
      ((a f) (mplus (g a) (λF () (bind (f) g))))
      ((f) (inc (bind (f) g))))))
```

To minimize heap allocation we create a single λ_G closure for each goal constructor, and we define **bind*** and **mplus*** to manage sequences, not lists, of goal-like expressions.

run, and therefore *take*, converts an f to a list. We wrap the result of (*reify x s*) in a list so that the **case**$^\infty$ in *take* can distinguish a singleton a^∞ from the other three a^∞ types. We could simplify **run** by using *var* to create the fresh variable x, but we prefer that **fresh** be the only operator that calls *var*.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g₀ g ...)
     (take n
       (λF ()
         (let ((ĝ (fresh (x)
                    (λG (s)
                      (bind* (g₀ s) g ...
                        (λG (s)
                          (list (reify x s))))))))
           (ĝ empty-s)))))))

(define take
  (lambda (n f)
    (if (and n (zero? n))
      ()
      (case∞ (f)
        ()
        ((a) a)
        ((a f) (cons (car a) (take (and n (− n 1)) f)))
        ((f) (take n f))))))
```

If the first argument to *take* is **#f**, we get the behavior of **run***. It is trivial to write a read-eval-print loop that uses the **run*** interface by redefining *take*.

This ends the implementation of the subset of miniKanren used in this paper. Below we define the three additional goal constructors that complete the entire embedding: **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program, and **project**, which can be used to access the values of variables.

cond^a and **cond**^{u} correspond to the *committed-choice* of Mercury, and are used in place of Prolog's *cut* [7, 12]. Unlike **cond**^{e}, only one **cond**^{a}-clause or **cond**^{u}-clause can return an a^∞: the first clause whose first goal succeeds. With **cond**^{a}, the entire stream returned by the first goal is passed to **bind*** (see **pick**^{a}). With **cond**^{u}, a singleton stream is passed to **bind***—this stream contains the first value of the stream returned by the first goal (see **pick**^{u}). The examples from chapter 10 of *The Reasoned Schemer* [6] demonstrate how **cond**^{a} and **cond**^{u} can be useful and the pitfalls that await the unsuspecting reader.

```
(define-syntax condᵃ
  (syntax-rules ()
    ((_ (g₀ g ...) (g₁ ĝ ...) ...)
     (λG (s)
       (if* (pickᵃ (g₀ s) g ...) (pickᵃ (g₁ s) ĝ ...) ...)))))

(define-syntax condᵘ
  (syntax-rules ()
    ((_ (g₀ g ...) (g₁ ĝ ...) ...)
     (λG (s)
       (if* (pickᵘ (g₀ s) g ...) (pickᵘ (g₁ s) ĝ ...) ...)))))

(define-syntax if*
  (syntax-rules ()
    ((_) (mzero))
    ((_ (pick e g ...) b ...)
     (let loop ((a∞ e))
       (case∞ a∞
         (if* b ...)
         ((a) (bind* a∞ g ...))
         ((a f) (bind* (pick a a∞) g ...))
         ((f) (inc (loop (f)))))))))

(define-syntax pickᵃ
  (syntax-rules ()
    ((_ a a∞) a∞)))

(define-syntax pickᵘ
  (syntax-rules ()
     ((_ a a∞) (unit a))))
```

project applies the implicit substitution to zero or more lexical variables, rebinds those variables to the values returned, and then evaluates the goal expressions in its body. The body of a **project** typically includes at least one **begin** expression—any expression is a goal expression if its value is a miniKanren goal. **project** has many uses, such as displaying the values associated with variables when tracing a program.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g₀ g ...)
     (λG (s)
       (let ((x (walk* x s)) ...)
         (bind* (g₀ s) g ...))))))
```