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 written in the miniKanren relational (logic programming) language for several subsets of Scheme, demonstrate these interpreters running "backwards," 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 three extensions to core miniKanren (disequality constraints and symbol/number type-constraints) can be used to avoid tagging expressions in the languages being interpreted, simplifying the interpreters, eliminating the need for parsers/unparsers, and allowing shadowing of core forms.

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 2000; Thompson II); the simplest Scheme quines are self-evaluating literals, such as numbers and booleans. Here is a classic, and much more interesting, quine (Thompson II):

```
(define quine1
  '((lambda (x)
      (list x (list (quote quote) x)))
    (quote
     (lambda (x)
       (list x (list (quote quote) x))))))
```

We can easily verify that *quine1* evaluates to itself:

(*equal?* (*eval quine1*) *quine1*) ⇒ #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 typical environment-passing interpreter from Scheme into the miniKanren relational (logic programming) language (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 programs p and q that evaluate to each other (where p and q are not equal).

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, most importantly, because tagging the application line of the interpreter greatly complicates the handling of **quote** and *list*. Our approach to avoiding tagging also has the important benefit of properly handling shadowing of language forms, 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 $symbol^o$ and $number^o$ constraints we introduce are also straight-forward to implement in cKanren. However, we have found that core miniKanren augmented with these three constraints is sufficient for implementing a wide variety of interesting programs, including

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

(*equal?* (**run**^{1} (*q*) (*eval-exp*^{o} *q* '() *q*)) `(,*quine1*)) ⇒ #t

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 three constraint operators: the disequality constraint $\not\equiv$; and type constraints $symbol^o$ and $number^o$, which are similar in spirit to Scheme's *symbol?* and *number?* predicates (section 2.2). These operators 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). This technique can also be used for translating type inferencers from Scheme to miniKanren.
- We show how $\not\equiv$, $symbol^o$, and $number^o$ can be used when writing an interpreter (or type inferencer) to avoid tagging expressions in the language being interpreted (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 D) 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 4 and appendix A).
- We provide a complete, concise, and easily modifiable implementation of core miniKanren extended with $\not\equiv$, $symbol^o$, and $number^o$ constraints (appendix B).
- We provide a generalized version of the **pmatch** pattern matcher first presented in Byrd and Friedman (2007); the updated **pmatch** (appendix C), now called **dmatch**, is based on Dijkstra guards (Dijkstra 1975), and handles quote expressions, which are necessary for writing evaluators 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 $\not\equiv$, $symbol^o$, and $number^o$ constraint operators 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 constraint operators, 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 `==`), $\mathbf{cond}^e$ (entered as `conde`), and **fresh**. Similarly, ($\mathbf{run}^{\mathbf{5}}$ (q) *body*) and ($\mathbf{run}^*$ (q) *body*) are entered as `(run 5 (q) body)` and `(run* (q) body)`, respectively.[2]

miniKanren extends Scheme with three operators: $\equiv$, **fresh**, and $\mathbf{cond}^e$ (three new operators, $\not\equiv$, $symbol^o$, and $number^o$, 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.

fresh, which syntactically looks like **lambda**, introduces into scope new lexical variables bound to new (logic) variables; $\equiv$ unifies two terms. Thus

(**fresh** (x y z) ($\equiv$ x z) ($\equiv$ 3 y))

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

($\mathbf{run}^{\mathbf{1}}$ (q) (**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 variable q and thus can be any value. q also remains unbound in

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

Of course we can get back other values, representing bound variables.

```
(run¹ (y)          (run¹ (q)          (run¹ (y)
  (fresh (x z)       (fresh (x z)       (fresh (x y)
    (≡ x z)            (≡ x z)            (≡ 4 x)
    (≡ 3 y)))          (≡ 3 z)            (≡ x y))
                       (≡ q x)))        (≡ 3 y))
```

Each of these examples returns (3); in the rightmost example, the y introduced by **fresh** is different from the y introduced by **run**. A **run** expression can also evaluate to the empty list. This indicates that there does not exist any value of the variable bound by the **run** expression that can cause its body to succeed.

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

We use $\mathbf{cond}^e$ to get several values. Syntactically, $\mathbf{cond}^e$ looks like **cond** but without $\Rightarrow$ or **else**. For example,

```
(run² (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 $\mathbf{cond}^e$ 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 value, which is why the reified value of x is $_{-0}$ in the first value but $_{-1}$ in the second value. The super-

[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 $\mathbf{cond}^e$ stands for 'every,' since every $\mathbf{cond}^e$ clause may contribute answers.

script 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))))) ⇒ ⊥
```

Had $*$ been replaced by a natural number n, then an n-element list of alternating #f's and #t's would be returned. The **cond**^{e} succeeds while associating q with #f, which accounts for the first value. When getting the second value, the second **cond**^{e} line is tried, and the association made between q and #f is forgotten—we say that q has been *refreshed*. In the third **cond**^{e} line, 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 in which values are returned.)

Now consider

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

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

Even if some **cond**^{e} lines loop indefinitely, other **cond**^{e} lines can contribute to the values returned by a **run** expression. However, we are not concerned with expressions looping indefinitely. For example,

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

returns (1 2 3); replacing **run**^{3} with **run**^{4} would cause divergence since there are only three values and *never*^{o} would loop indefinitely looking for the fourth.

2.2 Additional Constraint Operators

We extend core miniKanren with three constraint operators: the disequality constraint $\not\equiv$ (previously described in the context of the cKanren constraint logic programming framework (Alvis et al. 2011)); and type constraints *symbol*^{o} and *number*^{o}, which can be thought of as the miniKanren equivalent of Scheme's *symbol?* and *number?* type predicates.

Consider the **run** expression

```
(run^1 (q) (symbol^o q))
```

which evaluates to $((_{.0}\ (\mathsf{sym}\ _{.0})))$. This answer indicates that q remains unbound, but also that q can only be associated with a symbol, as demonstrated by the failure of these programs:

```
(run^2 (q)
  (symbol^o q)
  (≡ 4 q)) ⇒ ()
```

and

```
(run^3 (q)
  (symbol^o q)
  (number^o q)) ⇒ ()
```

If we were to replace *symbol*^{o} by *number*^{o} in the **run**^{1} and **run**^{2} examples, the first answer would contain the num tag rather than the sym tag, while the second example would produce the answer (4) rather than failing.

Now consider an example illustrating the disequality constraint $\not\equiv$.

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

Next, consider a slightly more complicated example: a disequality constraint between lists.

```
(run^1 (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 if p is associated with 1, r cannot be associated with 2 (and that if r is associated with 2, p cannot be associated with 1).

We would get the same behavior if we were to replace

```
(≢ '(1 2) `(,p ,r))
```

by

```
(≢ '((1) (2)) `((,p) (,r)))
```

or even

```
(≢ `((1) (,r)) `((,p) (2))).
```

Now consider the **run**^{1} expression

```
(run^1 (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**^{1} expression fails.

```
(run¹ (q)
  (fresh (p r)
    (≠ '(1 2) `(,p ,r))
    (≡ 1 p)
    (≡ 2 r)
    (≡ `(,p ,r) q))) ⇒ ()
```

Finally, we consider what happens when $(\equiv 2\ r)$ is replaced by $(symbol^o\ r)$ in the previous example. Then the $\mathbf{run^1}$ expression succeeds with the answer $(((1\ _0)\ (\mathsf{sym}\ _0)))$ which states that r can only be associated with a symbol. The reified constraint $(\not\equiv ((_0\ .\ 2)))$ (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.

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
      (() #f)
      (((,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 (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)))))
```

With **dmatch** we have the liberty of ordering our clauses in any way we want. The guard for the λ clause,

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

ensures that *eval-exp* will correctly evaluate programs in which the keyword **lambda** is shadowed (that is, programs containing a variable named *lambda*, which results in the symbol lambda being bound in the environment).

Here are two examples showing *eval-exp* in action (the empty list passed as the second argument represents the empty environment):

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

and

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

No set of examples of the untyped λ-calculus would be complete without self application, which demonstrates shadowing of the **lambda** keyword.

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

(eval-exp Ω '()) ⇒ ⊥
```

We now have a working λ-calculus interpreter in Scheme; our next task is to translate this version 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 substitution *env*.

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

If x is not bound in *env*, a call to $lookup^o$ will reach the base case and fail[4], rather than returning #f as in *lookup*.

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

Each $\mathbf{cond}^e$ clause in $lookup^o$ corresponds to a **dmatch** clause in *loookup*. Instead of pattern matching against *env*, $lookup^o$ uses unification, where *env* is one of the arguments to $\equiv$. The goal $(\not\equiv\ y\ x)$ is equivalent to the guard

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

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

(*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 lifting the unification of *env*.

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

With *lookup*^{o} defined, we can now 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 guard into a goal expression. This we do below, translating the guard (*not-in-env* 'lambda *env*) into the equivalent call (*not-in-env*^{o} 'lambda *env*).

```
(define eval-exp^o
  (lambda (exp env val)
    (cond^e
      ((fresh (rator rand x body env2 a)
         (≡ `(,rator ,rand) exp)
         (eval-exp^o rator env `(closure ,x ,body ,env2))
         (eval-exp^o rand env a)
         (eval-exp^o body `((,x . ,a) . ,env2) val)))
      ((fresh (x body)
         (≡ `(lambda (,x) ,body) exp)
         (≡ `(closure ,x ,body ,env) val)
         (not-in-env^o 'lambda env)))
      ((symbol^o exp) (lookup^o exp env val)))))

(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))))))
```

Here are the first five programs whose values are α-equivalent to the closure from the last terminating example:

```
(run^5 (q)
  (eval-exp^o 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)))
```

And, of course, we can generate expression/value pairs.

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

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

This **run**^{5} expression generates five λ-expressions, and the closures to which they evaluate. The $\not\equiv$ tags in the answers indicate disequality constraints between variables and the values they cannot assume. The last answer states that $_{-1}$ cannot be the symbol lambda and that $_{-0}$ must be a symbol.

```
(((lambda (_0) _1) ⇒ (closure _0 _1 ()))
 ((((lambda (_0) _0) (lambda (_1) _2))
   ⇒
   (closure _1 _2 ()))
  (sym _0))
 ((((lambda (_0) (lambda (_1) _2)) (lambda (_3) _4))
   ⇒
   (closure _1 _2 ((_0 closure _3 _4 ()))))
  (≢ ((_0 . lambda))))
 ((((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)))
```

4. Generating Quines

We have an interpreter that is capable of running backwards, but we cannot yet generate quines. In fact, our interpreter cannot evaluate *quine1* from section 1, even when running forward. We must first add support for **quote** and *list*.

4.1 Extending the Interpreter

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)
```

However, in order to handle shadowing correctly, we must allow the user to override the **quote** form. As with the λ clause, we handle this 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, the following two programs are equivalent:

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

and

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

The problem is that (quote (closure x x ())) and (lambda (x) x) both evaluate to (closure x x ()). We circumvent this issue by declaring that the closure tag is a unique symbol that cannot be typed by the user (in effect, a gensym).

We can add **quote** to the miniKanren interpreter as follows.

```
((fresh (v)
   (≡ `(quote ,v) exp)
```

```
      (not-in-envº 'quote env)
      (not-closureoº v)))
```

We have now explicitly forbidden the symbol closure from appearing in the quoted datum. In the Scheme version, we could assume that the user could not type the unique closure tag, but here we are interested in running our interpreter backwards; miniKanren has no compunction against generating expressions that include a symbol the user cannot type.

With **quote** added to our interpreters, we 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)
   (not-closureoº a*)
   (proper-listº a* env val)))
```

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

As with the other tagged clauses, we handle shadowing through a call to *not-in-envº*. The disequality constraints that allow the interpreter to handle shadowing of list and quote correctly also serve another purpose. Without these constraints, 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-listº* to ensure a^* is a proper list:

```
(define proper-listº
  (lambda (exp env val)
    (conde
      ((≡ '() exp)
       (≡ '() val))
      ((fresh (a d t-a t-d)
         (≡ `(,a . ,d) exp)
         (≡ `(,t-a . ,t-d) val)
         (eval-expº a env t-a)
         (proper-listº d env t-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)) (cdr (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 (not-in-env 'lambda env))
       `(closure ,x ,body ,env)))))
```

where *rator?* is defined as

```
(define rator?
  (lambda (x env)
    (dmatch x
      ((,a . ,d) #t)
      (,x (guard (symbol? x))
          (valid-name? x env)))))

(define valid-name?
  (lambda (f-name env)
    (dmatch env
      (((,x . ,v) . ,rest)
       (cond
         ((eq? x f-name))
         ((not (eq? x f-name))
          (valid-name? f-name rest))))
      (() (not-op-name? f-name op-names)))))

(define not-op-name?
  (lambda (x op-names)
    (dmatch op-names
      ((,y . ,rest)
       (cond
         ((eq? y x) #f)
         ((not (eq? y x)) (not-op-name? x rest))))
      (() #t))))

(define op-names '(lambda quote list))
```

The predicate, *rator?* special cases when the rator is a symbol. In the case that the symbol is in the environment, we know we have a valid name, even if it is an operand name. But, if it is not in the environment, but is an operator name, then the name is invalid. This means that

```
(rator? 'lambda)
```

returns #f unless the symbol lambda is in scope, as it is in the Ω example above. Why did we not need to use *rator?* in the definition of *eval-exp* in section 3? Each of the three expressions have defferent 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, which makes it acceptable to **dmatch**.

In the definition of *eval-expº*, below, we can drop the test to determine if a rator is valid, since we know that each name that might be considered will fail if *not-in-envº* fails. Thus if the expression is (quote (lambda (x) x)), and the variable *quote* is in scope, then *eval-expº* will treat the expression as a procedure application, rather than a use of Scheme's **quote** form.

```
(define eval-expº
  (lambda (exp env val)
    (conde
      ((fresh (v)
         (≡ `(quote ,v) exp)
         (not-in-envº 'quote env)
         (not-closureoº v)
         (≡ v val)))
      ((fresh (a*)
         (≡ `(list . ,a*) exp)
         (not-in-envº 'list env)
         (not-closureoº a*)
         (proper-listº a* env val)))
      ((symbolº exp) (lookupº exp env val))
      ((fresh (rator rand x body envˆ 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)
    (not-in-env^o 'lambda env)
    (≡ `(closure ,x ,body ,env) val))))))
```

4.2 Quines

After much work, we are finally ready to put *eval-exp*^{o} to work, 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.

```
(run^1 (q) (eval-exp^o q '() q))
⇒
((((lambda (_0) (list _0 (list 'quote _0)))
   '(lambda (_0) (list _0 (list 'quote _0))))
  (≠ ((_0 . list)) ((_0 . quote)))
  (not-closure _0)
  (sym _0)))
```

Sure enough, this is our old friend, *quine1*.

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 additional restriction that p and q are not equal.

```
(run^1 (q)
  (fresh (x y)
    (≠ x y)
    (eval-exp^o x '() y)
    (eval-exp^o y '() x)
    (≡ `(,x ,y) q)))
⇒
(((`((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 . list)) ((_0 . quote)))
 (not-closure _0)
 (sym _0)))
```

5. Conclusion

Quines have a long and interesting history: the term "quine" was coined by Douglas Hofstadter (Hofstadter 2000) in honor of the logician Willard van Orman Quine, but the concept goes back to Kleene's recursion theorems (Rogers 1987).

In section 3 we describe how disequality constraints 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**, which 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 was responsible for suggesting to us that if we indeed could run an interpreter backwards (and he witnessed it), we ought to be able to generate quines. It was this observation following our talk at Clojure Conj that was the start of our thinking about this specific problem. We thank you, Stuart, for this challenge.

Most of our group of programming languages researchers (PL-WONKS) joined us in an advanced PL course the following semester where one of the topics 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 class to build backward-running interprters. She, too, deserves much credit. We thank Jason Hemann, Aaron Hsu, and Cameron Swords for their involvement in developing software (some of which found its way into this paper) and their generally helpful comments. 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 Chien offered to revise our implementation of miniKanren, which first appeared in the first edition of *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, and still, we appreciate and thank Dorai Sitaram's SLaTeX. Working on a long-lived project, it is always difficult to recall all those that we want to thank, but our other papers and the book will hopefully not have 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*. Penguin, 20th anniversary edition, March 2000. ISBN 0140289208.

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.

Hartley Rogers, Jr. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, USA, 1987. ISBN 0-262-68052-1.

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 full interpreter for a uncurried Scheme with arithmetic operators, conditionals, and pairs that allows us to still generate quines, but perhaps a little slower.

Instead of using Scheme numbers, *eval-exp*^{o} uses the relational arithmetic (and relational number) system described in appendix D, (and first presented in Kiselyov et al. (2008) and Friedman et al. (2005)).[6]

```
(define eval-exp^o
  (lambda (exp env val)
    (cond^e
      ((fresh (v)
         (≡ `(quote ,v) exp)
         (not-in-env^o 'quote env)
         (not-closureo^o v)
         (not-into v)
         (≡ v val)))
      ((fresh (a*)
         (≡ `(list . ,a*) exp)
         (not-in-env^o 'list env)
         (not-closureo^o a*)
         (not-into 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^ a* res)
         (≡ `(,rator . ,rands) exp)
         (eval-exp^o rator env `(closure ,x* ,body ,env^))
         (proper-list^o rands env a*)
         (ext-env*^o x* a* env^ 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* env2)
         (≡ `(,x . ,dx*) x*)
         (≡ `(,a . ,da*) a*)
         (≡ `((,x . ,a) . ,env) env2)
         (ext-env*^o dx* da* env2 out))))))
```

The primitive notions are booleans, numbers, pairs, and operations over them. The goal *prim-exp*^{o} processes these notions independently. but the primitive operations, *cons*, *car*, *cdr*, *not*, *sub1*, *zero?*, and $*$ 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-primo 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))
```

[6] Since the relational arithmetic system uses a special representation of numbers, we allow the use of quoted arabic numerals, but they cannot be used in arithmetic.

```
    ((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 cons-prim^o
  (lambda (exp env val)
    (fresh (a d v-a v-d)
      (≡ `(cons ,a ,d) exp)
      (≡ `(,v-a . ,v-d) val)
      (not-closureo^o val)
      (not-into 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 a d)
      (≡ `(car ,p) exp)
      (≡ a val)
      (≠ 'into val)
      (not-in-env^o 'car env)
      (eval-exp^o p env `(,a . ,d)))))

(define cdr-prim^o
  (lambda (exp env val)
    (fresh (p a d)
      (≡ `(cdr ,p) exp)
      (≡ d val)
      (≠ 'into val)
      (not-in-env^o 'cdr env)
      (eval-exp^o p env `(,a . ,d)))))

(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 number-primo
  (lambda (exp env val)
    (fresh (n)
      (≡ `(numo ,n) exp)
      (≡ `(into ,n) val)
      (not-in-env^o 'numo env))))

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

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

(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))))))
```

Next we consider several examples using *eval-exp*^{o}. Consider this **run** expression, which returns the first 8 expressions whose values are six.

```
(run 8 (q) (eval-exp^o q '() `(into ,(build-num 6)))) ⇒
((numo (0 1 1))
 (sub1 (numo (1 1 1)))
 ((lambda () (numo (0 1 1))))
 (((lambda (_.0) (numo (0 1 1))) '_.1)
  (≠ ((_.0 . numo)))
  (not-closure _.1)
  (not-into _.1))
 (* (numo (1)) (numo (0 1 1)))
 (* (numo (0 1 1)) (numo (1)))
 (* (numo (0 1)) (numo (1 1)))
 (((lambda (_.0) (numo (0 1 1))) (list))
  (≠ ((_.0 . numo)))))
```

The 7th value in this list is
(* (numo (0 1)) (numo (1 1)))

And, if we look at the first 500 answers,

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

we discover
(sub1 (sub1 (sub1 (numo (1 0 0 1)))))
is the 240th value.

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

```
(define rel-fact5
  `((lambda (f)
      ((f f) ,(build-num 5)))
    (lambda (f)
      (lambda (n)
        (if (zero? n)
            ,(build-num 1)
            (* n ((f f) (sub1 n))))))))
(run* (q) (eval-exp^o rel-fact5 '() q))
⇒ ((numo (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^10 (q) (eval-exp^o q '() q))
⇒
(#t
 #f
 (((lambda (_0) (list _0 (list 'quote _0)))
   '(lambda (_0) (list _0 (list 'quote _0))))
  (≠ ((_0 . list)) ((_0 . quote)))
  (not-closure _0)
  (not-into _0)
  (sym _0))
 (((lambda (_0) (list _0 (list (car '(quote . _1)) _0)))
   '(lambda (_0) (list _0 (list (car '(quote . _1)) _0))))
  (≠ ((_0 . car)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0 _1)
  (not-into _0 _1)
  (sym _0))
 (((lambda (_0)
     (list (list 'lambda '(_0) _0) (list 'quote _0)))
   '(list (list 'lambda '(_0) _0) (list 'quote _0)))
  (≠ ((_0 . list)) ((_0 . quote)))
  (not-closure _0)
  (not-into _0)
  (sym _0))
 (((lambda (_0) (list (car _0) (list 'quote _0)))
   '((lambda (_0) (list (car _0) (list 'quote _0))) . _1))
  (≠ ((_0 . car)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0 _1)
  (not-into _0 _1)
  (sym _0))
 (((lambda (_0) (list _0 (list (cdr '(_1 . quote)) _0)))
   '(lambda (_0) (list _0 (list (cdr '(_1 . quote)) _0))))
  (≠ ((_0 . cdr)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0 _1)
  (not-into _0 _1)
  (sym _0))
 (((lambda (_0) (cons _0 (list (list 'quote _0))))
   '(lambda (_0) (cons _0 (list (list 'quote _0)))))
  (≠ ((_0 . cons)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0)
  (not-into _0)
  (sym _0))
 (((lambda (_0) ((lambda () (list _0 (list 'quote _0)))))
   '(lambda (_0) ((lambda () (list _0 (list 'quote _0))))))
  (≠ ((_0 . lambda)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0)
  (not-into _0)
  (sym _0))
 (((lambda (_0)
     (list
       (list 'lambda '(_0) _0)
       (list (car '(quote . _1)) _0)))
   '(list
      (list 'lambda '(_0) _0)
      (list (car '(quote . _1)) _0)))
  (≠ ((_0 . car)) ((_0 . list)) ((_0 . quote)))
  (not-closure _0 _1)
  (not-into _0 _1)
  (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 generated answers.

B. 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}, **cond**^{e}, and **fresh**, which take a *package* implicitly.

A package is a list of three values, each of which is, or contains, an association list of variables to values. The first value in a package is a substitution, s, which associates values with variables. The second value in a package is a list of association lists, c^*; each association list, c, represents a *disequality constraint*. The last value in a package is an association list, t, that associates variables with symbols. 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.

```
(define a→s (lambda (a) (car a)))
(define a→c* (lambda (a) (cadr a)))
(define a→t (lambda (a) (caddr a)))
```

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

```
(define-syntax λG
  (syntax-rules (:)
    ((_ (a) e) (lambda (a) e))
    ((_ (a : s c* t) e)
     (lambda (a)
       (let ((s (a→s a)) (c* (a→c* a)) (t (a→t a)))
         e)))))
```

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

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

To represent a stream containing multiple packages, we use (*choice a f*), where a 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 an a^∞—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 a is the same as (*choice a 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 a^∞'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) ((â) e2) ((a f) e3))
 (let ((a∞ e))
   (cond
     ((not a∞) e0)
     ((procedure? a∞) (let ((f̂ a∞)) e1))
     ((not (and (pair? a∞)
                (procedure? (cdr a∞))))
      (let ((â a∞)) e2))
     (else (let ((a (car a∞)) (f (cdr a∞)))
             e3)))))))
```

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))
         ((a) (cons a '()))
         ((a f) (cons a (take (and n (− n 1)) f))))))))
```

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 a produced by running all the goals in the empty package *empty-a*.

```
(define empty-a '(() () ()))

(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (λf ()
         ((fresh (x) g0 g ...
            (λg (final-a)
              (choice ((reify x) final-a) empty-f)))
          empty-a))))))

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

B.1 Goal Constructors

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

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λg (a)
       (inc
         (let ((x (var 'x)) ...)
           (bind* (g0 a) 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 a^∞. The resulting a^∞'s are then merged using *mplus*, which combines an a^∞ and an f to yield a single a^∞.

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

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

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

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 a for each **cond**^{e} line. To avoid unwanted divergence, we treat the **cond**^{e} lines as a single **inc** stream.

```
(define-syntax cond^e
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λg (a)
       (inc
         (mplus*
           (bind* (g0 a) g ...)
           (bind* (g1 a) ĝ ...) ...))))))

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

The function *make-tag* is the helper that is called from *symbol*^{o} and *number*^{o} and contains the essence of what those two operators accomplish. Elements of t act as daemons, making certain that associations are added to the substitution that violate the constraints in t. In addition, c^* may contain a disequality constraint bwteen, 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
  (lambda (tag pred)
    (letrec ((rec
      (lambda (u)
        (λg (a : s c* t)
          (let ((u (if (var? u) (walk u s) u)))
            (cond
              ((pair? u)
               (cond
                 ((pred u)
                  ((fresh ()
                     (rec (car u))
                     (rec (cdr u)))
                   a))
                 (else (mzero))))
              ((not (var? u))
               (cond
                 ((pred u) (unit a))
                 (else (mzero))))
```

```
                ((ext-t u tag t s) ⇒
                 (lambda (t0)
                   (cond
                     ((not (eq? t0 t))
                      (let ((t̂ (list (car t0))))
                        (let ((c* (subsume t̂ c*)))
                          (unit `(,s ,c* ,t0)))))
                     (else (unit a)))))
                (else (mzero))))))))
      rec)))

(define tag→pred
  (lambda (tag)
    (case tag
      ((sym) symbol?)
      ((num) number?)
      ((not-closure) not-closure?)
      ((not-into) not-into?)
      (else (error 'tag→pred "unknown tag ~s" tag)))))

(define ext-t
  (lambda (x tag t̂ s)
    (let ((x (walk x s)))
      (let loop ((t t̂))
        (cond
          ((null? t) (cons (cons x tag) t̂))
          ((not (eq? (walk (lhs (car t)) s) tag)) (loop (cdr t)))
          ((eq? (rhs (car t)) tag) t̂)
          ((works-together? (rhs (car t)) tag)
           (loop (cdr t)))
          (else #f))))))

(define subsume
  (lambda (t c*)
    (remp
      (lambda (c) (exists (subsumed-pr? t) c))
      c*)))

(define subsumed-pr?
  (lambda (t)
    (lambda (pr-c)
      (let ((u (rhs pr-c)))
        (and (not (var? u))
             (cond
               ((assq (lhs pr-c) t) ⇒
                (lambda (pr-t)
                  (not ((tag→pred (rhs pr-t)) u))))
               (else #f)))))))
```

Here is the implementation of the remaining interface operators.

```
(define bool^o
  (lambda (x)
    (cond^e
      ((≡ #f x))
      ((≡ #t x)))))

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

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

(define not-closure?
  (lambda (x)
    (not (eq? x 'closure))))

(define not-closureo^o
  (make-tag 'not-closure not-closure?))

(define not-into?
  (lambda (x)
    (not (eq? x 'into))))

(define not-into
  (make-tag 'not-into not-into?))

(define works-together?
  (lambda (t1 t2)
    (cond
      ((and (eq? t1 'sym) (eq? t2 'num)) #f)
      ((and (eq? t1 'num) (eq? t2 'sym)) #f)
      (else #t))))
```

The definitions of $\not\equiv$ and $\equiv$ both use *unify* (section B.3). But, when we succeed by invoking *unit*, we pass a different substitution. In the $\equiv$ case, we pass the (possibly) extended substitution, however, in the $\not\equiv$ case, we pass the original substitution. So what happens in the $\not\equiv$ case? The actual extension, here called the *prefix*, is a constraint. We can take the constraint and make sure that t is okay with each association in the prefix. Those associations that are not dropped from the prefix and the surviving associations are added as a new constraint to c^*. (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_0, assuming that s_0 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)
    (λg (a : s c* t)
      (cond
        ((unify u v s) ⇒
         (lambda (s0)
           (cond
             ((eq? s0 s) (mzero))
             (else
              (let ((p* (list (prefix-s s0 s))))
                (let ((p* (subsume t p*)))
                  (let ((c* (append p* c*)))
                    (unit `(,s ,c* ,t)))))))))
        (else (unit a))))))

(define prefix-s
  (lambda (s0 s)
    (cond
      ((eq? s0 s) '())
      (else (cons (car s0)
                  (prefix-s (cdr s0) s))))))
```

Just as $\not\equiv$ checked t before extendeding c^*, $\equiv$ must check c^* and t (both of which might change) before it can succeed.

```
(define ≡
  (lambda (u v)
    (λg (a : s c* t)
      (cond
        ((unify u v s) ⇒
         (lambda (s0)
           (cond
             ((eq? s0 s) (unit a))
             ((verify-c* c* s0) ⇒
              (lambda (c*)
                (cond
                  ((verify-t t s0) ⇒
                   (lambda (t)
                     (let ((c* (subsume t c*)))
                       (unit `(,s0 ,c* ,t)))))
                  (else (mzero))))))))))))
```

```
            (else (mzero)))))
        (else (mzero))))))

(define verify-c*
  (lambda (c* s)
    (cond
      ((null? c*) '())
      ((verify-c* (cdr c*) s) ⇒
       (let ((s0 (unify* (car c*) s)))
         (lambda (c*)
           (cond
             (s0 (and (not (eq? s0 s))
                      (cons (prefix-s s0 s) c*)))
             (else c*)))))
      (else #f))))

(define verify-t
  (lambda (t s)
    (cond
      ((null? t) '())
      ((verify-t (cdr t) s) ⇒
       (let* ((tag (rhs (car t)))
              (pred (tag→pred tag)))
         (letrec ((rec
                   (lambda (u)
                     (let ((u (if (var? u) (walk u s) u)))
                       (lambda (t)
                         (cond
                           ((var? u) (ext-t u tag t s))
                           ((pair? u)
                            (cond
                              (((rec (car u)) t) ⇒
                               (lambda (t)
                                 ((rec (cdr u)) t)))
                              (else #f)))
                           (else (and (pred u) t))))))))
           (rec (lhs (car t))))))
      (else #f))))

(define #s (≡ #f #f))

(define fail (≡ #f #t))
```

B.2 miniKanren Helpers

```
(define var (lambda (dummy) (vector dummy)))
(define var? (lambda (x) (vector? x)))
(define lhs (lambda (pr) (car pr)))
(define rhs (lambda (pr) (cdr pr)))
```

This definition of *walk* assumes that its first argument is a variable.

```
(define walk
  (lambda (x s)
    (let ((a (assq x s)))
      (cond
        (a (let ((u (rhs a)))
             (if (var? u) (walk u s) u)))
        (else x)))))

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

B.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 *valid?* calls a recursive function, *occurs*$^{\checkmark}$.

```
(define unify
  (lambda (u v s)
    (let ((u (if (var? u) (walk u s) u))
          (v (if (var? v) (walk v s) v)))
      (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)
       (and (or (not (pair? v)) (valid? u v s))
            (cons `(,u . ,v) s)))
      ((var? v)
       (and (or (not (pair? u)) (valid? v u s))
            (cons `(,v . ,u) s)))
      ((equal? u v) s)
      (else #f))))

(define valid?
  (lambda (x v s)
    (not (occurs✓ x v s))))

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

B.4 The Reifier

The role of *reify* is to make the relevant information that is stored in the final state *final-a* (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 a semantically equivalent program. For example, swapping conjuncts in a **fresh** should not change the appearance of the answers. But this equality must hold for c^* and t, which is why we sort lexicographically.

reify-s is the heart of the reifier. *reify-s* takes an arbitrary value v, 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 *pair?*

cond clause, below. 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 reify-s
  (lambda (v)
    (let reify-s ((v v) (r '()))
      (let ((v (if (var? v) (walk v r) v)))
        (cond
          ((var? v)
           (let ((n (length r)))
             (let ((name (reify-name n)))
               (cons `(,v . ,name) r))))
          ((pair? v)
           (let ((r (reify-s (car v) r)))
             (reify-s (cdr v) r)))
          (else r))))))

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

(define reify
  (lambda (x)
    (λg (a : s c* t)
      (let ((v (walk* x s)))
        (let ((r (reify-s v)))
          (reify-aux r v
            (let ((c* (remp
                        (lambda (c)
                          (anyvar? c r))
                        c*)))
              (rem-subsumed c*))
            (remp
              (lambda (pr)
                (var? (walk (lhs pr) r)))
              t)))))))

(define reify-aux
  (lambda (r v c* t)
    (let ((v (walk* v r))
          (c* (walk* c* r))
          (t (walk* t r)))
      (let ((c* (sorter (map sorter c*)))
            (p* (sorter
                  (map sort-t-vars
                    (partition* t)))))
        (cond
          ((and (null? c*) (null? p*)) v)
          ((null? c*) `(,v . ,p*))
          (else `(,v (≢ . ,c*) . ,p*))))))))

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

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

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)))))

(define anyvar?
  (lambda (c r)
    (cond
      ((pair? c)
       (or (anyvar? (car c) r)
           (anyvar? (cdr c) r)))
      (else (var? (walk c r))))))

(define rem-subsumed
  (lambda (c*)
    (let rem-subsumed ((c* c*) (c^* '()))
      (cond
        ((null? c*) c^*)
        ((or (subsumed? (car c*) (cdr c*))
             (subsumed? (car c*) c^*))
         (rem-subsumed (cdr c*) c^*))
        (else (rem-subsumed (cdr c*)
                (cons (car c*) c^*)))))))

(define subsumed?
  (lambda (c c*)
    (cond
      ((null? c*) #f)
      (else
        (let ((ĉ (unify* (car c*) c)))
          (or
            (and ĉ (eq? ĉ c))
            (subsumed? c (cdr c*))))))))

(define unify*
  (lambda (c s)
    (unify (map lhs c) (map rhs c) s)))

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

(define partition*
  (lambda (t)
    (cond
      ((null? t) '())
      (else
       (part (rhs (car t)) t '() '())))))
```

B.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 $\mathbf{cond}^a$ and $\mathbf{cond}^u$, 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 (a : s c* t)
       (let ((x (walk* x s)) ...)
         ((fresh () g g* ...) a))))))

(define-syntax cond^a
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (a)
       (inc
         (if^a ((g0 a) g ...)
               ((g1 a) ĝ ...) ...))))))

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

(define-syntax cond^u
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (a)
       (inc
         (if^u ((g0 a) g ...)
               ((g1 a) ĝ ...) ...))))))

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

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

C. Generalized Pattern Matcher

This appendix gives a definition of **dmatch** that is more general in several ways than Oleg Kiselyov's **pmatch**, which appeared in Byrd and Friedman (2007). It improves error reporting, since now it is possible to associate a name with each appearance of **dmatch**, as in the use of *example* in *h* below. We get more generality by not handling **quote** specially, which allows for certain common patterns to be specified that were previously not possible. Finally, there is no **else** clause and the order of the clauses is arbitrary, but only one pattern (plus guard) can succeed for each invocation of **dmatch**. Here is an example of **dmatch** using **guard**s.

```
(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 1 2) (apply h `(1 (3 4)))) ⇒ (3 8)
```

In this example, a dotted pair is matched against two different kinds of patterns. In the first pattern, 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 guard (no side-effects allowed in guards) is run within the scope of a and b. The guard succeeds only if x and y are numbers; if so, then the sum of x and y is returned.

The second pattern matches against a pair (a three-element list), provided that the optional guard succeeds. The value of x is `1` and the value of y is `(3 4)`. Then a matches against `1`, b matches against `3`, and c matches against `4`. They are all numbers, so both calls to h succeed.

The overall syntax of **dmatch** looks like this:

$$
\begin{aligned}
\mathit{match} &:= (\textbf{dmatch}\ \mathit{exp}\ \mathit{clause}\ \ldots) \\
&\quad\ \mid (\textbf{dmatch}\ \mathit{exp}\ \texttt{name}\ \mathit{clause}\ \ldots) \\
\mathit{clause} &:= (\mathit{pattern}\ \mathit{guard}\ \mathit{exp}\ \ldots) \\
\mathit{guard} &:= (\textbf{guard}\ \mathit{boolean\text{-}exp}\ \ldots) \mid \epsilon \\
\mathit{pattern} &:= ,\texttt{var} \\
&\quad\ \mid \mathit{exp} \\
&\quad\ \mid (\mathit{pattern}_1\ \mathit{pattern}_2\ \ldots) \\
&\quad\ \mid (\mathit{pattern}_1\ .\ \mathit{pattern}_2)
\end{aligned}
$$

Now we examine the implementation of **dmatch**. The main **dmatch** macro simply handles the optional name that we can provide, and passes off control to the auxiliary helpers which do most of the extra work. Our auxiliary macros will give us a package list 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*)))))
```

In our case we want to represent a package comprising the clause and a thunk. We use the following for our package abstraction.

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

The first step in processing a **dmatch** expression is to ensure that we only evaluate the input expression once, which is what the **dmatch-remexp** ensures.

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

```
      (dmatch-aux v cls ...)))
    ((_ v cls ...) (dmatch-aux v cls ...))))
```

At each expansion of **dmatch-aux**, we want to create a package list of some type. We have three cases: two recursive cases and a single base case. If we have a pattern without a guard and the pattern matches, we want to add its clause along with its thunk to the package list. In the case where we have a guard, we want to conditionally add the clause and thunk 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))))))
```

To do the heavy lifting, we abstract the actual pattern matching into another helper macro **ppat** that does the check on the pattern and 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. To encode the alternative, we build a thunk, which avoids expanding the same expression multiple times.

Now we consider how matching occurs using **ppat**, and leverage the **syntax-rules** pattern matcher to do most of the work. We need to do a bit of tree recursion on our expansion in the pair case to match the *car* and *cdr* cases. Since we may have vectors or other data we want to handle, we use *equal?* instead of *eq?*.

```
(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 using *no-matching-pattern*. If there is an overlap between two or more patterns/guards, then we report this error using *overlapping-patterns/guards*. Otherwise, if there is no overlap, then we invoke the thunk in the singleton package list.

```
(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 (without 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 variable a and b could have replaced vx and vy, respectively. Second, a and b should be parallel **let** bindings.

D. 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), Byrd (2009), and Friedman et al. (2005)—please see these references for a detailed description of the code, and Kiselyov et al. (2008) for termination proofs for the individual operators.

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

D.1 Relational Arithmetic

Relational arithmetic allows for answers to questions such as 'what are five triples of positive integers x, y, and z for which $x + y = z$?', and 'for which natural numbers x and y does $x*y = 24$ hold?', which can be expressed in miniKanren as

```
(run⁵ (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, with the restriction that the most significant bit cannot be 0; this restriction is to ensure each number has a unique representation. Zero is therefore represented by the empty list rather than (0), since that would violate the never-terminated-by-0 constraint. The number one is represented by (1), the number two by (0 1), etc. But, we are doing relational programming, so numbers need not be ground; however, there is still the constraint that no number ends with a 0.

For example, '(1 . ,x) represents any odd natural number, while '(0 . ,x) represents any *positive* even number (with the condition that x must be positive, which we shall assume in the rest of this description). There are opportunities to replace bits by variables, so (0 0 0 1) represents the number 8, but '(0 0 0 . ,x) represents multiples of 8. So, if x is (1), the multiple is just 8. If x is (0 1), the multiple of 8 is 16, and so forth. We can even have '(0 ,y 0 . ,x), which 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 **run** expressions above:

```
((_0 () _0)
 (() (_0 . _1) (_0 . _1))
 ((1) (1) (0 1))
 ((1) (0 _0 . _1) (1 _0 . _1))
 ((1) (1 1) (0 0 1)))
```

and

```
(((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)))
```

It is relatively simple to interpret the answers to the second example, but in the first example we need to understand how to read the answers. For example, the first answer states that the sum of a natural number n and zero is n. The second answer states that the sum of zero and a positive integer m is m. The fourth answer states that one plus all the positive even numbers is all the odd numbers starting at 3.

Our system includes other relational arithmetic operators: $-^o$, which subtracts one number from another leading to a result, $\div^o$ and log^o, each of which has two results, one of which is a remainder, and exp^o, which is derived from log^o.

D.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 zeroᵒ
  (lambda (n)
    (≡ '() n)))

(define posᵒ
  (lambda (n)
    (fresh (a d)
      (≡ '(,a . ,d) n))))

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

(define full-adderᵒ
  (lambda (b x y r c)
    (condᵉ
      ((≡ 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 adderᵒ
  (lambda (d n m r)
    (condᵉ
      ((≡ 0 d) (≡ '() m) (≡ n r))
      ((≡ 0 d) (≡ '() n) (≡ m r)
       (posᵒ m))
      ((≡ 1 d) (≡ '() m)
       (adderᵒ 0 n '(1) r))
      ((≡ 1 d) (≡ '() n) (posᵒ m)
       (adderᵒ 0 '(1) m r))
      ((≡ '(1) n) (≡ '(1) m)
       (fresh (a c)
         (≡ '(,a ,c) r)
         (full-adderᵒ d 1 1 a c)))
      ((≡ '(1) n) (gen-adderᵒ d n m r))
      ((≡ '(1) m) (>1ᵒ n) (>1ᵒ r)
       (adderᵒ d '(1) n r))
      ((>1ᵒ n) (gen-adderᵒ d n m r)))))

(define gen-adderᵒ
  (lambda (d n m r)
    (fresh (a b c e x y z)
      (≡ '(,a . ,x) n)
      (≡ '(,b . ,y) m) (posᵒ 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)
    (cond^e
      ((≡ '() n) (≡ '() p))
      ((pos^o n) (≡ '() m) (≡ '() p))
      ((≡ '(1) n) (pos^o m) (≡ m p))
      ((>1^o n) (≡ '(1) m) (≡ n p))
      ((fresh (x z)
         (≡ '(0 . ,x) n) (pos^o x)
         (≡ '(0 . ,z) p) (pos^o z)
         (>1^o m)
         (*^o x m z)))
      ((fresh (x y)
         (≡ '(1 . ,x) n) (pos^o x)
         (≡ '(0 . ,y) m) (pos^o y)
         (*^o m n p)))
      ((fresh (x y)
         (≡ '(1 . ,x) n) (pos^o x)
         (≡ '(1 . ,y) m) (pos^o 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)
    (cond^e
      ((≡ '() q) (pos^o p))
      ((fresh (a_0 a_1 a_2 a_3 x y z)
         (≡ '(,a_0 . ,x) q)
         (≡ '(,a_1 . ,y) p)
         (cond^e
           ((≡ '() n)
            (≡ '(,a_2 . ,z) m)
            (bound-*^o x y z '()))
           ((≡ '(,a_3 . ,z) n)
            (bound-*^o x y z m))))))))
```

D.3 Additional Arithmetic Operators

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

```
(define =l^o
  (lambda (n m)
    (cond^e
      ((≡ '() n) (≡ '() m))
      ((≡ '(1) n) (≡ '(1) m))
      ((fresh (a x b y)
         (≡ '(,a . ,x) n) (pos^o x)
         (≡ '(,b . ,y) m) (pos^o y)
         (=l^o x y))))))

(define <l^o
  (lambda (n m)
    (cond^e
      ((≡ '() n) (pos^o m))
      ((≡ '(1) n) (>1^o m))
      ((fresh (a x b y)
         (≡ '(,a . ,x) n) (pos^o x)
         (≡ '(,b . ,y) m) (pos^o y)
         (<l^o x y))))))

(define ≤l^o
  (lambda (n m)
    (cond^e
      ((=l^o n m))
      ((<l^o n m)))))

(define <^o
  (lambda (n m)
    (cond^e
      ((<l^o n m))
      ((=l^o n m)
       (fresh (x)
         (pos^o x)
         (+^o n x m))))))

(define ≤^o
  (lambda (n m)
    (cond^e
      ((≡ n m))
      ((<^o n m)))))

(define ÷^o
  (lambda (n m q r)
    (cond^e
      ((≡ r n) (≡ '() q) (<^o n m))
      ((≡ '(1) q) (=l^o n m) (+^o r m n)
       (<^o r m))
      ((<l^o m n)
       (<^o r m)
       (pos^o q)
       (fresh (n_h n_l q_h q_l qlm qlmr rr r_h)
         (split^o n r n_l n_h)
         (split^o q r q_l q_h)
         (cond^e
           ((≡ '() n_h)
            (≡ '() q_h)
            (−^o n_l r qlm)
            (*^o q_l m qlm))
           ((pos^o n_h)
            (*^o q_l m qlm)
            (+^o qlm r qlmr)
            (−^o qlmr n_l rr)
            (split^o rr r '() r_h)
            (÷^o n_h m q_h r_h)))))))))

(define split^o
  (lambda (n r l h)
    (cond^e
      ((≡ '() 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)
       (split^o `(,b . ,n̂) r̂ '() h)))
    ((fresh (n̂ a r̂)
       (≡ `(1 . ,n̂) n)
       (≡ `(,a . ,r̂) r)
       (≡ '(1) l)
       (split^o n̂ r̂ '() h)))
    ((fresh (b n̂ a r̂ l̂)
       (≡ `(,b . ,n̂) n)
       (≡ `(,a . ,r̂) r)
       (≡ `(,b . ,l̂) l)
       (pos^o l̂)
       (split^o 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 '())))
```