miniKanren, Live and Untagged

Quine Generation via Relational Interpreters (Programming Pearl)

William E. Byrd Eric Holk Daniel P. Friedman

School of Informatics and Computing, Indiana University, Bloomington, IN 47405

{webyrd,eholk,dfried}@cs.indiana.edu

Abstract

We present relational interpreters for several subsets of Scheme, written in the pure logic programming language miniKanren. We demonstrate these interpreters running "backwards"—that is, generating programs that evaluate to a specified value—and show how the interpreters can trivially generate *quines* (programs that evaluate to themselves). We demonstrate how to transform environment-passing interpreters written in Scheme into relational interpreters written in miniKanren. We show how constraint extensions to core miniKanren can be used to allow shadowing of the interpreter's primitive forms (using the $absent^o$ tree constraint), and to avoid having to tag expressions in the languages being interpreted (using disequality constraints and symbol/number type-constraints), simplifying the interpreters and eliminating the need for parsers/unparsers.

We provide four appendices to make the code in the paper completely self-contained. Three of these appendices contain new code: the complete implementation of core miniKanren extended with the new constraints; an extended relational interpreter capable of running factorial and doing list processing; and a simple pattern matcher that uses Dijkstra guards. The other appendix presents our preferred version of code that has been presented elsewhere: the miniKanren relational arithmetic system used in the extended interpreter.

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

General Terms Languages

Keywords quines, Scheme, miniKanren, relational programming, logic programming, interpreters, tagging

1. Introduction

A *quine* is a program that evaluates to itself (Hofstadter 1979; Thompson II); the simplest Scheme quines are self-evaluating literals, such as numbers and booleans. A classic non-trivial quine (Thompson II) is:

```
(define quine_c
  '((lambda (x)
      (list x (list (quote quote) x)))
    (quote
      (lambda (x)
        (list x (list (quote quote) x))))))
```

We can easily verify that $quine_c$ evaluates to itself:

$(equal?\ (eval\ quine_c)\ quine_c) \Rightarrow$ #t

For decades programmers have amused themselves by writing quines in countless programming languages. Some quines, such as those featured in the International Obfuscated C Code Contest (Broukhis et al.), are intentionally baroque. Here we demonstrate a disciplined approach to the problem: we show how to translate a standard environment-passing interpreter written as a Scheme function into a relation in the pure logic programming language miniKanren (Byrd 2009; Friedman et al. 2005), then show how this relational interpreter can be used, without modification, to trivially generate quines.[1]

We also show how to generate *twines* (twin quines), which are distinct programs p and q that evaluate to each other, and *thrines*, which are distinct programs p, q, and r such that p evaluates to q, q evaluates to r, and r evaluates to p.

While generating quines is fun and interesting, our approach also illustrates advanced techniques of relational programming, such as translating functional programs into relational programs, and using constraints to avoid having to tag the expressions being interpreted. This last point is especially important, as tagging implies the need to write parsers and unparsers, and because tagging the application line of the interpreter greatly complicates the handling of **quote** and *list*. Our use of constraints also has the important benefit of properly handling shadowing of the interpreter's built-in primitives, such as *list*, **quote**, and **lambda**.

Our approach requires adding several constraint operators to core miniKanren. We have previously presented disequality constraints in cKanren (Alvis et al. 2011), a general constraint logic programming (Apt 2003) framework inspired by miniKanren; the $symbol^o$, $number^o$, and

[1] For readers already familiar with miniKanren, the punchline of the paper can be summarized by the one-liner:

$(equal?\ (caar\ (\mathbf{run^1}\ (q)\ (eval\text{-}exp^o\ q\ \text{'}()\ q)))\ quine_c) \Rightarrow$ #t

(More accurately, the generated quine is α-equivalent to $quine_c$.)

absent^o constraints we introduce are also straightforward to implement in cKanren. However, we have found that core miniKanren augmented with these four constraints is sufficient for implementing a wide variety of interesting programs, including interpreters and inferencers. This extended miniKanren is conceptually simpler than cKanren, and its implementation is easier to understand and modify. Programmers needing to use domain-specific constraints, such as arithmetic over finite domains (CLP(FD)), will find that the techniques described here, up to and including quine generation, work equally well in cKanren.

Our paper makes the following contributions:

- We extend the miniKanren core language with four constraint operators (section 2.2): the disequality constraint $\not\equiv$; type constraints *symbol*^{o} and *number*^{o}, which are similar in spirit to Scheme's *symbol?* and *number?* predicates; and the tree constraint *absent*^{o}, which ensures a symbol *tag* does not occur inside a term t. These constraints are extremely useful when writing logic programs, especially interpreters and type inferencers.
- We describe and demonstrate our methodology for translating interpreters from Scheme to miniKanren (section 3). Our methodology is equally useful for translating type inferencers.
- We show how $\not\equiv$, *symbol*^{o}, and *number*^{o} can be used when writing an interpreter (or type inferencer) to avoid having to tag expressions in the language being interpreted, and how *absent*^{o} can be used to allow shadowing of primitive forms (section 3).
- We present relational interpreters for three subsets of Scheme: the call-by-value λ-calculus (section 3); λ-calculus extended with *list* and **quote** (section 4); and an extended language supporting pairs, conditionals, and arithmetic operators, and capable of running factorial (appendix A). The relational arithmetic system (appendix B) used in the third interpreter was first presented in Friedman et al. (2005); we include it for completeness.
- We demonstrate these interpreters running "backwards" (generating input expressions from the expected output), and show how the interpreters supporting *list* and **quote** can be used to trivially generate quines (section 5 and appendix A).
- We provide a complete, concise, and easily modifiable implementation of core miniKanren extended with $\not\equiv$, *symbol*^{o}, *number*^{o}, and *absent*^{o} constraints (appendix D).
- We provide a generalized version of the **pmatch** pattern matcher first presented in Byrd and Friedman (2007); the updated **pmatch**, now called **dmatch** (appendix C), is based on Dijkstra guards (Dijkstra 1975). This generalized pattern matcher properly handles **quote** expressions, which is important when writing evaluators, inferencers, and reducers.

We begin by introducing the extended miniKanren language we will use to write the relational interpreters.

2. The Extended miniKanren Language

In this section we briefly review the core miniKanren language (section 2.1), then introduce the $\not\equiv$, *symbol*^{o}, *number*^{o}, and *absent*^{o} constraints used in the relational interpreters (section 2.2). Readers already familiar with miniKanren can safely skip to section 2.2 to learn about the new constraints, while those wishing to learn more about miniKanren should see Byrd (2009), Byrd and Friedman (2006) (from which this subsection has been adapted), and Friedman et al. (2005).

2.1 miniKanren Refresher

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. By our convention, names of relations end with a superscript *o*—for example *any*^{o}, which is entered as `anyo`. Some relational operators do not follow this convention: $\equiv$ (entered as `==`), **cond**^{e} (entered as `conde`), and **fresh**. Similarly, $(\mathbf{run^5}\ (q)\ body)$ and $(\mathbf{run}^*\ (q)\ body)$ are entered as `(run 5 (q) body)` and `(run* (q) body)`.[2]

Core miniKanren extends Scheme with three operators: $\equiv$, **fresh**, and **cond**^{e}. (Four additional constraint operators are introduced in section 2.2.) There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

$\equiv$ unifies two terms. **fresh**, which syntactically looks like **lambda**, introduces lexically-scoped Scheme variables that are bound to new logic variables; **fresh** also performs conjunction of the relations within its body. Thus

$$(\mathbf{fresh}\ (x\ y\ z)\ (\equiv x\ z)\ (\equiv 3\ y))$$

would introduce logic variables x, y, and z, then 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.

$$(\mathbf{run^1}\ (q)\ (\mathbf{fresh}\ (x\ y\ z)\ (\equiv x\ z)\ (\equiv 3\ y))) \Rightarrow (\_{0})$$

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

$$(\mathbf{run^1}\ (q)\ (\mathbf{fresh}\ (x\ y)\ (\equiv x\ q)\ (\equiv 3\ y))) \Rightarrow (\_{0})$$

We can get back more interesting values by unifying the query variable with another term.

$$\begin{array}{lll}
(\mathbf{run^1}\ (y) & (\mathbf{run^1}\ (q) & (\mathbf{run^1}\ (y) \\
\quad (\mathbf{fresh}\ (x\ z) & \quad (\mathbf{fresh}\ (x\ z) & \quad (\mathbf{fresh}\ (x\ y) \\
\qquad (\equiv x\ z) & \qquad (\equiv x\ z) & \qquad (\equiv 4\ x) \\
\qquad (\equiv 3\ y))) & \qquad (\equiv 3\ z) & \qquad (\equiv x\ y)) \\
 & \qquad (\equiv q\ x))) & \quad (\equiv 3\ y))
\end{array}$$

Each of these examples returns (3); in the rightmost example, the y introduced by **fresh** is different from the y introduced by **run**.

A **run** expression can return the empty list, indicating that the body of the expression is logically inconsistent.

$$(\mathbf{run^1}\ (x)\ (\equiv 4\ 3)) \Rightarrow ()$$

$$(\mathbf{run^1}\ (x)\ (\equiv 5\ x)\ (\equiv 6\ x)) \Rightarrow ()$$

We say that a logically inconsistent relation *fails*, while a logically consistent relation, such as $(\equiv 3\ 3)$, *succeeds.*

cond^e, which resembles **cond** syntactically, is used to produce multiple answers. Logically, **cond**^{e} can be thought of as disjunctive normal form: each clause represents a disjunct, and is independent of the other clauses, with the relations within a clause acting as the conjuncts. For example, this expression produces two answers.

[2] It is conventional in Scheme for the names of predicates to end with the '?' character. We have therefore chosen to end the names of miniKanren goals with a superscript *o*, which is meant to resemble the top of a '?'. The superscript *e* in **cond**^{e} stands for 'every,' since every **cond**^{e} clause may contribute answers.

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

Although the two **cond**^{e} lines are different, the values returned are identical. This is because distinct reified unbound variables are assigned distinct subscripts, increasing from left to right—the numbering starts over again from zero within each answer, which is why the reified value of x is $\_{0}$ in the first answer but $\_{1}$ in the second. The superscript 2 in **run** 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))))) ⇒ ⊥
```

If we replace $*$ by a natural number n, then an n-element list of alternating #f's and #t's is returned. The first answer is produced by the first **cond**^{e} clause, which associates q with #f. To produce the second answer, the second **cond**^{e} clause is tried. Since **cond**^{e} clauses are independent, the association between q and #f made in the first clause is forgotten—we say that q has been *refreshed*. In the third **cond**^{e} clause, q is refreshed again.

We now look at several interesting examples that rely on any^o, which tries g an unbounded number of times.

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

Consider the first example,

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

which does not terminate because the call to any^o succeeds an unbounded number of times. If $*$ were replaced by 5, then we would get (#t #f #f #f #f). (The user should not be concerned with the order of the answers produced.)

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 is repeated indefinitely.

Even if a relation within a **cond**^{e} clause loops indefinitely (or *diverges*), other **cond**^{e} clauses can contribute to the answers returned by a **run** expression. 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**$^{\mathbf{3}}$ with **run**$^{\mathbf{4}}$ would cause divergence, since $never^o$ would loop indefinitely looking for the non-existent fourth answer.

2.2 Additional Constraint Operators

We extend core miniKanren with four constraint operators: the disequality constraint $\not\equiv$ (previously described in the context of the cKanren constraint logic programming framework (Alvis et al. 2011)); type constraints $symbol^o$ and $number^o$, which are the miniKanren equivalent of Scheme's *symbol?* and *number?* type predicates; and $absent^o$, which ensures a symbol *tag* does not occur in a term t.

We begin with the $symbol^o$ type constraint.

```
(run* (q) (symbol^o q)) ⇒ ((_0 (sym _0)))
```

The single answer ($\_{0}$ (sym $\_{0}$)) indicates that q remains unbound, and also that q represents a symbol. Any attempt to associate q with a non-symbol value should therefore lead to failure.

```
(run* (q)                (run* (q)
  (symbol^o q)             (symbol^o q)
  (≡ 4 q)) ⇒ ()            (number^o q)) ⇒ ()
```

If we were to replace all occurrences of $symbol^o$ by $number^o$ in the three examples above, the new answers produced would be (($\_{0}$ (num $\_{0}$))), (4), and (($\_{0}$ (num $\_{0}$))), respectively.

Next we consider the disequality constraint $\not\equiv$.

```
(run* (p) (≢ p 1)) ⇒ ((_0 (≢ ((_0 1)))))
```

The answer states that p remains unbound, but cannot be associated with 1. Of course, violating the constraint leads to failure:

```
(run* (p) (≢ 1 p) (≡ 1 p)) ⇒ ()
```

A slightly more complicated example is a disequality constraint between two lists.

```
(run* (q)
  (fresh (p r)
    (≢ '(1 2) `(,p ,r))
    (≡ `(,p ,r) q))) ⇒ (((_0 _1) (≢ ((_0 1) (_1 2)))))
```

The answer states that p and r are unbound, and that p cannot be associated with 1 while r is associated with 2 (and the other way around). We would get the same answer if we were to replace (≢ '(1 2) `(,p ,r)) by either (≢ '((1) (2)) `((,p) (,r))) or (≢ `((1) (,r)) `((,p) (2))).

Now consider the **run** expression

```
(run* (q)
  (fresh (p r)
    (≢ '(1 2) `(,p ,r))
    (≡ 1 p)
    (≡ `(,p ,r) q))) ⇒ (((1 _0) (≢ ((_0 2)))))
```

If we also associate r with 2, the **run** expression fails.

```
(run* (q)
  (fresh (p r)
    (≢ '(1 2) `(,p ,r))
    (≡ 1 p)
    (≡ 2 r)
    (≡ `(,p ,r) q))) ⇒ ()
```

Now consider what happens when (≡ 2 r) is replaced by (*symbol^o* r) in the previous example. Then the **run** expression succeeds with the answer (((1 $_{-0}$) (sym $_{-0}$))) which states that r can only be associated with a symbol. The reified constraint (≢ (($_{-0}$ 2))) (stating that r cannot be associated with 2) is not included in the answer, since it is subsumed by the constraint that r must be a symbol.

Finally we consider *absent^o*, which ensures a symbol *tag* does not appear in a term t. Assume we have a term q containing predators such as jackals and leopards, and we desire to keep gentle pandas out of this dangerous term. We can use *absent^o* to ensure that this will occur.

```
(run* (q)
  (fresh (x y)
    (≡ `(jackal (,y leopard ,x)) q)
    (absentᵒ 'panda q)))
⇒
(((jackal (_0 leopard _1))
  (absent panda _0)
  (absent panda _1)))
```

The answer states that the two unbound variables, x and y, cannot be associated with a term that contains the term panda. If we violate this constraint by associating x with panda (or with a list containing panda), the **run** expression no longer returns any answers, keeping the pandas safe.

```
(run* (q)
  (fresh (x y)
    (≡ `(jackal (,y leopard ,x)) q)
    (absentᵒ 'panda q)
    (≡ 'panda x))) ⇒ ()
```

If x is known to be a symbol, the *absent^o* constraint on x can be simplified to a disequality constraint between x and panda.

```
(run* (q)
  (fresh (x y)
    (≡ `(jackal (,y leopard ,x)) q)
    (absentᵒ 'panda q)
    (symbolᵒ x)))
⇒
(((jackal (_0 leopard _1))
 (≢ ((_1 panda)))
 (absent panda _0)
 (sym _1)))
```

The answer still contains the full *absent^o* constraint on y; violating this constraint does indeed cause failure.

```
(run* (q)
  (fresh (x y z)
    (≡ `(jackal (,y leopard ,x)) q)
    (absentᵒ 'panda q)
    (symbolᵒ x)
    (≡ `(c ,z d) y)
    (≡ 'panda z))) ⇒ ()
```

3. Translating an Interpreter from Scheme to miniKanren

In this section we start with a standard environment-passing interpreter for the call-by-value λ-calculus, then show how the interpreter can be translated into miniKanren in order to run "backwards."

We begin by defining variable lookup in an environment represented as an association list.

```
(define lookup
  (lambda (x env)
    (dmatch env
      (() (error 'lookup "unbound variable"))
      (((,y . ,v) . ,rest) (guard (eq? y x))
       v)
      (((,y . ,v) . ,rest) (guard (not (eq? y x)))
       (lookup x rest)))))
```

lookup uses **dmatch** (appendix C), a simple pattern matcher with guards in the style of Dijkstra's Guarded Commands (Dijkstra 1975). **dmatch** ensures that the patterns and optional guards of different clauses do not overlap.[3] This *non-overlapping property* ensures that the ordering of the clauses does not matter, and is required for writing correct relational programs (Byrd 2009). By ensuring the non-overlapping property holds in the Scheme version of the interpreter, we simplify the translation to miniKanren.

Now that we have defined *lookup*, we can write our simple interpreter using **dmatch**.

```
(define eval-exp
  (lambda (exp env)
    (dmatch exp
      ((,rator ,rand)
       (let ((proc (eval-exp rator env))
             (arg (eval-exp rand env)))
         (dmatch proc
           ((closure ,x ,body ,env₂)
            (eval-exp body `((,x . ,arg) . ,env₂))))))
      ((lambda (,x) ,body)
       (guard (symbol? x) (not-in-env? 'lambda env))
       `(closure ,x ,body ,env))
      (,x (guard (symbol? x)) (lookup x env)))))

(define not-in-env?
  (lambda (x env)
    (dmatch env
      (() #t)
      (((,y . ,v) . ,rest) (guard (eq? y x)) #f)
      (((,y . ,v) . ,rest) (guard (not (eq? y x)))
       (not-in-env? x rest)))))
```

The guard in *eval-exp*'s lambda clause includes the test

```
(not-in-env? 'lambda env)
```

When we extend the interpreter in section 4, and again in appendix A, we will use a similar *not-in-env?* test in the guard of every new language form and primitive operator. This use of *not-in-env?* serves two critical and related purposes. First, the test ensures that procedure application does not overlap with any other clause in the interpreter, such as the (quote ,v) and (list . ,a^*) clauses we add in section 4.

[3] For this reason **dmatch** does not support **else**, since the always-true implicit test of the **else** clause overlaps with the patterns and guards of all other clauses.

Second, the test ensures that *eval-exp* will correctly evaluate expressions in which lambda (or quote or list) is shadowed. For example, the interpreter correctly handles shadowing of lambda in this definition of Ω.

```
(define Ω
  '((lambda (lambda) (lambda lambda))
    (lambda (lambda) (lambda lambda))))
```

As expected, Ω diverges.

```
(eval-exp Ω '()) ⇒ ⊥
```

The empty list passed as the second argument to *eval-exp* represents the empty environment.

Here are two more examples showing *eval-exp* in action:

```
(eval-exp
  '(((lambda (x)
       (lambda (y) x))
     (lambda (z) z))
    (lambda (a) a))
  '())
⇒
(closure z z ())
```

```
(eval-exp
  '((lambda (x)
      (lambda (y) x))
    (lambda (z) z))
  '())
⇒
(closure y x
  ((x . (closure z z ()))))
```

We now have a working λ-calculus interpreter in Scheme that properly handles shadowing, and in which the **dmatch** clauses can be reordered arbitrarily. Our next task is to translate the interpreter directly into miniKanren. We start again with environment lookup; a faithful translation of *lookup* into *lookup*^{o} might be:

```
(define lookup^o
  (lambda (x env t)
    (cond^e
      ((≡ '() env) fail)
      ((fresh (y v rest)
         (≡ `((,y . ,v) . ,rest) env) (≡ y x)
         (≡ v t)))
      ((fresh (y v rest)
         (≡ `((,y . ,v) . ,rest) env) (≢ y x)
         (lookup^o x rest t))))))
```

lookup^o takes a third argument, t, which corresponds to the value returned by the Scheme function *lookup*. That is, t represents the term associated with variable x in environment *env*.

```
(run* (q) (lookup^o 'y '((x . foo) (y . bar)) q)) ⇒ (bar)
```

If x is not bound in *env*, a call to *lookup*^{o} will reach the base case and fail[4], rather than signaling an error as in *lookup*.

```
(run* (q) (lookup^o 'w '((x . foo) (y . bar)) q)) ⇒ ()
```

Each **cond**^{e} clause in *lookup*^{o} corresponds to a **dmatch** clause in *lookup*. Instead of pattern matching against *env*, *lookup*^{o} uses $\equiv$ to unify terms with *env*. The goal $(\not\equiv y\ x)$ is equivalent to the guard (*not* (*eq?* *y* *x*)) in *lookup*. As with **dmatch**, the order of clauses does not affect the meaning of a **cond**^{e} expression (but may affect its performance). Unlike with **dmatch**, the order of expressions *within* a **cond**^{e} clause is unimportant—there are no patterns or guards within a **cond**^{e} clause, only goals that succeed or fail.[5]

We can simplify *lookup*^{o} by removing the first **cond**^{e} clause (which always fails), and by moving the unification of *env* above the **cond**^{e}.

```
(define lookup^o
  (lambda (x env t)
    (fresh (y v rest)
      (≡ `((,y . ,v) . ,rest) env)
      (cond^e
        ((≡ y x) (≡ v t))
        ((≢ y x) (lookup^o x rest t))))))
```

With *lookup*^{o} defined, we can write *eval-exp*^{o}, which in turn relies on *not-in-env*^{o}. Since there is no notion of a guard in miniKanren, we must translate each **dmatch** guard into one or more goal expressions; the Scheme predicate (*symbol?* *x*) becomes the type constraint (*symbol*^{o} *x*), while (*not-in-env?* 'lambda *env*) becomes (*not-in-env*^{o} 'lambda *env*).

```
(define eval-exp^o
  (lambda (exp env val)
    (cond^e
      ((fresh (rator rand x body env_2 a)
         (≡ `(,rator ,rand) exp)
         (eval-exp^o rator env `(closure ,x ,body ,env_2))
         (eval-exp^o rand env a)
         (eval-exp^o body `((,x . ,a) . ,env_2) val)))
      ((fresh (x body)
         (≡ `(lambda (,x) ,body) exp)
         (symbol^o x)
         (≡ `(closure ,x ,body ,env) val)
         (not-in-env^o 'lambda env)))
      ((symbol^o exp) (lookup^o exp env val)))))
```

not-in-env^o differs from *lookup*^{o} and *eval-exp*^{o} in that it does not take an additional argument with respect to the Scheme function from which it was translated. This is because *not-in-env?* is a predicate—the success or failure of *not-in-env*^{o} is equivalent to *not-in-env?* returning #t or #f, respectively, so no "output" argument is needed.

```
(define not-in-env^o
  (lambda (x env)
    (cond^e
      ((≡ '() env))
      ((fresh (y v rest)
         (≡ `((,y . ,v) . ,rest) env)
         (≢ y x)
         (not-in-env^o x rest))))))
```

We can use *eval-exp*^{o} to generate expression/value pairs.

```
(run^5 (q)
  (fresh (e v)
    (eval-exp^o e '() v)
    (≡ `(,e → ,v) q)))
```

This **run**$^{\mathbf{5}}$ expression generates five λ-expressions, and the closures to which they evaluate.

[4] *fail* can be defined as (**define** *fail* (≡ #f #t)).

[5] Recall from section 2.1 that all goals within a **cond**^{e} clause must succeed for the entire clause to succeed.

```
((((lambda (_.0) _.1)
   → (closure _.0 _.1 ()))
  (sym _.0))
 ((((lambda (_.0) _.0) (lambda (_.1) _.2))
   → (closure _.1 _.2 ()))
  (sym _.0 _.1))
 ((((lambda (_.0) (lambda (_.1) _.2)) (lambda (_.3) _.4))
   → (closure _.1 _.2 ((_.0 . (closure _.3 _.4 ())))))
  (≢ ((_.0 lambda)))
  (sym _.0 _.1 _.3))
 ((((lambda (_.0) (_.0 _.0)) (lambda (_.1) _.1))
   → (closure _.1 _.1 ()))
  (sym _.0 _.1))
 ((((lambda (_.0) (_.0 _.0))
    (lambda (_.1) (lambda (_.2) _.3)))
   → (closure _.2 _.3 ((_.1 . (closure _.1 (lambda (_.2) _.3) ())))))
  (≢ ((_.1 lambda)))
  (sym _.0 _.1 _.2)))
```

The $\not\equiv$ tags in the answers indicate disequality constraints between variables and the values they cannot assume. The constraints in the last answer state that $_{.1}$ cannot be the symbol lambda and that $_{.0}$, $_{.1}$, and $_{.2}$ are symbols.

To demonstrate *eval-exp^o* running backwards, here are five Scheme expressions that evaluate to the closure from the last *eval-exp* example:

```
(run⁵ (q)
  (eval-expᵒ q '() '(closure y x ((x . (closure z z ()))))))
⇒
(((lambda (x) (lambda (y) x)) (lambda (z) z))
 ((lambda (x) (x (lambda (y) x))) (lambda (z) z))
 (((lambda (x) (lambda (y) x))
   ((lambda (_.0) _.0) (lambda (z) z)))
  (sym _.0))
 ((((lambda (_.0) _.0) (lambda (x) (lambda (y) x)))
   (lambda (z) z))
  (sym _.0))
 (((lambda (_.0) _.0)
   ((lambda (x) (lambda (y) x)) (lambda (z) z)))
  (sym _.0)))
```

The constraint (sym $_{.0}$) states that the fresh variable reified as $_{.0}$ can only be associated with a symbol.

4. Extending the Interpreter

We have a relational interpreter, but we cannot yet generate quines, or even evaluate *quine$_c$* from section 1 when running forward. We must add **quote** and *list* to the interpreter, first to the Scheme version, then to miniKanren translation.

Adding **quote** to the Scheme interpreter is relatively simple. Since **quote** means "do not evaluate the argument," we simply have to return the argument unmodified. Thus, we can support **quote** by adding this clause to our interpreter:

```
((quote ,v) v)
```

In order to handle shadowing correctly, we must allow the user to override the **quote** form. As with the lambda clause, we do so by calling *not-in-env?* within a guard:

```
((quote ,v) (guard (not-in-env? 'quote env)) v)
```

Unfortunately, **quote** introduces a new problem: it is possible for quoted data to conflict with our representation of closures. For example, in the context of our interpreter, these two expressions are equivalent:

```
((lambda (x) x) (lambda (y) y)))
```

and

```
((quote (closure x x ())) (lambda (y) y))
```

This is because (lambda (x) x) and (quote (closure x x ())) both evaluate to (closure x x ()). We circumvent this issue by declaring that the closure tag is a unique symbol that is not part of the expression language (a gensym by convention).

We can now translate the quote clause to miniKanren. In the Scheme version, we can assume that the user will not write expressions containing the closure tag; alternatively, we could use an actual gensym for the tag. However, we are interested in running our relational interpreter backwards, and miniKanren has no compunction against generating expressions that include a symbol the user cannot or should not type. We can solve this problem by adding an *absent^o* constraint with the symbol closure as the first argument, ensuring the closure tag does not appear within v.

```
((fresh (v)
   (≡ `(quote ,v) exp)
   (not-in-envᵒ 'quote env)
   (absentᵒ 'closure v)))
```

Like the lambda clause presented in section 3, the quote clause uses *not-in-env^o* to handle shadowing.

We now turn our attention to *list*. For the Scheme interpreter, *list* is simply a matter of mapping recursive calls to *eval-exp* over the arguments:

```
((list . ,a*) (guard (not-in-env? 'list env))
 (map (lambda (e) (eval-exp e env)) a*))
```

Similarly, we can add *list* to the miniKanren interpreter:

```
((fresh (a*)
   (≡ `(list . ,a*) exp)
   (not-in-envᵒ 'list env)
   (absentᵒ 'closure a*)
   (proper-listᵒ a* env val)))
```

Once again, we use the *absent^o* constraint to prevent miniKanren from generating list *expressions* that contain closures. (Of course, a list expression containing λ expressions will *evaluate* to a list containing closures, but the expression being evaluated must not contain closures.)

As with the other tagged clauses, proper handling of shadowing is ensured through use of *not-in-env^o*. These uses of *not-in-env^o* also serve another purpose: without this constraint, the expression (list x) would be recognized both as a procedure application (of whatever procedure might be bound to the variable *list*), and as a use of the built-in primitive *list*.

The list clause relies on *proper-list^o* to ensure a^* is a proper list:

```
(define proper-listᵒ
  (lambda (exp env val)
    (condᵉ
      ((≡ '() exp)
       (≡ '() val))
      ((fresh (a d v-a v-d)
         (≡ `(,a . ,d) exp)
         (≡ `(,v-a . ,v-d) val)
         (eval-expᵒ a env v-a)
         (proper-listᵒ d env v-d))))))
```

Our final definition of *eval-exp* is

```
(define eval-exp
  (lambda (exp env)
    (dmatch exp
      ((quote ,v) (guard (not-in-env? 'quote env)) v)
      ((list . ,a*) (guard (not-in-env? 'list env))
       (map (lambda (e) (eval-exp e env)) a*))
      (,x (guard (symbol? x)) (lookup x env))
      ((,rator ,rand)
       (guard (rator? rator env))
       (let ((proc (eval-exp rator env))
             (arg (eval-exp rand env)))
         (dmatch proc
           ((closure ,x ,body ,env)
            (eval-exp body `((,x . ,arg) . ,env))))))
      ((lambda (,x) ,body)
       (guard (symbol? x) (not-in-env? 'lambda env))
       `(closure ,x ,body ,env)))))

(define rator?
  (let ((op-names '(lambda quote list)))
    (lambda (x env)
      (not (and (symbol? x)
                (memq x op-names)
                (not-in-env? x env))))))
```

The *rator?* predicate is used to ensure the procedure application clause does not overlap with the lambda, list, or quote clauses. An expression x is an operator, *unless* it one of the symbols lambda, list, or quote, *and* it is not bound in the environment. (If x is bound in *env*, it means the operator is being shadowed, and that *exp* is a procedure application).

The definition of *eval-exp* in section 3 does not need to use *rator?* because the three λ-calculus expressions have different shapes: applications are represented by lists of length two, abstractions are represented by lists of length three, and variables are represented by symbols. Therefore there is no overlap between clauses, which is required when using **dmatch**.

Here is the extended relational interpreter.

```
(define eval-exp^o
  (lambda (exp env val)
    (cond^e
      ((fresh (v)
         (≡ `(quote ,v) exp)
         (not-in-env^o 'quote env)
         (absent^o 'closure v)
         (≡ v val)))
      ((fresh (a*)
         (≡ `(list . ,a*) exp)
         (not-in-env^o 'list env)
         (absent^o 'closure a*)
         (proper-list^o a* env val)))
      ((symbol^o exp) (lookup^o exp env val))
      ((fresh (rator rand x body env^ a)
         (≡ `(,rator ,rand) exp)
         (eval-exp^o rator env `(closure ,x ,body ,env^))
         (eval-exp^o rand env a)
         (eval-exp^o body `((,x . ,a) . ,env^) val)))
      ((fresh (x body)
         (≡ `(lambda (,x) ,body) exp)
         (symbol^o x)
         (not-in-env^o 'lambda env)
         (≡ `(closure ,x ,body ,env) val))))))
```

It is not necessary to test if a rator is valid, since the application clause will fail if *rator* is a symbol not bound in the environment. If *exp* is (quote (lambda (x) x)), for example, and *env* contains a binding between quote and a closure, then *eval-exp*^{o} will treat the expression as a procedure application, rather than a use of the **quote** form; otherwise, if quote is not bound to a closure in *env*, the application clause will fail.

5. Generating Quines

After much work, we are finally ready to put *eval-exp*^{o} through its paces, and generate a quine. The call to *eval-exp*^{o} is trivial—we want to find a Scheme expression q that, when evaluated in the empty environment, returns itself.

```
(run¹ (q) (eval-exp^o q '() q)) ⇒
((((lambda (_0) (list _0 (list 'quote _0)))
   '(lambda (_0) (list _0 (list 'quote _0))))
  (≢ ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Sure enough, this is our old friend, *quine*$_c$.

We can push things further by attempting to generate *twines*, also known as "twin quines" or "double quines." That is, we want to find programs p and q such that (*eval* p) $\Rightarrow$ q and (*eval* q) $\Rightarrow$ p. According to this definition every quine is trivially a twine, so we add the restriction that p and q are not equal.

```
(run¹ (x)
  (fresh (p q)
    (≢ p q)
    (eval-exp^o p '() q) (eval-exp^o q '() p)
    (≡ `(,p ,q) x)))
⇒
((('((lambda (_0)
       (list 'quote (list _0 (list 'quote _0))))
     '(lambda (_0) (list 'quote (list _0 (list 'quote _0)))))
    ((lambda (_0) (list 'quote (list _0 (list 'quote _0))))
     '(lambda (_0) (list 'quote (list _0 (list 'quote _0))))))
  (≢ ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Finally, we generate *thrines*: distinct programs p, q, and r such that (*eval* p) $\Rightarrow$ q, (*eval* q) $\Rightarrow$ r, and (*eval* r) $\Rightarrow$ p.

```
(run¹ (x)
  (fresh (p q r)
    (≢ p q) (≢ q r) (≢ r p)
    (eval-exp^o p '() q) (eval-exp^o q '() r) (eval-exp^o r '() p)
    (≡ `(,p ,q ,r) x)))
⇒
(((''((lambda (_0)
        (list 'quote (list 'quote (list _0 (list 'quote _0)))))
      '(lambda (_0)
         (list 'quote (list 'quote (list _0 (list 'quote _0))))))
    '((lambda (_0)
        (list 'quote (list 'quote (list _0 (list 'quote _0)))))
      '(lambda (_0)
         (list 'quote (list 'quote (list _0 (list 'quote _0))))))
    ((lambda (_0)
       (list 'quote (list 'quote (list _0 (list 'quote _0)))))
     '(lambda (_0)
        (list 'quote (list 'quote (list _0 (list 'quote _0)))))))
  (≢ ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

6. Conclusion

Quines have a long and interesting history: the term "quine" was coined by Douglas Hofstadter (1979) in honor of the logician Willard van Orman Quine, but the concept goes back to Kleene's recursion theorems (Rogers 1967).

In section 4 we describe how the *absent^o* constraint can be used to distinguish general procedure application from uses of built-in primitives, and how this approach correctly handles shadowing of primitives. However, there are other ways to distinguish between application and uses of primitives. Our first efforts involved tagging procedure applications—that is, the Scheme expression (e_1 e_2) would be written in the interpreted language as (app e_1 e_2). Although this works, it is problematic in that generated programs are not quite Scheme programs. The tagging of application is especially problematic in the presence of **quote** and becomes most obvious when attempting to generate and interpret quines. A special "unparser" could be used to remove the app tags, making the answers readable. The tagless approach, however, allows the results of running the interpreter backwards to be pasted directly into the Scheme REPL and run without modification, while also allowing built-in forms to be shadowed.

Acknowledgments

Stuart Halloway asked if an earlier version of our relational interpreter could generate quines. His thirty-second question inspired the work described in this paper and resulted in significant improvements to miniKanren. We thank Stuart for his question, and look forward to his next challenge.

Many students in the Indiana University pl-wonks group joined us for an advanced programming languages course the semester after Stuart asked his question; one of the main topics of the course was running interpreters backwards. That group of students deserves our appreciation and thanks both for their wisdom and enthusiasm. Among that group was Claire Alvis, who was instrumental in implementing our constraint system cKanren, which allowed everyone in this course to build backward-running interpreters. She, too, deserves much credit. We thank Jason Hemann, Aaron Todd, and Cameron Swords for their involvement in developing software (some of which found its way into this paper) and their generally helpful comments. Jason also helped Dan with LaTeX and revision control issues, and was instrumental in improving appendix D.

None of this would have been possible without the incredibly useful and timely observations of Mitchell Wand and Steve Ganz. Oleg Kiselyov joined this effort early on and he has been a constant rudder, keeping us grounded to the mathematics of logic programming. Everything in miniKanren has been influenced by the ideas of Oleg. Please accept our thanks for all your help. Chung-chieh Shan offered to revise our implementation of miniKanren, which first appeared in the *The Reasoned Schemer*. Our later implementations, including cKanren, αKanren, and this newer miniKanren, have barely veered away from Chung-chieh's masterful code. We appreciate and thank him for this extraordinarily elegant code. As always, we thank Dorai Sitaram for SLaTeX. We also thank Ramana Kumar and the anonymous reviewers for their comments.

Working on a long-lived project, it is always difficult to recall all those we want to thank, but hopefully between the acknowledgments in our other papers and book we have not missed anyone.

References

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. cKanren: miniKanren with constraints. In *Workshop on Scheme and Functional Programming*, October 2011.

Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001. URL `citeseer.ist.psu.edu/baader99unification.html`.

Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.

Leo Broukhis, Simon Cooper, and Landon Curt Noll. The International Obfuscated C Code Contest. `http://www.ioccc.org/`.

William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* PhD thesis, Indiana University, 2009.

William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In Robby Findler, editor, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago Technical Report TR-2006-06, pages 105–117, 2006.

William E. Byrd and Daniel P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Universite Laval Technical Report DIUL-RT-0701*, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18 (8):453–457, August 1975.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer.* The MIT Press, Cambridge, MA, 2005.

Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, Montreal, Canada, September 18–21, 2000*, pages 186–197. ACM Press, 2000.

Douglas R. Hofstadter. *Gödel, Escher, Bach : an eternal golden braid.* Basic, 1979.

Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In Jacques Garrigue and Manuel Hermenegildo, editors, *Proceedings of the 9th International Symposium on Functional and Logic Programming*, volume 4989 of *LNCS*, pages 64–80. Springer, 2008.

David B. MacQueen, Philip Wadler, and Walid Taha. How to add laziness to a strict language without even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, September 1998. Baltimore, MD.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

Hartley Rogers, Jr. *Theory of recursive functions and effective computability.* McGraw-Hill, New York, NY, 1967.

Gary P. Thompson II. The quine page (self-reproducing code). `http://www.nyx.org/~gthompso/quine.htm`.

Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud, editor, *Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Nancy, France, September 16–19, 1985. Springer-Verlag.

Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January, 1992. ACM Press.

A. An Extended Interpreter

Here we present a fully relational interpreter for an uncurried Scheme with arithmetic operators, conditionals, and pairs; this interpreter generates quines more slowly than the one in section 5, due to the higher branching factor.

Instead of using Scheme numbers, *eval-exp*^{o} uses the relational arithmetic system described in appendix B.

```
(define eval-exp^o
  (lambda (exp env val)
    (cond^e
      ((fresh (v)
         (≡ `(quote ,v) exp)
         (not-in-env^o 'quote env)
         (absent^o 'closure v)
         (absent^o 'int-val v)
         (≡ v val)))
      ((fresh (a*)
         (≡ `(list . ,a*) exp)
         (not-in-env^o 'list env)
         (absent^o 'closure a*)
         (absent^o 'int-val a*)
         (proper-list^o a* env val)))
      ((prim-exp^o exp env val))
      ((symbol^o exp) (lookup^o exp env val))
      ((fresh (rator x* rands body env_2 a* res)
         (≡ `(,rator . ,rands) exp)
         (eval-exp^o rator env `(closure ,x* ,body ,env_2))
         (proper-list^o rands env a*)
         (ext-env*^o x* a* env_2 res)
         (eval-exp^o body res val)))
      ((fresh (x* body)
         (≡ `(lambda ,x* ,body) exp)
         (not-in-env^o 'lambda env)
         (≡ `(closure ,x* ,body ,env) val))))))
```

```
(define ext-env*^o
  (lambda (x* a* env out)
    (cond^e
      ((≡ '() x*) (≡ '() a*) (≡ env out))
      ((fresh (x a dx* da* env_2)
         (≡ `(,x . ,dx*) x*)
         (≡ `(,a . ,da*) a*)
         (≡ `((,x . ,a) . ,env) env_2)
         (ext-env*^o dx* da* env_2 out))))))
```

Primitive expressions include boolean literals, numbers (represented as tagged little-endian lists of bits), pairs, and operations over these values. The goal *prim-exp*^{o} dispatches to other goals to handle these primitives. The goals to handle *sub1*, *zero?*, $*$, *cons*, *car*, *cdr*, *not*, and **if**, rely on one or more mutually-recursive calls to *eval-exp*^{o}.

```
(define prim-exp^o
  (lambda (exp env val)
    (cond^e
      ((boolean-prim^o exp env val))
      ((number-prim^o exp env val))
      ((sub1-prim^o exp env val))
      ((zero?-prim^o exp env val))
      ((*-prim^o exp env val))
      ((cons-prim^o exp env val))
      ((car-prim^o exp env val))
      ((cdr-prim^o exp env val))
      ((not-prim^o exp env val))
      ((if-prim^o exp env val)))))
```

```
(define boolean-prim^o
  (lambda (exp env val)
    (cond^e
      ((≡ #t exp) (≡ #t val))
      ((≡ #f exp) (≡ #f val)))))
```

```
(define number-prim^o
  (lambda (exp env val)
    (fresh (n)
      (≡ `(int-exp ,n) exp)
      (≡ `(int-val ,n) val)
      (not-in-env^o 'int-exp env))))
```

```
(define sub1-prim^o
  (lambda (exp env val)
    (fresh (e n n-1)
      (≡ `(sub1 ,e) exp)
      (≡ `(int-val ,n-1) val)
      (not-in-env^o 'sub1 env)
      (eval-exp^o e env `(int-val ,n))
      (−^o n '(1) n-1))))
```

```
(define zero?-prim^o
  (lambda (exp env val)
    (fresh (e n)
      (≡ `(zero? ,e) exp)
      (cond^e
        ((zero^o n) (≡ #t val))
        ((pos^o n) (≡ #f val)))
      (not-in-env^o 'zero? env)
      (eval-exp^o e env `(int-val ,n)))))
```

```
(define *-prim^o
  (lambda (exp env val)
    (fresh (e_1 e_2 n_1 n_2 n_3)
      (≡ `(* ,e_1 ,e_2) exp)
      (≡ `(int-val ,n_3) val)
      (not-in-env^o '* env)
      (eval-exp^o e_1 env `(int-val ,n_1))
      (eval-exp^o e_2 env `(int-val ,n_2))
      (*^o n_1 n_2 n_3))))
```

```
(define cons-prim^o
  (lambda (exp env val)
    (fresh (a d v-a v-d)
      (≡ `(cons ,a ,d) exp)
      (≡ `(,v-a . ,v-d) val)
      (absent^o 'closure val)
      (absent^o 'int-val val)
      (not-in-env^o 'cons env)
      (eval-exp^o a env v-a)
      (eval-exp^o d env v-d))))
```

```
(define car-prim^o
  (lambda (exp env val)
    (fresh (p d)
      (≡ `(car ,p) exp)
      (≢ 'int-val val)
      (≢ 'closure val)
      (not-in-env^o 'car env)
      (eval-exp^o p env `(,val . ,d)))))

(define cdr-prim^o
  (lambda (exp env val)
    (fresh (p a)
      (≡ `(cdr ,p) exp)
      (≢ 'int-val a)
      (≢ 'closure a)
      (not-in-env^o 'cdr env)
      (eval-exp^o p env `(,a . ,val)))))

(define not-prim^o
  (lambda (exp env val)
    (fresh (e b)
      (≡ `(not ,e) exp)
      (cond^e
        ((≡ #t b) (≡ #f val))
        ((≡ #f b) (≡ #t val)))
      (not-in-env^o 'not env)
      (eval-exp^o e env b))))

(define if-prim^o
  (lambda (exp env val)
    (fresh (e_1 e_2 e_3 t)
      (≡ `(if ,e_1 ,e_2 ,e_3) exp)
      (not-in-env^o 'if env)
      (eval-exp^o e_1 env t)
      (cond^e
        ((≡ #t t) (eval-exp^o e_2 env val))
        ((≡ #f t) (eval-exp^o e_3 env val))))))
```

Now we can consider several examples using *eval-exp*^{o}. Consider this **run** expression, which returns 12 expressions that evaluate to six in the empty environment.

```
(run^12 (q) (eval-exp^o q '() `(int-val ,(build-num 6))))
⇒
((int-exp (0 1 1))
 ((lambda () (int-exp (0 1 1))))
 (sub1 (int-exp (1 1 1)))
 (((lambda (_0) (int-exp (0 1 1))) '_1)
  (≢ ((_0 int-exp)))
  (absent closure _1)
  (absent int-val _1))
 (∗ (int-exp (1)) (int-exp (0 1 1)))
 (∗ (int-exp (0 1 1)) (int-exp (1)))
 (∗ (int-exp (0 1)) (int-exp (1 1)))
 (((lambda (_0) (int-exp (0 1 1))) (list))
  (≢ ((_0 int-exp))))
 (car (list (int-exp (0 1 1))))
 ((lambda () ((lambda () (int-exp (0 1 1))))))
 (sub1 ((lambda () (int-exp (1 1 1)))))
 ((lambda () (sub1 (int-exp (1 1 1))))))
```

The 7th value in this list is

```
(∗ (int-exp (0 1)) (int-exp (1 1)))
```

And, if we look at the first 500 answers,

```
(run^500 (q) (eval-exp^o q '() `(int-val ,(build-num 6))))
```

we discover

```
(sub1 (sub1 (sub1 (int-exp (1 0 0 1)))))
```

is the 270th value.

Next, we calculate the factorial of five, using "The Poorman's Y Combinator."

```
(define rel-fact5
  `((lambda (f)
      ((f f) (int-exp ,(build-num 5))))
    (lambda (f)
      (lambda (n)
        (if (zero? n)
            (int-exp ,(build-num 1))
            (∗ n ((f f) (sub1 n))))))))

(run* (q) (eval-exp^o rel-fact5 '() q))
⇒ ((int-val (0 0 0 1 1 1 1)))
```

Now that we know our interpreter works, we are ready to generate quines in our extended language:

```
(run^5 (q) (eval-exp^o q '() q))
⇒
(#t
 #f
 (((lambda (_0) (list _0 (list 'quote _0)))
   '(lambda (_0) (list _0 (list 'quote _0))))
  (≢ ((_0 closure))
     ((_0 int-val))
     ((_0 list))
     ((_0 quote)))
  (sym _0))
 (((lambda (_0) (list _0 (list (car '(quote . _1)) _0)))
   '(lambda (_0) (list _0 (list (car '(quote . _1)) _0))))
  (≢ ((_0 car))
     ((_0 closure))
     ((_0 int-val))
     ((_0 list))
     ((_0 quote)))
  (absent closure _1)
  (absent int-val _1)
  (sym _0))
 (((lambda (_0) (list (list 'lambda '(_0) _0) (list 'quote _0)))
   '(list (list 'lambda '(_0) _0) (list 'quote _0)))
  (≢ ((_0 closure))
     ((_0 int-val))
     ((_0 list))
     ((_0 quote)))
  (sym _0)))
```

Not surprisingly, booleans are quines, since they are self-evaluating literals. The other answers are more interesting—we encourage the reader to look for patterns in the answers.

B. A Relational Arithmetic System

To make the paper self-contained, we present the relational arithmetic system used in the extended interpreter of appendix A. Variants of this arithmetic system have been described in Kiselyov et al. (2008) (with termination proofs for the individual operators), Byrd (2009), and Friedman et al. (2005)—please see these references for a detailed description of the code.

A note on typography: $+^o$ is entered as `pluso`, $-^o$ as `minuso`, $*^o$ as `*o`, and $\div^o$ as `/o`.

B.1 Relational Arithmetic

Relational arithmetic allows for queries such as 'what are five triples of natural numbers x, y, and z for which $x + y = z$?', and 'for which natural numbers x and y does $x \cdot y = 24$ hold?', which can be expressed in miniKanren as

```
(run^5 (q)
  (fresh (x y z)
    (+^o x y z)
    (≡ `(,x ,y ,z) q)))
```

and

```
(run^* (q)
  (fresh (x y)
    (*^o x y (build-num 24))
    (≡ `(,x ,y ,(build-num 24)) q)))
```

respectively.

In order to understand the answers to these **run**s, it is necessary to know how we represent numbers. Ground numbers are represented in "little endian" style using lists of bits. The most significant bit in the list cannot be 0; this restriction ensures each number has a unique representation. Zero is therefore represented by the empty list rather than (0), to avoid violating this restriction. The number one is represented by (1), the number two by (0 1), etc.

A number need not be ground, so long as it is not instantiated to a list ending with 0. For example, `(1 . ,$x$) represents any odd natural number, while `(0 . ,x) represents any *positive* even number (with the restriction that x must represent a positive integer, which we shall assume in the rest of this description). The list (0 0 0 1) represents the number 8, while `(0 0 0 . ,$x$) represents multiples of 8. If $x$ is (1), the multiple is just 8. If $x$ is (0 1), the multiple of 8 is 16, and so forth. Numbers can be even more sophisticated: `(0 ,y 0 . ,x) represents multiples of 8 if y is 0, and numbers of the form $8x + 2$ if y is 1.

Here are the answers to the **run⁵** expression above. The first answer states that for $x \geq 0$, $x + 0 = x$; the second answer states that for $x > 0$, $0 + x = x$; the third answer states that $1 + 1 = 2$; the fourth answer states that for $x > 0$ and $y \leq 1$, $1 + (4x + y) = 4x + y + 1$; and the fifth answer states that $1 + 3 = 4$.

```
((_.0 () _.0)
 (() (_.0 . _.1) (_.0 . _.1))
 ((1) (1) (0 1))
 ((1) (0 _.0 . _.1) (1 _.0 . _.1))
 ((1) (1 1) (0 0 1)))
```

And here are the answers to the **run*** expression above. There are eight answers, each one producing a different way to multiply two numbers to yield 24. For example, the fifth answer states that 8 times 3 is 24.

```
(((1) (0 0 0 1 1) (0 0 0 1 1))
 ((0 0 0 1 1) (1) (0 0 0 1 1))
 ((0 1) (0 0 1 1) (0 0 0 1 1))
 ((0 0 1) (0 1 1) (0 0 0 1 1))
 ((0 0 0 1) (1 1) (0 0 0 1 1))
 ((1 1) (0 0 0 1) (0 0 0 1 1))
 ((0 1 1) (0 0 1) (0 0 0 1 1))
 ((0 0 1 1) (0 1) (0 0 0 1 1)))
```

Our system includes other relational arithmetic operators: subtraction ($-^o$), integer division with remainder ($\div^o$), integer logarithm with remainder (log^o), and exponentiation (exp^o, which is derived from log^o).

B.2 Core Arithmetic Operators

This subsection contains arithmetic operators used in the extended interpreter in appendix A.

```
(define build-num
  (lambda (n)
    (cond
      ((odd? n)
       (cons 1
         (build-num (÷ (− n 1) 2))))
      ((and (not (zero? n)) (even? n))
       (cons 0
         (build-num (÷ n 2))))
      ((zero? n) '()))))

(define zero^o
  (lambda (n)
    (≡ '() n)))

(define pos^o
  (lambda (n)
    (fresh (a d)
      (≡ `(,a . ,d) n))))

(define >1^o
  (lambda (n)
    (fresh (a ad dd)
      (≡ `(,a ,ad . ,dd) n))))

(define full-adder^o
  (lambda (b x y r c)
    (cond^e
      ((≡ 0 b) (≡ 0 x) (≡ 0 y) (≡ 0 r) (≡ 0 c))
      ((≡ 1 b) (≡ 0 x) (≡ 0 y) (≡ 1 r) (≡ 0 c))
      ((≡ 0 b) (≡ 1 x) (≡ 0 y) (≡ 1 r) (≡ 0 c))
      ((≡ 1 b) (≡ 1 x) (≡ 0 y) (≡ 0 r) (≡ 1 c))
      ((≡ 0 b) (≡ 0 x) (≡ 1 y) (≡ 1 r) (≡ 0 c))
      ((≡ 1 b) (≡ 0 x) (≡ 1 y) (≡ 0 r) (≡ 1 c))
      ((≡ 0 b) (≡ 1 x) (≡ 1 y) (≡ 0 r) (≡ 1 c))
      ((≡ 1 b) (≡ 1 x) (≡ 1 y) (≡ 1 r) (≡ 1 c)))))

(define adder^o
  (lambda (d n m r)
    (cond^e
      ((≡ 0 d) (≡ '() m) (≡ n r))
      ((≡ 0 d) (≡ '() n) (≡ m r)
       (pos^o m))
      ((≡ 1 d) (≡ '() m)
       (adder^o 0 n '(1) r))
      ((≡ 1 d) (≡ '() n) (pos^o m)
       (adder^o 0 '(1) m r))
      ((≡ '(1) n) (≡ '(1) m)
       (fresh (a c)
         (≡ `(,a ,c) r)
         (full-adder^o d 1 1 a c)))
      ((≡ '(1) n) (gen-adder^o d n m r))
      ((≡ '(1) m) (>1^o n) (>1^o r)
       (adder^o d '(1) n r))
      ((>1^o n) (gen-adder^o d n m r)))))

(define gen-adder^o
  (lambda (d n m r)
    (fresh (a b c e x y z)
      (≡ `(,a . ,x) n)
      (≡ `(,b . ,y) m) (pos^o y)
      (≡ `(,c . ,z) r) (pos^o z)
      (full-adder^o d a b c e)
      (adder^o e x y z))))

(define +^o (lambda (n m k) (adder^o 0 n m k)))

(define −^o (lambda (n m k) (+^o m k n)))
```

```
(define *o
  (lambda (n m p)
    (conde
      ((≡ '() n) (≡ '() p))
      ((poso n) (≡ '() m) (≡ '() p))
      ((≡ '(1) n) (poso m) (≡ m p))
      ((>1o n) (≡ '(1) m) (≡ n p))
      ((fresh (x z)
         (≡ `(0 . ,x) n) (poso x)
         (≡ `(0 . ,z) p) (poso z)
         (>1o m)
         (*o x m z)))
      ((fresh (x y)
         (≡ `(1 . ,x) n) (poso x)
         (≡ `(0 . ,y) m) (poso y)
         (*o m n p)))
      ((fresh (x y)
         (≡ `(1 . ,x) n) (poso x)
         (≡ `(1 . ,y) m) (poso y)
         (odd-*o x n m p))))))

(define odd-*o
  (lambda (x n m p)
    (fresh (q)
      (bound-*o q p n m)
      (*o x m q)
      (+o `(0 . ,q) m p))))

(define bound-*o
  (lambda (q p n m)
    (conde
      ((≡ '() q) (poso p))
      ((fresh (a_0 a_1 a_2 a_3 x y z)
         (≡ `(,a_0 . ,x) q)
         (≡ `(,a_1 . ,y) p)
         (conde
           ((≡ '() n) (≡ `(,a_2 . ,z) m)
            (bound-*o x y z '()))
           ((≡ `(,a_3 . ,z) n) (bound-*o x y z m))))))))
```

B.3 Additional Arithmetic Operators

This subsection contains useful arithmetic operators, beyond those used in the extended interpreter in appendix A.

```
(define =lo
  (lambda (n m)
    (conde
      ((≡ '() n) (≡ '() m))
      ((≡ '(1) n) (≡ '(1) m))
      ((fresh (a x b y)
         (≡ `(,a . ,x) n) (poso x)
         (≡ `(,b . ,y) m) (poso y)
         (=lo x y))))))

(define <lo
  (lambda (n m)
    (conde
      ((≡ '() n) (poso m))
      ((≡ '(1) n) (>1o m))
      ((fresh (a x b y)
         (≡ `(,a . ,x) n) (poso x)
         (≡ `(,b . ,y) m) (poso y)
         (<lo x y))))))

(define ⩽lo
  (lambda (n m)
    (conde
      ((=lo n m))
      ((<lo n m)))))

(define <o
  (lambda (n m)
    (conde
      ((<lo n m))
      ((=lo n m)
       (fresh (x)
         (poso x)
         (+o n x m))))))

(define ⩽o
  (lambda (n m)
    (conde
      ((≡ n m))
      ((<o n m)))))

(define ÷o
  (lambda (n m q r)
    (conde
      ((≡ r n) (≡ '() q) (<o n m))
      ((≡ '(1) q) (=lo n m) (+o r m n) (<o r m))
      ((<lo m n) (<o r m) (poso q)
       (fresh (n_h n_l q_h q_l qlm qlmr rr r_h)
         (splito n r n_l n_h)
         (splito q r q_l q_h)
         (conde
           ((≡ '() n_h) (≡ '() q_h)
            (−o n_l r qlm)
            (*o q_l m qlm))
           ((poso n_h)
            (*o q_l m qlm)
            (+o qlm r qlmr)
            (−o qlmr n_l rr)
            (splito rr r '() r_h)
            (÷o n_h m q_h r_h)))))))))

(define splito
  (lambda (n r l h)
    (conde
      ((≡ '() n) (≡ '() h) (≡ '() l))
      ((fresh (b n̂)
         (≡ `(0 ,b . ,n̂) n)
         (≡ '() r)
         (≡ `(,b . ,n̂) h)
         (≡ '() l)))
      ((fresh (n̂)
         (≡ `(1 . ,n̂) n)
         (≡ '() r)
         (≡ n̂ h)
         (≡ '(1) l)))
      ((fresh (b n̂ a r̂)
         (≡ `(0 ,b . ,n̂) n)
         (≡ `(,a . ,r̂) r)
         (≡ '() l)
         (splito `(,b . ,n̂) r̂ '() h)))
      ((fresh (n̂ a r̂)
         (≡ `(1 . ,n̂) n)
         (≡ `(,a . ,r̂) r)
         (≡ '(1) l)
         (splito n̂ r̂ '() h)))
      ((fresh (b n̂ a r̂ l̂)
         (≡ `(,b . ,n̂) n)
         (≡ `(,a . ,r̂) r)
         (≡ `(,b . ,l̂) l)
         (poso l̂)
         (splito n̂ r̂ l̂ h))))))
```

```
(define log^o
  (lambda (n b q r)
    (cond^e
      ((≡ '(1) n) (pos^o b) (≡ '() q) (≡ '() r))
      ((≡ '() q) (<^o n b) (+^o r '(1) n))
      ((≡ '(1) q) (>1^o b) (=l^o n b) (+^o r b n))
      ((≡ '(1) b) (pos^o q) (+^o r '(1) n))
      ((≡ '() b) (pos^o q) (≡ r n))
      ((≡ '(0 1) b)
       (fresh (a ad dd)
         (pos^o dd)
         (≡ `(,a ,ad . ,dd) n)
         (exp2^o n '() q)
         (fresh (s)
           (split^o n dd r s))))
      ((fresh (a ad add ddd)
         (cond^e
           ((≡ '(1 1) b))
           ((≡ `(,a ,ad ,add . ,ddd) b))))
       (<l^o b n)
       (fresh (bw1 bw nw nw1 ql1 q_l s)
         (exp2^o b '() bw1)
         (+^o bw1 '(1) bw)
         (<l^o q n)
         (fresh (q_1 bwq1)
           (+^o q '(1) q_1)
           (*^o bw q_1 bwq1)
           (<^o nw1 bwq1))
         (exp2^o n '() nw1)
         (+^o nw1 '(1) nw)
         (÷^o nw bw ql1 s)
         (+^o q_l '(1) ql1)
         (≤l^o q_l q)
         (fresh (bql q_h s qdh qd)
           (repeated-mul^o b q_l bql)
           (÷^o nw bw1 q_h s)
           (+^o q_l qdh q_h)
           (+^o q_l qd q)
           (≤^o qd qdh)
           (fresh (bqd bq1 bq)
             (repeated-mul^o b qd bqd)
             (*^o bql bqd bq)
             (*^o b bq bq1)
             (+^o bq r n)
             (<^o n bq1))))))))

(define exp2^o
  (lambda (n b q)
    (cond^e
      ((≡ '(1) n) (≡ '() q))
      ((>1^o n) (≡ '(1) q)
       (fresh (s)
         (split^o n b s '(1))))
      ((fresh (q_1 b_2)
         (≡ `(0 . ,q_1) q)
         (pos^o q_1)
         (<l^o b n)
         (append^o b `(1 . ,b) b_2)
         (exp2^o n b_2 q_1)))
      ((fresh (q_1 n_h b_2 s)
         (≡ `(1 . ,q_1) q)
         (pos^o q_1)
         (pos^o n_h)
         (split^o n b s n_h)
         (append^o b `(1 . ,b) b_2)
         (exp2^o n_h b_2 q_1))))))

(define repeated-mul^o
  (lambda (n q nq)
    (cond^e
      ((pos^o n) (≡ '() q) (≡ '(1) nq))
      ((≡ '(1) q) (≡ n nq))
      ((>1^o q)
       (fresh (q_1 nq1)
         (+^o q_1 '(1) q)
         (repeated-mul^o n q_1 nq1)
         (*^o nq1 n nq))))))

(define exp^o
  (lambda (b q n)
    (log^o n b q '())))
```

C. Generalized Pattern Matcher

This appendix describes and defines the **dmatch** pattern matcher, which is a generalization of Oleg Kiselyov's **pmatch** that appeared in Byrd and Friedman (2007). **dmatch** improves error reporting, since now it is possible to associate a string with each use of pattern matching, as with the name "example" in the definition of *h* below. **dmatch** does not handle **quote** specially, which allows for certain common patterns to be specified that were previously not possible. Finally, **dmatch** does not support the **else** auxiliary keyword, and the order of the clauses is arbitrary—however, only one pattern (plus guard) can succeed for each invocation of **dmatch**. Here is an example of **dmatch**.

```
(define h
  (lambda (x y)
    (dmatch `(,x . ,y) "example"
      ((,a . ,b)
       (guard (number? a) (number? b))
       (* a b))
      ((,a ,b ,c)
       (guard (number? a) (number? b) (number? c))
       (+ a b c)))))

(list (h 3 4) (apply h `(1 (3 4)))) ⇒ (12 8)
```

In this example, a dotted pair is matched against two different patterns. In the first clause, the value of *x* is lexically bound to *a* and the value of *y* is lexically bound to *b*. Before the pattern match succeeds, however, an optional side-effect-free guard is run within the scope of *a* and *b*. The guard succeeds only if *a* and *b* are bound to numbers; if so, then their product is returned. The second clause attempts to match the dotted pair against a three-element list, once again with an optional guard. If the values bound to *a*, *b*, and *c* are all numbers, the second clause returns their sum.

Here is the grammar for **dmatch**, where *exp* is any pure Scheme expression, *boolean-exp* is any pure Scheme predicate expression, *var* is any valid Scheme identifier, literal is any Scheme literal, and name-string is any literal Scheme string:

```
  match := (dmatch exp {name-string} clause ...)
 clause := (pattern {guard} exp ...)
  guard := (guard boolean-exp ...)
pattern := ,var
         | literal
         | (pattern_1 pattern_2 ...)
         | (pattern_1 . pattern_2)
```

Now we examine the implementation of **dmatch**. The main **dmatch** macro simply handles the optional name string, and passes off control to the auxiliary helpers. The auxiliary macros will produce a list of "packages," which is then processed by the *run-a-thunk* procedure.

```
(define-syntax dmatch
  (syntax-rules ()
    ((_ v (e ...) ...)
     (let ((pkg* (dmatch-remexp v (e ...) ...)))
       (run-a-thunk 'v v #f pkg*)))
    ((_ v name (e ...) ...)
     (let ((pkg* (dmatch-remexp v (e ...) ...)))
       (run-a-thunk 'v v 'name pkg*)))))
```

A package comprises a clause and a thunk, and is constructed/destructed using these functions:

```
(define pkg (lambda (cls thk) (cons cls thk)))
(define pkg-clause (lambda (pkg) (car pkg)))
(define pkg-thunk (lambda (pkg) (cdr pkg)))
```

dmatch-remexp ensures that the input expression to **dmatch** is evaluated only once.

```
(define-syntax dmatch-remexp
  (syntax-rules ()
    ((_ (rator rand ...) cls ...)
     (let ((v (rator rand ...)))
       (dmatch-aux v cls ...)))
    ((_ v cls ...) (dmatch-aux v cls ...))))
```

Each expansion of **dmatch-aux** creates a package list of some type. There are three cases: two recursive cases and a single base case. If a pattern without a guard matches the input, that clause and its thunk are added to the package list. If a matching pattern has a guard, the clause and thunk are added to the package list only if the guard also succeeds.

```
(define-syntax dmatch-aux
  (syntax-rules (guard)
    ((_ v) '())
    ((_ v (pat (guard g ...) e0 e ...) cs ...)
     (let ((fk (lambda () (dmatch-aux v cs ...))))
       (ppat v pat
         (if (not (and g ...))
           (fk)
           (cons (pkg '(pat (guard g ...) e0 e ...)
                      (lambda () e0 e ...))
                 (fk)))
         (fk))))
    ((_ v (pat e0 e ...) cs ...)
     (let ((fk (lambda () (dmatch-aux v cs ...))))
       (ppat v pat
         (cons (pkg '(pat e0 e ...)
                    (lambda () e0 e ...))
               (fk))
         (fk))))))
```

The **ppat** helper macro does the actual pattern matching, then expands into one of two forms. The consequent expression is the result of the expansion of **ppat** if the pattern matches, and the alternate expression otherwise. In all cases, the alternate is just another **dmatch-aux** macro that drops the first pattern and continues the recursive expansion. The alternative is encoded as a thunk, to avoid expanding the same expression multiple times.

ppat leverages the **syntax-rules** pattern matcher to do most of the work. The pair case performs tree recursion to match against the *car* and *cdr*. The last clause uses *equal?* rather than *eq?* in order to handle vectors and other data.

```
(define-syntax ppat
  (syntax-rules (unquote)
    ((_ v (unquote var) kt kf) (let ((var v)) kt))
    ((_ v (x . y) kt kf)
     (if (pair? v)
       (let ((vx (car v)) (vy (cdr v)))
         (ppat vx x (ppat vy y kt kf) kf))
       kf))
    ((_ v lit kt kf) (if (equal? v (quote lit)) kt kf))))
```

If there is no match, the error is reported by *no-matching-pattern* (using the non-standard *printf* function). If there is an overlap between two or more patterns/guards, then *overlapping-patterns/guards* signals an error. Otherwise, if there is no overlap, the thunk in the singleton package list is invoked.

```
(define run-a-thunk
  (lambda (v-expr v name pkg*)
    (cond
      ((null? pkg*) (no-matching-pattern name v-expr v))
      ((null? (cdr pkg*)) ((pkg-thunk (car pkg*))))
      (else
       (ambiguous-pattern/guard name v-expr v pkg*)))))

(define no-matching-pattern
  (lambda (name v-expr v)
    (if name
      (printf "dmatch ~d failed~n~d ~d~n"
              name v-expr v)
      (printf "dmatch failed~n~d ~d~n"
              v-expr v))
    (error 'dmatch "match failed")))

(define overlapping-patterns/guards
  (lambda (name v-expr v pkg*)
    (if name
      (printf "dmatch ~d overlapping matching clauses~n"
              name)
      (printf "dmatch overlapping matching clauses~n"))
    (printf "with ~d evaluating to ~d~n" v-expr v)
    (printf "____________________________________~n")
    (for-each pretty-print (map pkg-clause pkg*))))
```

Here is the definition of *h* (eliding the second clause) after macro expansion.

```
(lambda (x y)
  (let ((pkg*
    (let ((v (cons x y)))
      (let ((fk (lambda () ...)))
        (if (pair? v)
          (let ((vx (car v)) (vy (cdr v)))
            (let ((a vx))
              (let ((b vy))
                (if (not (if (number? a) (number? b) #f))
                  (fk)
                  (cons
                    (pkg
                      '((,a . ,b)
                        (guard
                          (number? a)
                          (number? b))
                        (* a b))
                      (lambda () (* a b)))
                    (fk))))))
          (fk))))))
    (run-a-thunk '(,x . ,y) (cons x y) "example" pkg*)))
```

There are two kinds of improvements that should be resolved by the compiler. First, *vx* and *vy* are not needed, so they should not get bindings. The lexical variables *a* and *b* could have replaced *vx* and *vy*, respectively. Second, *a* and *b* should be parallel **let** bindings.

D. miniKanren Implementation

Our miniKanren implementation comprises two kinds of operators: the interface operators **run** and **run***; and goal constructors $\equiv$, $\not\equiv$, *symbol*^{o}, *number*^{o}, *absent*^{o}, **cond**^{e}, and **fresh**, which take a *package* implicitly.

A package is a list of four values, each of which is, or contains, an association list of variables to values. The first value in a package is a substitution, S. The second value in a package is a list of association lists, D; each association list, d, represents a *disequality constraint*. The third value in a package is an association list, A, that associates a variable with a pair consisting of a tag and a predicate. If a variable, say x, is associated with the tag `sym`, then we know that x may only be associated in the substitution with either a fresh variable or a symbol. Any attempt to associate x with any other kind of value leads to failure. A has within it constraints that are used to support *symbol*^{o} and *number*^{o}. The final value in a package is also an association list, T, whose members associate a variable with a pair consisting of a tag and a predicate. T has within it constraints that are used to support *absent*^{o}.

```
(define c→S (lambda (c) (car c)))
(define c→D (lambda (c) (cadr c)))
(define c→A (lambda (c) (caddr c)))
(define c→T (lambda (c) (cadddr c)))
(define empty-c '(() () () ()))
```

A goal g is a function that maps a package c to an ordered sequence c^∞ of zero or more packages. (For clarity, we notate **lambda** as λ_{G} when creating such a function g.)

```
(define-syntax λG
  (syntax-rules (:)
    ((_ (c) e) (lambda (c) e))
    ((_ (c : S D A T) e)
     (lambda (c)
       (let ((S (c→S c))
             (D (c→D c))
             (A (c→A c))
             (T (c→T c)))
         e)))))
```

Because a sequence of packages may be infinite, we represent it not as a list but as a c^∞, a special kind of stream that can contain either precisely zero, precisely one, or one or more packages (Hinze 2000; Wadler 1985). *mzero*, and *unit*, represent these first two options. We use `#f` to denote the empty stream of packages. If c is a package, then c itself represents the stream containing just c.

```
(define mzero (lambda () #f))
(define unit (λG (c) c))
(define choice (lambda (c f) (cons c f)))
```

To represent a stream containing multiple packages, we use (*choice* c f), where c is the first package in the stream, and where f is a thunk that, when invoked, produces the remainder of the stream. (For clarity, we notate **lambda** as λ_{F} when creating such a function f.) To represent an incomplete stream, we use (**inc** e), where e is an *expression* that evaluates to a c^∞—thus **inc** creates an f.

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

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

(define empty-f (λF () (mzero)))
```

A singleton stream c is the same as (*choice* c *empty-f*). For goals that return only a single package, however, using this special representation of a singleton stream avoids the cost of unnecessarily building and taking apart pairs, and creating and invoking thunks.

To ensure that the values produced by these four kinds of c^∞'s can be distinguished, we assume that a package 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 (() e0) ((f̂) e1) ((ĉ) e2) ((c f) e3))
     (let ((c∞ e))
       (cond
         ((not c∞) e0)
         ((procedure? c∞) (let ((f̂ c∞)) e1))
         ((not (and (pair? c∞)
                    (procedure? (cdr c∞))))
          (let ((ĉ c∞)) e2))
         (else (let ((c (car c∞)) (f (cdr c∞)))
                 e3)))))))
```

To get answers from the potentially infinite stream, we use the **run** interface. The interface operator **run** uses *take* to convert an f to an *even* stream (MacQueen et al. 1998). The definition of **run** places an artificial goal at the tail of g_0 g ... This artificial goal invokes *reify* (section 2.1) on the variable x using the final package *final-c* produced by running all the goals in the empty package *empty-c*.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (λF ()
         ((fresh (x) g0 g ...
            (λG (final-c)
              (let ((z ((reify x) final-c)))
                (choice z empty-f))))
          empty-c))))))

(define-syntax run*
  (syntax-rules ()
    ((_ (x) g ...) (run #f (x) g ...))))
```

If the first argument to *take* is `#f`, then *take* returns the entire stream of reified values as a list, thereby providing the behavior of **run***. The **and** expressions within *take* detect this `#f` case.

```
(define take
  (lambda (n f)
    (cond
      ((and n (zero? n)) '())
      (else
       (case∞ (f)
         (() '())
         ((f) (take n f))
         ((c) (cons c '()))
         ((c f) (cons c (take (and n (− n 1)) f))))))))
```

D.1 Goal Constructors

To take the conjunction of goals, we define **fresh**, a goal constructor that first lexically binds variables built by *var* and then combines successive goals using **bind***.

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

bind* is short-circuiting, since the empty stream is represented by #f. **bind*** relies on *bind* (Moggi 1991; Wadler 1992), which applies the goal g to each element in the stream c^∞. The resulting c^∞'s are then merged using *mplus*, which combines a c^∞ and an f to yield a single c^∞.

```
(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
    ((_ e g0 g ...) (bind* (bind e g0) g ...))))

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

(define mplus
  (lambda (c∞ f)
    (case∞ c∞
      (() (f))
      ((f^) (inc (mplus (f) f^)))
      ((c) (choice c f))
      ((c f^) (choice c (λF () (mplus (f) f^)))))))
```

To take the disjunction of goals we define **cond**^{e}, a goal constructor that combines successive **cond**^{e} lines using **mplus***, which in turn relies on *mplus*. We use the same implicit package c for each **cond**^{e} line. To avoid divergence, we treat the lines of each **cond**^{e} as a single **inc** stream.

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

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

The pattern structure of **case-value** is similar to **case**$^\infty$ in that it lists the scoped variables. If u's value is a variable, the scope of e_0 includes u's value. If u's value is a pair, then the scope of e_1 includes each item in the pair. Otherwise, the scope of e_2 includes the value of u.

```
(define-syntax case-value
  (syntax-rules ()
    ((_ u ((t1) e0) ((at dt) e1) ((t2) e2))
     (let ((t u))
       (cond
         ((var? t) (let ((t1 t)) e0))
         ((pair? t) (let ((at (car t)) (dt (cdr t))) e1))
         (else (let ((t2 t)) e2)))))))
```

The function *make-tag-A* is used to create the *symbol*^{o} and *number*^{o} goal constructors and contains the essence of what those operators accomplish. Elements of A act as daemons. These daemons make certain that associations which are added to the substitution do not violate A's constraints. In addition, D may contain a disequality constraint between, say, a variable y and 3; if we also know that y must be a symbol, then the disequality constraint is subsumed by the symbol constraint on y, and can be discarded.

```
(define make-tag-A
  (lambda (tag pred)
    (lambda (u)
      (λG (c : S D A T)
        (case-value (walk u S)
          ((x) (cond
                 ((make-tag-A+ x tag pred c S D A T)
                  ⇒ unit)
                 (else (mzero))))
          ((au du) (mzero))
          ((u) (cond
                 ((pred u) (unit c))
                 (else (mzero)))))))))

(define make-tag-A+
  (lambda (u tag pred c S D A T)
    (cond
      ((ext-A (walk u S) tag pred S A) ⇒
       (lambda (A+)
         (cond
           ((null? A+) c)
           (else (let ((D (subsume A+ D))
                       (A (append A+ A)))
                   (subsume-A S D A T))))))
      (else #f))))

(define subsume-A
  (lambda (S D A T)
    (let ((x* (rem-dups (map lhs A))))
      (subsume-A+ x* S D A T))))

(define subsume-A+
  (lambda (x* S D A T)
    (cond
      ((null? x*) `(,S ,D ,A ,T))
      (else (let ((x (car x*)))
              (let ((D/T (update-D/T x S D A T)))
                (let ((D (car D/T)) (T (cdr D/T)))
                  `(,S ,D ,A ,T))))))))

(define ext-A
  (lambda (x tag pred S A)
    (cond
      ((null? A) `((,x . (,tag . ,pred))))
      (else
       (let ((a (car A)) (A (cdr A)))
         (let ((a-tag (pr→tag a)))
           (cond
             ((eq? (walk (lhs a) S) x)
              (cond
                ((tag=? a-tag tag) '())
                (else #f)))
             (else (ext-A x tag pred S A)))))))))

(define booleano
  (lambda (x)
    (conde
      ((≡ #f x))
      ((≡ #t x)))))
```

```
(define symbol^o (make-tag-A 'sym symbol?))

(define number^o (make-tag-A 'num number?))

(define pr→tag (lambda (pr) (car (rhs pr))))

(define pr→pred (lambda (pr) (cdr (rhs pr))))
```

Here is the implementation of the remaining goal constructors. The definitions of ≢ and ≡ both use *unify* (section D.3). But, when we succeed by invoking *unit*, we pass a different substitution. In the ≡ case, we pass the (possibly) extended substitution, however, in the ≢ case, we pass the original substitution. In the ≢ case, the actual extension (here called the *prefix*) is a disequality constraint. We can take that constraint and make sure that A and T are okay with each association in the prefix. Those associations that are not dropped from the prefix are added as a new constraint to D. (There is a subtlety in the simplicity of the definition of *prefix-S*: we know that if we keep taking *cdr*s starting at $S+$, assuming that $S+$ and S are not *eq?*, we will eventually arrive at S. This *eq?* is not strictly necessary, since we are basically trying to determine if the lengths of the two lists are the same but more efficiently.)

```
(define ≢
  (lambda (u v)
    (λ_G (c : S D A T)
      (cond
        ((unify u v S) ⇒ (post-unify-≢ S D A T))
        (else (unit c))))))

(define post-unify-≢
  (lambda (S D A T)
    (lambda (S+)
      (cond
        ((eq? S+ S) (mzero))
        (else (let ((D+ (list (prefix-S S+ S))))
                (let ((D+ (subsume A D+)))
                  (let ((D+ (subsume T D+)))
                    (let ((D (append D+ D)))
                      (unit `(,S ,D ,A ,T)))))))))))

(define prefix-S
  (lambda (S+ S)
    (cond
      ((eq? S+ S) '())
      (else (cons (car S+) (prefix-S (cdr S+) S))))))

(define subsume
  (lambda (A/T D)
    (remp (lambda (d) (exists (subsumed-pr? A/T) d))
      D)))

(define subsumed-pr?
  (lambda (A/T)
    (lambda (pr-d)
      (let ((u (rhs pr-d)))
        (cond
          ((var? u) #f)
          (else
           (let ((pr (assq (lhs pr-d) A/T)))
             (and pr
               (let ((tag (pr→tag pr)))
                 (cond
                   ((and (tag? tag)
                         (tag? u)
                         (tag=? u tag)))
                   (((pr→pred pr) u) #f)
                   (else #t)))))))))))
```

Just as ≢ checks A and T before extending D, ≡ must first check D, A, and T (all of which might change) before succeeding.

```
(define ≡
  (lambda (u v)
    (λ_G (c : S D A T)
      (cond
        ((unify u v S) ⇒
         (post-unify-≡ c S D A T))
        (else (mzero))))))

(define post-unify-≡
  (lambda (c S D A T)
    (lambda (S+)
      (cond
        ((eq? S+ S) (unit c))
        ((verify-D D S+) ⇒
         (lambda (D)
           (cond
             ((post-verify-D S+ D A T) ⇒ unit)
             (else (mzero)))))
        (else (mzero))))))

(define verify-D
  (lambda (D S)
    (cond
      ((null? D) '())
      ((verify-D (cdr D) S) ⇒
       (lambda (D+)
         (verify-D+ (car D) D+ S)))
      (else #f))))

(define verify-D+
  (lambda (d D S)
    (cond
      ((unify* d S) ⇒
       (lambda (S+)
         (cond
           ((eq? S+ S) #f)
           (else (cons (prefix-S S+ S) D)))))
      (else D))))

(define post-verify-D
  (lambda (S D A T)
    (cond
      ((verify-A A S) ⇒
       (post-verify-A S D T))
      (else #f))))

(define verify-A
  (lambda (A S)
    (cond
      ((null? A) '())
      ((verify-A (cdr A) S) ⇒
       (lambda (A_0)
         (let ((u (walk (lhs (car A)) S))
               (tag (pr→tag (car A)))
               (pred (pr→pred (car A))))
           (cond
             ((var? u)
              (cond
                ((ext-A u tag pred S A_0) ⇒
                 (lambda (A+)
                   (append A+ A_0)))
                (else #f)))
             (else (and (pred u) A_0))))))
      (else #f))))
```

```
(define post-verify-A
  (lambda (S D T)
    (lambda (A)
      (let ((D (subsume A D)))
        (cond
          ((verify-T T S) ⇒ (post-verify-T S D A))
          (else #f))))))

(define verify-T
  (lambda (T S)
    (cond
      ((null? T) '())
      ((verify-T (cdr T) S)
       ⇒ (verify-T+ (lhs (car T)) T S))
      (else #f))))

(define verify-T+
  (lambda (x T S)
    (lambda (T0)
      (let ((tag (pr→tag (car T)))
            (pred (pr→pred (car T))))
        (case-value (walk x S)
          ((x) (cond
                 ((ext-T+ x tag pred S T0) ⇒
                  (lambda (T+) (append T+ T0)))
                 (else #f)))
          ((au du) (cond
                     (((verify-T+ au T S) T0) ⇒
                      (verify-T+ du T S))
                     (else #f)))
          ((u) (and (pred u) T0)))))))

(define post-verify-T
  (lambda (S D A)
    (lambda (T)
      (subsume-T T S (subsume T D) A '()))))

(define subsume-T
  (lambda (T+ S D A T)
    (let ((x* (rem-dups (map lhs A))))
      (subsume-T+ x* T+ S D A T))))

(define subsume-T+
  (lambda (x* T+ S D A T)
    (cond
      ((null? x*) (let ((T (append T+ T)))
                    `(,S ,D ,A ,T)))
      (else (let ((x (car x*)) (x* (cdr x*)))
              (let ((D/T (update-D/T x S D A T+)))
                (let ((D (car D/T)) (T+ (cdr D/T)))
                  (subsume-T+ x* T+ S D A T))))))))

(define update-D/T
  (lambda (x S D A T)
    (cond
      ((null? A) (let ((T (remp (lambda (t)
                                  (eq? (lhs t) x))
                                T)))
                   `(,D . ,T)))
      (else
       (let ((a (car A)))
         (cond
           ((and (eq? (lhs a) x)
                 (or (tag=? (pr→tag a) 'sym)
                     (tag=? (pr→tag a) 'num)))
            (update-D/T+ x '() S D T))
           (else (update-D/T x S D (cdr A) T))))))))

(define update-D/T+
  (lambda (x T+ S D T)
    (cond
      ((null? T)
       `(,D . ,T+))
      (else
       (let ((t (car T))
             (T (cdr T)))
         (cond
           ((eq? (lhs t) x)
            (let ((D (ext-D x (pr→tag t) D S)))
              (update-D/T+ x T+ S D T)))
           (else
            (let ((T+ (cons t T+)))
              (update-D/T+ x T+ S D T))))))))))

(define ext-D
  (lambda (x tag D S)
    (cond
      ((exists
         (lambda (d)
           (and (null? (cdr d))
             (let ((y (lhs (car d)))
                   (u (rhs (car d))))
               (and
                 (eq? (walk y S) x)
                 (tag? u)
                 (tag=? u tag)))))
         D)
       D)
      (else (cons `((,x . ,tag)) D)))))
```

The final goal constructor, *absent*^{o}, takes a *tag*, which must be a symbol, and a term u (possibly a variable), and generates a constraint requiring that that tag not occur within the term.

The main driver of *absent*^{o} is *absento+*, which takes care of the three possible types of terms: variables, pairs, and ground values.

If the term is a variable, we add the constraint provided it is not already present. For ground values, we ensure that the ground value does not in fact match the tag, which would violate the constraint. For pairs, we recur on the *car* and the *cdr*, which might themselves generate further constraints. This guarantees that every pair has a left-hand side that is a variable and a right-hand side that contains both the tag and a predicate that determines whether other tags do not match (using the value returned by *make-pred-T*).

ext-T is the helper which actually adds the newly-formed constraint to T, provided it is not already present.

```
(define make-pred-T
  (lambda (tag)
    (lambda (x)
      (not (and (tag? x) (tag=? x tag))))))

(define absent^o
  (lambda (tag u)
    (cond
      ((not (tag? tag)) fail)
      (else
       (λG (c : S D A T)
         (cond
           ((absento+ u tag c S D A T)
            ⇒ unit)
           (else (mzero))))))))
```

```
(define absento+
  (lambda (u tag c S D A T)
    (case-value (walk u S)
      ((x)
       (let ((T+ (ext-T x tag S T)))
         (cond
           ((null? T+) c)
           (else
            (let ((D (subsume T+ D)))
              (subsume-T T+ S D A T))))))
      ((au du)
       (let ((c (absento+ au tag c S D A T)))
         (and c
           (let ((S (c→S c))
                 (D (c→D c))
                 (A (c→A c))
                 (T (c→T c)))
             (absento+ du tag c S D A T)))))
      ((u)
       (cond
         ((and (tag? u) (tag=? u tag)) #f)
         (else c))))))

(define ext-T
  (lambda (x tag S T)
    (cond
      ((null? T)
       (let ((pred (make-pred-T tag)))
         `((,x . (,tag . ,pred)))))
      (else
       (let ((t (car T)) (T (cdr T)))
         (let ((t-tag (pr→tag t)))
           (cond
             ((eq? (walk (lhs t) S) x)
              (cond
                ((tag=? t-tag tag) '())
                (else (ext-T x tag S T))))
             ((tag=? t-tag tag)
              (let ((t-pred (pr→pred t)))
                (ext-T+ x tag t-pred S T)))
             (else (ext-T x tag S T)))))))))

(define ext-T+
  (lambda (x tag pred S T)
    (cond
      ((null? T) `((,x . (,tag . ,pred))))
      (else
       (let ((t (car T)))
         (let ((t-tag (pr→tag t)))
           (cond
             ((eq? (walk (lhs t) S) x)
              (cond
                ((tag=? t-tag tag) '())
                (else
                 (ext-T+ x tag pred S
                   (cdr T)))))
             (else
              (ext-T+ x tag pred S
                (cdr T))))))))))
```

D.2 miniKanren Helpers

We have chosen vectors to represent logic variables merely as a simple way to distinguish variables from other valid datatypes. This could have been avoided any number of ways. For example, each variable could have been given a unique index.

The function *rem-dups* generates a list of unique logic variables. It is used in two places, both times merely to avoid re-doing computations.

```
(define var (lambda (dummy) (vector dummy)))
(define var? (lambda (x) (vector? x)))

(define rem-dups
  (lambda (x*)
    (cond
      ((null? x*) '())
      ((memq (car x*) (cdr x*))
       (rem-dups (cdr x*)))
      (else (cons (car x*)
              (rem-dups (cdr x*)))))))

(define tag?
  (lambda (tag)
    (symbol? tag)))

(define tag=?
  (lambda (tag1 tag2)
    (eq? tag1 tag2)))

(define lhs (lambda (pr) (car pr)))
(define rhs (lambda (pr) (cdr pr)))

(define succeed (≡ #f #f))
(define fail (≡ #f #t))
```

This definition of *walk* assumes a simple representation of substitutions. Various persistent structures, such as those in (Okasaki 1999), would improve the performance.

```
(define walk
  (lambda (u S)
    (cond
      ((and (var? u) (assq u S)) ⇒
       (lambda (pr) (walk (rhs pr) S)))
      (else u))))
```

D.3 The Unifier

Below is *unify*, which uses triangular substitutions (Baader and Snyder 2001) instead of the more common idempotent substitutions. After possibly walking the first two arguments to get a representative, the two-pairs case is treated. Otherwise, if there are not two pairs, then *unify-nonpair* gets the two representatives, which might extend the substitution. There is no explicit recursion in *unify-nonpair*, but *unify-nonpair* calls *ext-S*, which calls a recursive function, *occurs*$^{\checkmark}$.

```
(define unify
  (lambda (u v S)
    (let ((u (walk u S)) (v (walk v S)))
      (cond
        ((and (pair? u) (pair? v))
         (let ((S (unify (car u) (car v) S)))
           (and S (unify (cdr u) (cdr v) S))))
        (else (unify-nonpair u v S))))))

(define unify-nonpair
  (lambda (u v S)
    (cond
      ((eq? u v) S)
      ((var? u) (ext-S u v S))
      ((var? v) (ext-S v u S))
      ((equal? u v) S)
      (else #f))))
```

```
(define ext-S
  (lambda (x v S)
    (case-value v
      ((y) (cons `(,x . ,y) S))
      ((au du) (cond
                 ((occurs√ x v S) #f)
                 (else (cons `(,x . ,v) S))))
      ((v) (cons `(,x . ,v) S)))))
```

```
(define occurs√
  (lambda (x v S)
    (case-value (walk v S)
      ((y) (eq? y x))
      ((av dv) (or (occurs√ x av S)
                   (occurs√ x dv S)))
      ((v) #f))))
```

D.4 The Reifier

The role of *reify* is to make the relevant information that is stored in the final state *final-c* (see **run**) as accessible as possible. Realizing that there might be a lot of relevant information about the variables in the final value of the variable created in **run** makes it essential that much care goes into writing the reifier. Specifically, we insist on a kind of Church-Rosser property (Barendregt 1984). Regardless of how a program is written, if it terminates it should be equal to every semantically equivalent program. For example, swapping conjuncts in a **fresh** should not change the appearance of the answers. But this equality must hold for D, A, and T which is why we sort lexicographically.

reify-S is the heart of the reifier. *reify-S* takes an arbitrary value v and a substitution S, and returns a substitution that maps every distinct variable in v to a unique symbol. The trick to maintaining left-to-right ordering of the subscripts on these symbols is to process v from left to right, as can be seen in the case of *reify-S* which handles pairs. When *reify-S* encounters a variable, it determines if we already have a mapping for that entity. If not, *reify-S* extends the substitution with an association between the variable and a new, appropriately-subscripted symbol built using *reify-name*.

```
(define walk*
  (lambda (v S)
    (case-value (walk v S)
      ((x) x)
      ((av dv)
       (cons (walk* av S) (walk* dv S)))
      ((v) v))))
```

```
(define reify-S
  (lambda (v S)
    (case-value (walk v S)
      ((x) (let ((n (length S)))
             (let ((name (reify-name n)))
               (cons `(,x . ,name) S))))
      ((av dv) (let ((S (reify-S av S)))
                 (reify-S dv S)))
      ((v) S))))
```

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

The remaining helpers form the final value, which includes relevant information in the constraints.

```
(define reify
  (lambda (x)
    (lambda (c)
      (let ((S (c→S c)) (D (c→D c))
            (A (c→A c)) (T (c→T c)))
        (let ((v (walk* x S)))
          (let ((S (reify-S v '())))
            (reify+ v S
              (let ((D (remp
                         (lambda (d) (anyvar? d S))
                         D)))
                (rem-subsumed D))
              (remp
                (lambda (a)
                  (var? (walk (lhs a) S)))
                A)
              (remp
                (lambda (t)
                  (var? (walk (lhs t) S)))
                T))))))))
```

In the definition of *reify+* below, we have removed the predicates from tag-predicate pairs of A and T, but we still use the names A and T to refer to these new data structures.

```
(define reify+
  (lambda (v S D A T)
    (let ((D (subsume A D)))
      (let ((A (map (lambda (a)
                      (let ((x (lhs a))
                            (tag (pr→tag a)))
                        `(,x . ,tag)))
                    A))
            (T (map (lambda (t)
                      (let ((x (lhs t))
                            (tag (pr→tag t)))
                        `(,x . ,tag)))
                    T)))
        (form (walk* v S)
              (walk* D S)
              (walk* A S)
              (rem-subsumed-T (walk* T S)))))))
```

```
(define form
  (lambda (v D A T)
    (let ((fd (drop-dot-D (sorter (map sorter D))))
          (f^a (sorter (map sort-part (partition* A))))
          (ft (drop-dot-T (sorter T))))
      (let ((fb (append ft f^a)))
        (cond
          ((and (null? fd) (null? fb)) v)
          ((null? fd) `(,v . ,fb))
          ((null? fb) `(,v . ((≢ . ,fd))))
          (else `(,v (≢ . ,fd) . ,fb)))))))
```

```
(define drop-dot-D
  (lambda (D)
    (map (lambda (d)
           (map (lambda (pr)
                  (let ((x (lhs pr))
                        (u (rhs pr)))
                    `(,x ,u)))
                d))
         D)))
```

```
(define drop-dot-T
  (lambda (T)
    (map (lambda (t)
           (let ((x (lhs t))
                 (tag (rhs t)))
             `(absent ,tag ,x)))
         T)))

(define sorter (lambda (ls) (sort lex≤? ls)))

(define sort-part
  (lambda (pr)
    (let ((tag (car pr))
          (x* (sorter (cdr pr))))
      `(,tag . ,x*))))

(define anyvar?
  (lambda (u S)
    (case-value u
      ((x) (var? (walk x S)))
      ((au du) (or (anyvar? au S)
                   (anyvar? du S)))
      ((u) #f))))

(define rem-subsumed
  (lambda (D)
    (let loop ((D D) (D+ '()))
      (cond
        ((null? D) D+)
        ((or (subsumed? (car D) (cdr D))
             (subsumed? (car D) D+))
         (loop (cdr D) D+))
        (else (loop (cdr D)
                    (cons (car D) D+)))))))

(define subsumed?
  (lambda (d D)
    (cond
      ((null? D) #f)
      (else (let ((d̂ (unify* (car D) d)))
              (or (and d̂ (eq? d̂ d))
                  (subsumed? d (cdr D))))))))

(define rem-subsumed-T
  (lambda (T)
    (let loop ((T T) (T^ '()))
      (cond
        ((null? T) T^)
        (else
         (let ((x (lhs (car T)))
               (tag (rhs (car T))))
           (cond
             ((or (subsumed-T? x tag (cdr T))
                  (subsumed-T? x tag T^))
              (loop (cdr T) T^))
             (else (loop (cdr T)
                         (cons (car T) T^))))))))))

(define subsumed-T?
  (lambda (x tag₁ T)
    (cond
      ((null? T) #f)
      (else
       (let ((y (lhs (car T)))
             (tag₂ (rhs (car T))))
         (or
           (and (eq? y x) (tag=? tag₂ tag₁))
           (subsumed-T? x tag₁ (cdr T))))))))

(define unify*
  (lambda (S+ S)
    (unify (map lhs S+) (map rhs S+) S)))

(define part
  (lambda (tag A x* y*)
    (cond
     ((null? A)
      (cons `(,tag . ,x*) (partition* y*)))
     ((tag=? (rhs (car A)) tag)
      (let ((x (lhs (car A))))
        (let ((x* (cond
                    ((memq x x*) x*)
                    (else (cons x x*)))))
          (part tag (cdr A) x* y*))))
     (else
      (let ((y* (cons (car A) y*)))
        (part tag (cdr A) x* y*))))))

(define partition*
  (lambda (A)
    (cond
      ((null? A) '())
      (else
       (part (rhs (car A)) A '() '())))))
```

The definition of *lex≤?* along with *datum→string* uses the effectful operator *display*, The functional version is tedious, because of the number of different built-in Scheme types, and we have opted to use this version instead.

```
(define lex≤?
  (lambda (x y)
    (string≤? (datum→string x) (datum→string y))))

(define datum→string
  (lambda (x)
    (call-with-string-output-port
      (lambda (p) (display x p)))))
```

D.5 Impure Control Operators

For completeness, we define three additional miniKanren goal constructors: **project**, which can be used to access the values of variables, and **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program. The examples from *Thin Ice* of *The Reasoned Schemer* (Friedman et al. 2005) demonstrate how **cond**^{a} and **cond**^{u} can be useful and the pitfalls that await the unsuspecting reader. Also, we have included an additional operator *once*^{o}, defined in terms of **cond**^{u}, which forces the input goal to succeed at most once.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g g* ...)
     (λG (c : S D A T)
       (let ((x (walk* x S)) ...)
         ((fresh () g g* ...) c))))))

(define-syntax condᵃ
  (syntax-rules ()
    ((_ (g₀ g ...) (g₁ ĝ ...) ...)
     (λG (c)
       (inc
         (ifᵃ ((g₀ c) g ...)
              ((g₁ c) ĝ ...) ...))))))
```

```
(define-syntax if$^a$
  (syntax-rules ()
    ((_) (mzero))
    ((_ (e g ...) b ...)
     (let loop ((c$^\infty$ e))
       (case$^\infty$ c$^\infty$
         (() (if$^a$ b ...))
         ((f) (inc (loop (f))))
         ((a) (bind$^*$ c$^\infty$ g ...))
         ((a f) (bind$^*$ c$^\infty$ g ...)))))))

(define-syntax cond$^u$
  (syntax-rules ()
    ((_ ($g_0$ g ...) ($g_1$ $\hat{g}$ ...) ...)
     ($\lambda_{\mathsf{G}}$ (c)
       (inc
         (if$^u$ (($g_0$ c) g ...)
               (($g_1$ c) $\hat{g}$ ...) ...))))))

(define-syntax if$^u$
  (syntax-rules ()
    ((_) (mzero))
    ((_ (e g ...) b ...)
     (let loop ((c$^\infty$ e))
       (case$^\infty$ c$^\infty$
         (() (if$^u$ b ...))
         ((f) (inc (loop (f))))
         ((c) (bind$^*$ c$^\infty$ g ...))
         ((c f) (bind$^*$ (unit c) g ...)))))))

(define once$^o$ (lambda (g) (cond$^u$ (g))))
```