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):

\[
\text{(define quine1 \( \lambda (x) (\text{list} \ (\text{list} \ (\text{quote} \ \text{quote}) \ x))\))}
\]

We can easily verify that quine1 evaluates to itself:

\[
\text{(equal? \ (eval quine1) quine1)} \Rightarrow \text{#t}
\]

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.3 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 \text{quote} and \text{list}. Our approach to avoiding tagging also has the important benefit of properly handling shadowing of language forms, such as \text{list}, \text{quote}, and \text{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 \text{symbol} and \text{number} 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

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

\[
\text{(equal? \ (run1 q (eval-exp o q) q) quine1)} \Rightarrow \text{#t}
\]
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 \(\text{symbol}\) and \(\text{number}\), which are similar in spirit to Scheme's \(\text{symbol}\) and \(\text{number}\) 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 \(\not\equiv\), \(\text{symbol}\), and \(\text{number}\) 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 \(\lambda\)-calculus (section 3); \(\lambda\)-calculus extended with \text{list} and \text{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 \text{list} and \text{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 \(\not\equiv\), \(\text{symbol}\), and \(\text{number}\) constraints (appendix B).

We provide a generalized version of the \text{pmatch} pattern matcher first presented in Byrd and Friedman (2007); the updated \text{pmatch} (appendix C), now called \text{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 \(\not\equiv\), \(\text{symbol}\), and \(\text{number}\) 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 \texttt{any}o. Some relational operators do not follow this convention: \(\equiv\) (entered as \(==\)), \texttt{cond} (entered as \texttt{conde}), and \texttt{fresh}. Similarly, \(\texttt{run}^q\) (\(q\) body) and \(\texttt{run}^e\) (\(q\) body) are entered as \(\texttt{run 5}\) (\(q\) body) and \(\texttt{run*}\) (\(q\) body), respectively.\footnote{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 \texttt{cond} \(e\) stands for 'every,' since every \texttt{cond} \(e\) clause may contribute answers.}

miniKanren extends Scheme with three operators: \(\equiv\), \texttt{fresh}, and \texttt{cond} \(e\) (three new operators, \(\not\equiv\), \(\text{symbol}\), and \(\text{number}\), are introduced in section 2.2). There is also \texttt{run}, which serves as an interface between Scheme and miniKanren, and whose value is a list. \texttt{fresh}, which syntactically looks like \texttt{lambda}, introduces into scope new lexical variables bound to new (logic) variables; \(\equiv\) unifies two terms. Thus

\[
\text{fresh} (x y z) (\equiv x z) (\equiv 3 y)
\]

would associate \(x\) with \(z\) and \(y\) with \(3\). This, however, is not a legal miniKanren program—we must wrap a \texttt{run} around the entire expression.

\[
\text{run}^1 (q) (\text{fresh} (x y z) (\equiv x z) (\equiv 3 y)) \Rightarrow (\_)
\]

The value returned is a list containing the single value \(\_\); we say that \(\_\) is the \textit{reified} value of the unbound variable \(q\) and thus can be any value. \(q\) also remains unbound in

\[
\text{run}^1 (q) (\text{fresh} (x y) (\equiv x q) (\equiv 3 y)) \Rightarrow (\_)
\]

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

\[
\text{run}^1 (y) \quad \text{run}^1 (q) \quad \text{run}^1 (y)
\]

\[
(\text{fresh} (x z) \quad \text{fresh} (x z) \quad \text{fresh} (x y))
\]

\[
(\equiv x z) \quad (\equiv x z) \quad (\equiv 4 z) \quad (\equiv 3 y)) \quad (\equiv 3 z) \quad (\equiv 3 y) \quad (\equiv q x)) \quad (\equiv 3 y)
\]

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

\[
\text{run}^1 (x) (\equiv 4 3) \Rightarrow ()
\]

We use \texttt{cond} \(e\) to get several values. Syntactically, \texttt{cond} \(e\) looks like \texttt{cond} but without \(\Rightarrow\) or \texttt{else}. For example,

\[
\text{run}^2 (q)
\]

\[
\text{fresh} (w x y)
\]

\[
(\text{cond} e)
\]

\[
((\equiv '(x ,w ,x) q)\quad (\equiv y w))
\]

\[
((\equiv '(w ,x ,w) q)\quad (\equiv y w))) \Rightarrow ((\_\_\_\_\_\_\_\_\_\_\_)\quad (\_\_\_\_\_\_\_\_\_\_\_))
\]

Although the two \texttt{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 \(\_\_\_\_\_\_\_\_\_\_\_\_) in the first value but \(\_\_\_\_\_\_\_\_\_\_\_\_) in the second value. The super-
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:

\[
\begin{align*}
\text{run}^* (q) \\
\text{let } \text{loop} () \\
\text{cond}^c \\
((\equiv \# f \ q)) \\
((\equiv \# t \ q)) \\
((\text{loop}()))) \Rightarrow \bot
\end{align*}
\]

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* succeeds while associating \(q\) with \#f, which accounts for the first value. When getting the second value, the second cond* line is tried, and the association made between \(q\) and \#f is forgotten—we say that \(q\) has been refreshed. In the third cond* 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.

\[
\begin{align*}
\text{define any}^o \\
\text{lambda } (q) \\
\text{cond}^c \\
((\text{any}^o \equiv \# f \ q))) \\
((\equiv \# t \ q)))
\end{align*}
\]

Consider the first example,

\[
\begin{align*}
\text{run}^* (q) \\
\text{cond}^c \\
((\text{any}^o \equiv \# f \ q))) \\
((\equiv \# t \ q)))
\end{align*}
\]

which does not terminate because the call to any\(^o\) succeeds an unbounded number of times. If * were replaced by \(5\), the second cond* line is tried, and the association made between \(q\) and \#f is forgotten—we say that \(q\) has been refreshed. Consider the second example,

\[
\begin{align*}
\text{run}^a \ (q) \\
\text{any}^o \\
\text{cond}^c \\
((\equiv \# f \ q)) \\
((\equiv \# t \ q)))
\end{align*}
\]

which does not terminate because the call to any\(^o\) succeeds an unbounded number of times. If * were replaced by \(5\), the second cond* line is tried, and the association made between \(q\) and \#f is forgotten—we say that \(q\) has been refreshed. The user should not be concerned with the order in which values are returned.

Now consider

\[
\begin{align*}
\text{run}^{10} \ (q) \\
\text{any}^o \\
\text{cond}^c \\
((\equiv \# f \ q)) \\
((\equiv \# t \ q))) \\
(((1 2 3 1 2 3 1 2 3 1)) \Rightarrow (1 2 3 1 2 3 1 2 3 1)
\end{align*}
\]

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\(^c\) lines loop indefinitely, other cond\(^c\) lines can contribute to the values returned by a run expression. However, we are not concerned with expressions looping indefinitely. For example,

\[
\begin{align*}
\text{run}^3 \ (q) \\
\text{let} ((\text{never}^o \ (\text{any}^o \equiv \# f \ # t))) \\
\text{cond}^c \\
((\equiv \# f \ q)) \\
((\equiv \# t \ q))) \\
((\equiv \# f \ # f \ # f \ # f) \Rightarrow \bot)
\end{align*}
\]

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

\[
\text{run}^2 \ (q) \ (\text{symbol}^o \ q) \\
(\equiv \# f \ q)) \\
((\equiv \# f \ q)) \Rightarrow (1)
\]

which evaluates to \((\omega \ (\text{sym} \ \omega)))\). 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:

\[
\begin{align*}
\text{run}^3 \ (q) \\
(\equiv \# f \ q)) \\
((\equiv \# f \ q)) \Rightarrow (1)
\end{align*}
\]

and

\[
\begin{align*}
\text{run}^3 \ (q) \\
(\equiv \# f \ q)) \\
(\equiv \# f \ q)) \Rightarrow (1)
\end{align*}
\]

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 \(\not\equiv\).

\[
\begin{align*}
\text{run}^4 \ (p) \ (\not\equiv \ (p \ 1)) \\
(\equiv \ 1 \ p)) \Rightarrow (1)
\end{align*}
\]

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

\[
\begin{align*}
\text{run}^4 \ (p) \ (\not\equiv \ 1 \ p)) \\
(\equiv \ 1 \ p)) \Rightarrow (1)
\end{align*}
\]

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

\[
\begin{align*}
\text{run}^4 \ (q) \\
(\equiv \ '(1 \ 2) \ '((p \ ,r)) \\
(\equiv \ 1 \ p)) \Rightarrow (1)
\end{align*}
\]

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

\[
\equiv \ '(1 \ 2) \ '((p \ ,r))
\]

by

\[
\equiv \ '(((2) \ '((p \ ,r))
\]

or even

\[
\equiv \ '(((1) \ ,(r)) \ '((p \ ) \ 2)).
\]

Now consider the run\(^1\) expression

\[
\begin{align*}
\text{run}^4 \ (q) \\
(\equiv \ '(1 \ 2) \ '((p \ ,r)) \\
(\equiv \ 1 \ p)) \\
(\equiv \ '(p \ ,r) \ q)) \Rightarrow ((1 \ ,2) \ ((\omega) \ ((\omega) \ 2)))
\end{align*}
\]

If we also associate \(r\) with 2, the run\(^1\) expression fails.
((run\(^3\) \((q)\))
   (fresh \((p \ r)\)
      \((\not\equiv (1 \ 2) \ (p \ r))\))
   \((\equiv 1 \ p)\)
   \((\equiv 2 \ r)\)
   \((\equiv (\lambda r \ q))\) => ()
)

Finally, we consider what happens when \((\equiv 2 \ r)\) is replaced by \((\text{symbol} \ r)\) in the previous example. Then the run\(^3\) expression succeeds with the answer \(((\lambda \omega) \ (\text{sym} \omega))\) which states that \(r\) can only be associated with a symbol. The reified constraint \((\not\equiv (\omega \ 2))\) (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 \(\lambda\)-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.

\[
\text{(define lookup}
\begin{array}{l}
\text{(lambda (x env)}
\text{(dmatch env)}
\text{\((() \ #\))}
\text{\(((y \ . \ v) \ . \ rest) \ (\text{guard} \ (eq? \ y \ x))\)}
\text{\(((y \ . \ v) \ . \ rest) \ (\text{guard} \ (not \ eq? \ y \ x))\)}
\text{\((\text{lookup} \ x \ rest)\))}
\end{array}
\]

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.

\[
\text{(define eval-exp}
\begin{array}{l}
\text{(lambda (exp env)}
\text{(dmatch exp)}
\text{\(((\lambda \ (\text{rator} \ . \ \text{rand}))))}
\text{\(((\text{let} \ ((\text{proc} \ (\text{eval-exp} \ \text{rator} \ \text{env}))))
\text{\(((\text{arg} \ (\text{eval-exp} \ \text{rand} \ \text{env}))))\)}
\text{\(((\text{dmatch \ proc})
\text{\(((\text{fresh} \ x \ . \ \text{body} \ . \ \text{env2}))
\text{\(((\text{eval-exp} \ \text{body} \ (((x \ . \ \text{arg}) \ . \ \text{env2})))))\)}
\text{\(((\lambda \ x) \ . \ \text{body})))
\text{\(((\text{guard} \ (\text{not-in-env} \ \lambda \ \text{env})))
\text{\(((\text{closure} \ x \ . \ \text{body} \ . \ \text{env}))\)}
\text{\(((\text{guard} \ (\text{symbol?} \ x)) \ (\text{lookup} \ x \ \text{env}))))\))\})}
\text{\(((\text{cond} \ t))\)}
\text{\(((\text{fresh} \ y \ \text{rest})))\)
\text{\(((\text{fresh} \ y \ \text{rest})))\)
\text{\(((\text{fresh} \ y \ \text{rest})))\)}
\text{\(((\text{fresh} \ y \ \text{rest})))\)}
\text{\(((\text{fresh} \ y \ \text{rest})))\)}
\text{\(((\text{fresh} \ y \ \text{rest})))\)}
\end{array}
\]

lookup\(^\#\) 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.

\[
\text{(run\(^*\) \((q) \ (\text{lookup} \ y \ '((x . \ \text{foo}) \ (y . \ \text{bar})))\)) \Rightarrow \(((y . \ \text{bar})))\)
\]

If \(x\) is not bound in \(env\), a call to lookup\(^\#\) will reach the base case and fail\(^4\), rather than returning \#f as in lookup.

\[
\text{(run\(^*\) \((q) \ (\text{lookup} \ w \ '((x . \ \text{foo}) \ (y . \ \text{bar})))\)) \Rightarrow ()}
\]

Each cond\(^\#\) clause in lookup\(^\#\) corresponds to a dmatch clause in loookup. Instead of pattern matching against \(env\), lookup\(^\#\) 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 \((\text{define fail} \ (eq? \ #f))\).
(not (eq? y x)) in lookup. As with dmatch, the order of clauses does not affect the meaning of a cond expression (but may affect its performance). Unlike with dmatch, the order of expressions within a cond clause is unimportant—there are no patterns or guards within a cond clause, only goals that succeed or fail.\footnote{Recall from section 2.1 that all goals within a cond clause must succeed for the entire clause to succeed.}

We can simplify lookup by removing the first cond clause (which always fails), and by lifting the unification of env.

\begin{verbatim}
(define lookup
  (lambda (x env t)
    (fresh (rest y v)
      (cond
       ((= x y) (≡ v t))
       ((≠ x y) (lookup o x rest t))))))

With lookup defined, we can now write eval-exp, which in turn relies on not-in-env. 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 'lambda env).

\begin{verbatim}
(define eval-exp
  (lambda (exp env val)
    (cond
     ((fresh (rator rand x body env2 a)
       (≡ '(rator ,rand) exp)
       (eval-exp rator env '((closure ,x ,body ,env2)))
       (eval-exp rand env a)
       (eval-exp body ((x ,a) ,.env2) val))
     ((fresh (x body)
       (≡ '(lambda ,x ,body) val)
       (not-in-env 'lambda env)))
     ((symbol exp) (lookup o exp env val))))

(define not-in-env
  (lambda (x env)
    (cond
     ((≡ '()) env)
     ((fresh (y 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:

\begin{verbatim}
(run5\(^5\) q)
  (fresh v
    (equiv (quote v) (quote v)))

This run\(^5\) expression generates five λ-expressions, and the closures to which they evaluate. The ≢ tags in the answers indicate disequality constraints between variables and the values they cannot assume. The last answer states that \(\dot{\omega}\) cannot be the symbol lambda and that \(\omega\) must be a symbol.

\begin{verbatim}
(((lambda (\omega) \dot{\omega}) ⇒ (closure \omega \dot{\omega}))
  (((lambda (\omega) \omega) (lambda \dot{\omega} \omega)) ⇒
  (sym \omega))
  (((lambda (\omega) (lambda \dot{\omega} \omega)) (lambda \dot{\omega} \omega)) ⇒
  (closure \omega \dot{\omega} ((\omega) closure \omega \dot{\omega} ()))
  (≠ ((\omega) lambda)))
  (((lambda (\omega) ((\omega) (\omega) \omega)) (lambda (\omega) \omega)) ⇒
  (closure \omega \dot{\omega} ((\omega) closure \omega \dot{\omega} (lambda (\omega) \omega)))
  (≠ ((\omega) lambda)))
  (sym \omega)))
\end{verbatim}

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:

\begin{verbatim}
((quote ,v) v)
\end{verbatim}

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:

\begin{verbatim}
((quote ,v) (guard (not-in-env 'quote env) v))
\end{verbatim}

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:

\begin{verbatim}
(((lambda (x) (lambda (y) x)) (lambda (z) z))
  (((lambda (x) (lambda (y) x)) (lambda (z) z))
   ((lambda (x) (lambda (y) x))
    (sym \omega))
   ((lambda (\omega) \omega) (lambda (x) (lambda (y) x)))
   (lambda (z) z))
   (sym \omega))
   ((lambda (\omega) \omega) (lambda (x) (lambda (y) x)) (lambda (z) z))
   (sym \omega)))
\end{verbatim}

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

\begin{verbatim}
(fresh v
  (≡ '(quote ,v) exp)
\end{verbatim}
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:

\[
(list . a) \rightarrow (\text{guard (not-in-env 'list env)})
\]

\[
\text{(map (lambda (e) (eval-exp e env)) a')}
\]

Similarly, we can add list to the miniKanren interpreter:

\[
(fresh (a'))
\]

\[
\text{(not-in-env 'quote env)}
\]

\[
\text{(not-closureo v)}
\]

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-env. 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:

\[
\text{(define proper-listo)}
\]

\[
\text{(lambda (exp env val)}
\]

\[
\text{(cond}
\]

\[
\text{((quote (quote env)) v))}
\]

\[
\text{(list . a') (guard (not-in-env 'list env))}
\]

\[
\text{(map (lambda (e) (eval-exp e env)) a'))}
\]

\[
(x (\text{guard (symbol? x)}) (cdr (lookup x env)))
\]

\[
\text{((rator rand env))}
\]

\[
\text{(guard (rator? rator env))}
\]

\[
\text{(let ((proc (eval-exp rator env))}
\]

\[
\text{(arg (eval-exp rand env))}
\]

\[
\text{(dmatch proc}
\]

\[
\text{((closure . x . body . env))}
\]

\[
\text{(eval-exp body '((. x . arg) . env)))}
\]

\[
\text{((lambda (. x) . body)}
\]

\[
\text{(guard (not-in-env 'lambda env))}
\]

\[
\text{((quote . x . body . env)))}
\]

where rator? is defined as

\[
\text{(define rator?}
\]

\[
\text{(lambda (x env)}
\]

\[
\text{((. a . d) #)}
\]

\[
(x (\text{guard (symbol? x)}) \text{(valid-name? x env)))}
\]

\[
\text{(define valid-name?)}
\]

\[
\text{(lambda (f-name env)}
\]

\[
\text{(dmatch env}
\]

\[
\text{((. x . v) . rest)}
\]

\[
\text{(cond}
\]

\[
\text{((eq? x f-name))}
\]

\[
\text{((not (eq? x f-name))}
\]

\[
\text{((valid-name? f-name rest)))}
\]

\[
\text{((not-op-name f-name op-names)))}
\]

\[
\text{(define not-op-name?}
\]

\[
\text{(lambda (x op-names)}
\]

\[
\text{(dmatch op-names}
\]

\[
\text{((. y . rest)}
\]

\[
\text{(cond}
\]

\[
\text{((eq? y x) #f)}
\]

\[
\text{((not (eq? y x)) (not-op-name? x rest))}
\]

\[
\text{((#t))}
\]

\[
\text{(define op-names (lambda quote list))}
\]

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

\[
\text{(rator? 'lambda)}
\]

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 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, 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.

\[
\text{(define eval-expo)}
\]

\[
\text{(lambda (exp env val)}
\]

\[
\text{(cond}
\]

\[
\text{((fresh (v))}
\]

\[
\text{((quote v) exp)}
\]

\[
\text{(not-in-env 'quote env) exp)}
\]

\[
\text{(not-closureo v)}
\]

\[
\text{(equiv v val)}
\]

\[
\text{((fresh (a'))}
\]

\[
\text{((quote (quote env) exp)}
\]

\[
\text{(not-in-env 'quote env) exp)}
\]

\[
\text{(not-closureo a')}\]
(... '(rator . rand) exp)
(eval-exp' rator env '(closure .x . body . env'))
(eval-exp' rand env a)
(eval-exp' body '((x . a) . env') val))
((fresh (x body)
  (≡ '(lambda (x) body) exp)
  (not-in-env' lambda env)
  (≡ '(closure .x . body . env) val))))

Proofs

4.2 Quines

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

(run1 (q) (eval-exp q '(q q)))
⇒
'(([(lambda _1) (list _1 (list 'quote _1))])
  '(lambda _1 (list _1 (list 'quote _1))))
  (≠ _1 (list _2)) ((_1 . quote)))
  (not-close _1)
  (sym _1)))

Sure enough, this is our old friend, quine1.

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

(run1 (q) (fresh (x y)
  (≠ x y)
  (eval-exp x '(y y))
  (eval-exp y '(x x))
  (≡ '(x . y) q))
⇒
'(([(lambda _1) (list _1 (list 'quote _1))])
  '(lambda _1 (list 'quote (list _1 (list 'quote _1)))))
  '(lambda _1 (list 'quote (list _1 (list 'quote _1))))
  (≠ _1 (list _2)) (_1 . quote)))
  (not-close _1)
  (sym _1)))

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 (e1 e2) would be written in the interpreted language as (app e1 e2). 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 Hallway 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 interpreters. 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 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 SFPJX. 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


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 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)).

(define eval-exp
(lambda (exp env val)
  (cond
    ((fresh v)
     (≡ '(quote ,v) exp)
     (not-in-env v quote env)
     (not-closure v)
     (not-into v)
     (≡ v val)))
    ((fresh a)
     (≡ '(list ,a) exp)
     (not-in-env 'list env)
     (not-closure a)
     (not-into a)
     (proper-list a env val))
    ((prim-exp exp env val)
     (symbol-exp (lookup exp env val))
     (fresh (rator x) (rands body env a res)
     (≡ '(rator ,rands) exp)
     (eval-exp rator env '(closure ,x ,body ,env))
     (proper-list (rands env a)
     (ext-env x a env res)
     (eval-exp body res val)))
    ((fresh (x) body)
     (≡ '(lambda ,x ,body) exp)
     (not-in-env 'lambda env)
     (≡ '(closure ,x ,body ,env val))))

(define ext-env
(lambda (x a env out)
  (cond
    ((≡ '() x) (≡ '() a) (≡ env out))
    ((fresh x a dxs daw env2)
     (≡ '([x ,d_xs] x))
     (≡ '([a ,d_aws] a))
     (≡ '([x ,a] ,d_ens env2)
     (ext-env dx daw env2 out))))

The primitive notions are booleans, numbers, pairs, and operations over them. The goal prim-exp 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.

(define prim-exp
(lambda (exp env val)
  (cond
    ((boolean-prim exp env val))
    ((number-prim exp env val))
    ((sub1-prim exp env val))
    ((zero?-prim exp env val))
    ((*-prim exp env val))
    ((cons-prim exp env val))
    ((car-prim exp env val))
    ((cdr-prim exp env val))

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.
(define number-primo
  (lambda (exp env val)
    (cond
      ((# t exp) (# t val))
      ((# f exp) (# f val)))))

(define cons-primo
  (lambda (exp env val)
    (fresh (a d v-a v-d)
      (fresh (cons . a . d) exp)
      (fresh (v-a . v-d) val)
      (not-closure o val)
      (not-into val)
      (not-in-env ^ cons env)
      (eval-exp a env v-a)
      (eval-exp ^ d env v-d) ))))

(define car-primo
  (lambda (exp env val)
    (fresh (p a d)
      (fresh (car . p) exp)
      (fresh (a val)
        (fresh (not-into val)
          (not-in-env ^ car env)
          (eval-exp ^ p env '(a . d)) ))))

(define cdr-primo
  (lambda (exp env val)
    (fresh (p a d)
      (fresh (cdr . p) exp)
      (fresh (d val)
        (fresh (not-into val)
          (not-in-env ^ cdr env)
          (eval-exp ^ p env '(a . d)) ))))

(define not-primo
  (lambda (exp env val)
    (fresh (e b)
      (fresh (not . e) exp)
      (cond
        ((# t b) ( # f val))
        ((# f b) ( # t val))
        (not-in-env ^ not env)
        (eval-exp ^ e env b ))) ))

(define number-primo
  (lambda (exp env val)
    (fresh (n)
      (fresh (numo . n) exp)
      (fresh (into . n) val)
      (not-in-env ^ numero env)))))

(define sub1-primo
  (lambda (exp env val)
    (fresh (e n -1)
      (fresh (sub1 . e) exp)
      (fresh (into . n-1) val)
      (not-in-env ^ sub1 env)
      (eval-exp ^ e env ' (into . n))
      (^ n (1 n -1)))))

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

(define *-prim o
  (lambda (exp env val)
    (fresh (e_1 e_2 . e_3 . e_4)
      (fresh (e_1 . e_2 . e_3)
        (fresh (e_1 into . e_2 . e_3)
          (not-in-env ^ * env)
          (eval-exp ^ e_2 env ' (into . e_3))
          (eval-exp ^ e_3 env ' (into . e_4))
          (^ e_3 n_1 n_2 n_3))))))

(define if-prim o
  (lambda (exp env val)
    (fresh (e_1 e_2 . e_3 . e_4)
      (fresh (e_1 . e_2 . e_3)
        (fresh (e_1 into . e_2 . e_3)
          (not-in-env ^ if env)
          (eval-exp ^ e_2 env t)
          (cond
            ((# t t) (eval-exp ^ e_2 env val))
            ((# f t) (eval-exp ^ 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 ^ q ^ (into . (build-num 6)))) ⇒
  ((numo (0 1 1))
  (sub1 (numo (1 0 1)))
  (((lambda () (numo (0 1 1))))
  (lambda (lambda (_ _ 0) (numo (0 1 1))) '. _)
  (not-into _)
  (not-closure _)
  (eval-exp t
  (fresh (e_1 e_2 . e_3 . e_4)
      (fresh (e_1 . e_2 . e_3)
        (fresh (e_1 into . e_2 . e_3)
          (not-in-env ^ if env)
          (eval-exp ^ e_2 env t)
          (cond
            ((# t t) (eval-exp ^ e_2 env val))
            ((# f t) (eval-exp ^ e_3 env val))))))

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 ^ q ^ (into . (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)
        (*) (f (f (sub1 n)))))
    (run 5 (eval-exp ^ rel-fact5 ^ (l) 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:
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 \( \equiv, \not\equiv, \text{symbol}^*, \text{number}^*, \text{cond}^* \), 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 values to variables. 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 \( \text{sym} \), 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.

\[
\begin{align*}
\text{(define a->s} & \ (\text{lambda} (a) (\text{car} a)) \\
\text{(define a->c}^* & \ (\text{lambda} (a) (\text{cdr} a))) \\
\text{(define a->t} & \ (\text{lambda} (a) (\text{caddr} a)))
\end{align*}
\]

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

\[
\begin{align*}
\text{(define-syntactic-lambda} & \ (\lambda_a) \\
\text{(syntax-rules} & \ (t) \\
\ ((_a e) & \ (\text{lambda} (a) e)) \\
\ ((_a (a : c^* t) e) & \ (\text{lambda} (a) \\
\ (\text{let} & \ ((s (a->s a)) (c^* (a->c^* a)) (t (a->t a))) \\
\ e))))
\end{align*}
\]

Because a sequence of packages may be infinite, we represent it not as a list but as an \( a^\infty \), 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 \( a \).

\[
\begin{align*}
\text{(define mzero} & \ (\text{lambda} () \ #f)) \\
\text{(define unit} & \ (\lambda_a (a) a)) \\
\text{(define choice} & \ (\text{lambda} (a f) (\text{cons} a f)))
\end{align*}
\]

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

\[
\begin{align*}
\text{(define-syntactic-lambda} & \ (\lambda_e) \\
\text{(syntax-rules} & \ ((_e e) \ (\text{lambda} (e))))) \\
\text{(define-syntactic-} & \ (\text{inc}) \\
\text{(syntax-rules} & \ ((_e e) \ (\lambda_e (e))))) \\
\text{(define empty-f} & \ (\lambda_e (\text{mzero})))
\end{align*}
\]

An iterative stream \( a \) is the same as \( \text{(choice a empty-f)} \). 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 \( a^\infty \)’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 \text{case}^\infty.

\[
\begin{align*}
\text{(define-syntactic case}^\infty & \ (\text{syntax-rules} ()))
\end{align*}
\]

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.
((λ e ((f) e1) ((a) e2) ((a f) e3))
(λ (a n) e))
(let ((a n e))
(cond
  ((not a n) e)
  ((procedure? a n) (let ((f (a n)) e1))
    (not (and (pair? a n))
      (procedure? (cdr a n)))))
  (let ((a (a n)) e2))
  (else (let ((a (a (car a n)) (f (cdr a n)))
    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)
          ((0) '())
          ((f) (take n f))
          ((a) (cons a (f))
          ((a f) (cons a (take (and n (not 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 g0 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 ()
    ((λ x g0 g ... )
      (take n
        (λ ()
          ((fresh x) g0 g ... )
          (λ (final-a)
            (choice ((reify x) final-a) 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 ... )
      (λ (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; Wandler 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 ()
    ((bind* (bind e g0 g ... )))
    (define bind
      (lambda (a n g)
        (case ((and (pair? a n))
          ((0) (mzero))
          ((f) (inc (bind (f) g)))
          ((a) (g a))
          ((a f) (mplus (g a) (λ () (bind (f) g))))))))

(define mplus
  (lambda (a n f)
    (case ((and (pair? a n))
      ((0) (f))
      ((f) (inc (mplus (f) f)))
      ((a) (choice a f))
      ((a f) (choice a (λ () (mplus (f) f))))))))

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

(define-syntax cond*
  (syntax-rules ()
    ((λ x g0 g ... )
      (λ (a)
        (inc
          (mplus*
            (bind* (g0 a) g ... )
            (bind* (g1 a) g ... ) ... )))))

(define-syntax mplus*
  (syntax-rules ()
    ((λ x g0 g ... )
      (λ (a)
        (mplus*
          (bind* (g0 a) g ... )
            (bind* (g1 a) g ... ) ... )))))

The function make-tag is the helper that is called from symbol* and number* 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 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
  (lambda (tag pred)
    (letrec ((rec (rec)
      (λ (u)
        ((fresh ()
          (λ (a : s c t)
            (let ((u (if (var? u) (walk u s) u)))
              (cond
                ((pair? u)
                  (cond
                    ((pred u)
                      (rec (car u))
                      (rec (cdr u)))
                    a))
                (else (mzero))))
                (not (var? u))
                (cond
                  ((pred u) (unit a))
                  (else (mzero)))))))))))

11
\begin{verbatim}
((ext-t u tag t s) \Rightarrow
 (lambda (t) (cond ((not (eq? t0 t))
 (let ((t (list (car t0))))
 (let ((c" (subsume t c"))))
 (unit (s ,c" ,t0)))))
 (else (unit a))))
 (else (mzero))))

rec))

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

(define ext-t
 (lambda (x tag t s)
 (let ((x (walk x))
 (let (loop 
 (cond ((null? t) (cons (cons x tag) t))
 ((not (eq? (walk (lhs (car t1)) s) tag)) (loop (cdr t)))
 ((eq? (rhs (car t)) tag) t)
 (works-together? (rhs (car t)) tag)
 (loop (cdr t)))
 (else #f))))))))

(define subsume
 (lambda (t c")
 (rump
 (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) \Rightarrow
 (lambda (pr-t)
 (not (subsumed-pr? (rhs pr-t)) u)))
 (else #f))))))))

Here is the implementation of the remaining interface operators.

(define bool
 (lambda (x) (cond ((\# t x))
 ((\# f x))))

(define symbol" (make-tag 'sym symbol?)

(define number" (make-tag 'num number?))

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

(define not-closure\" (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 (t t) (cond ((and (eq? t1 'sym) (eq? t2 'num)) #f)
 ((and (eq? t1 'num) (eq? t2 'sym)) #f)
 (else #t))))))

The definitions of \# 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 \# case, we pass the original substitution. So what happens in the \# 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, assuming that s0 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)
 (lambda (a : s c')
 (cond ((unify u v s) \Rightarrow
 (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'))
 (subsume t c')
 (unit '(s ,c' ,t))))))))))))

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

Just as \# checked t before extending c', \equiv must check c' and t (both of which might change) before it can succeed.

(define \equiv-
 (lambda (u v)
 (lambda (a : s c')
 (cond ((unify u v s) \Rightarrow
 (lambda (s0)
 (cond ((eq? s0 s) (unit a))
 (else (cons (car s0) (prefix-s (cdr s0) s)))))
 ))

12
\[
(\text{else (mzero)}) Davis)
(\text{else (mzero)}) Davis)
\]

\[
(\text{define verify-c}^*\rangle
(\text{lambda} (c^* s)
\text{(cond}
(\text{null? c^*}) '())
(\text{verify-c}^* (cdr c^*) s) \Rightarrow
\text{(let* ((s_0 (unify* (car c^*) s)))
(\text{lambda} (c^*)
\text{(cond}
(s_0 (and (not (eq? s_0 s))
(cons (prefix-s s_0 s) c^*))
\text{else c^*))
\text)))))
(\text{else #f})))))
\]

\[
(\text{define verify-t}
(\text{lambda} (t s)
\text{(cond}
(\text{null? t}) '())
(\text{verify-t (cdr t) s) \Rightarrow
\text{(let* ((t_0 (unify (rhs (car t)) t))
(\text{pred} (tag (pred tag)))
\text{letrec (rec}
(\text{lambda} (u)
\text{(let ((u (if (var? u) (walk u s) u)))
\text{lambda} (t)
\text{(cond}
(\text{var? u) (ext-t u tag t s))
(\text{pair? u) ( ordeal t s))
\text{else #f)))))
\text{else (and (pred u) t)))}})
\text{else #f)))
(\text{else #f)))))
\]

\[
(\text{define #s (≡ #f #f))}
(\text{define fail (≡ #f #f))}
\]

**B.2 miniKanren Helpers**

\[
(\text{define var (lambda (dummy) (vector dummy))})
(\text{define var? (lambda (x) (vector? x))})
(\text{define lhs (lambda (pr) (car pr))})
(\text{define rhs (lambda (pr) (cdr pr))})
\]

This definition of \text{walk} assumes that its first argument is a variable.

\[
(\text{define walk}
(\text{lambda} (x s)
\text{(let ((a (assq x s)))
\text{(cond}
(a (let ((u (rhs a)))
  (if (var? u) (walk u s) u)))
\text{else x)))})
\]

\[
(\text{define walk*}
(\text{lambda} (v s)
\text{(let ((v (if (var? v) (walk v s) v)))
\text{(cond}
(var? v) v)
(pair? v)
(cons (walk* (car v) s) (walk* (cdr v) s))
\text{else v))))
\]

\[
B.3 The Unifier
\]

Below is \text{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 \text{unify-nonpair} gets the two representatives, which might extend the substitution. There is no explicit recursion in \text{unify-nonpair}, but \text{valid?} calls a recursive function, \text{occursV}.

\[
(\text{define unify}
(\text{lambda} (u v s)
\text{(let ((u (if (var? u) (walk u s) u)))
\text{(v (if (var? v) (walk v s) v)))
\text{(cond}
(and (pair? u) (pair? v))
\text{(let ((s (unify (car u) (car v) s)))
\text{(and s
\text{(unify (cdr u) (cdr v) s)))
\text{else (unify-nonpair u v s)))}})))
\]

\[
(\text{define unify-nonpair}
(\text{lambda} (u v s)
\text{(cond}
(eq? u v) s)
(var? u)
\text{(and (or (not (pair? v)) (valid? u v s)))
\text{(cons '(u . v) s))}
(var? v)
\text{(and (or (not (pair? u)) (valid? v u s)))
\text{(cons '(v . u) s))}
(equal? u v) s)
\text{else #f})))))
\]

\[
(\text{define valid?}
(\text{lambda} (x v s)
\text{(not (occursV x v s)))}
\]

\[
(\text{define occursV}
(\text{lambda} (x v s)
\text{(let ((v (if (var? v) (walk v s) v)))
\text{(cond}
(var? v) (eq? v x))
(pair? v)
\text{(or (occursV x (car v) s)
\text{(occursV x (cdr v) s)))
\text{else #f)))))}
\]

**B.4 The Reifier**

The role of \text{reify} is to make the relevant information that is stored in the final state \text{final-a} (see \text{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 \text{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 \text{fresh} should not change the appearance of the answers. But this equality must hold for \text{c^*} and \text{t}, which is why we sort lexicographically.

\text{reify}-s is the heart of the reifier. \text{reify}-s takes an arbitrary value \text{v}, and returns a substitution that maps every distinct variable in \text{v} to a unique symbol. The trick to maintaining left-to-right ordering of the subscripts on these symbols is to process \text{v} from left to right, as can be seen in the \text{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 '()))
      (cond
        ((var? v)
          (let ((n (length r)))
            (let ((name (reify-name n)))
              (cond ((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 "." n" (number->string n))))))

(define reify
  (lambda (x)
    (λx (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'))
            (rem
             (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') (cons '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 r)
    (let rem-subsumed ((c* c*) (c' '()))
      (cond
        ((null? c*) #f)
        (else
         (let ((c (unify* (car c*) c))
           (or
            (and c (eq? c c))
            (subsumed? c (cdr c)))))
         (rem-subsumed (cdr c*)
          (cons (car c*) (c'))))))))

(define subsumed?
  (lambda (c c')
    (cond
      ((null? c') #f)
      (else
       (let ((c (unify* (car c*) c))
         (or
          (and c (eq? c c))
          (subsumed? c (cdr c)))))))
       (rem-subsumed (cdr c*)
        (cons (car c*) (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 't . x')
       (partition* y'))
      (else
       (let ((x (lhs (car t)))
         (y (cons (car t) y'))
         (part tag (cdr t) x' y'))))))

(define partition*
  (lambda (t)
    (cond
      ((null? t) '())
      (else
       (partition (rhs (car t)) 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 cond^a and cond^b, which can be used to prune the search tree of a program. The examples from Thin Ice of The Reasoned Scheme (Friedman et al.
2005) demonstrate how \texttt{cond\textsuperscript{a}} and \texttt{cond\textsuperscript{u}} can be useful and the pitfalls that await the unsuspecting reader. Also, we have included an additional operator \texttt{once\textsuperscript{a}}, defined in terms of \texttt{cond\textsuperscript{a}}, which forces the input goal to succeed at most once.

\begin{verbatim}
(define-syntax project
  (syntax-rules ()
    ((_ x . . .) g g' . . .)
      (lambda (a : c e k)
        (let ((x (walk x s)) . . .)
          ((fresh (g g' . . .) a))))))

(define-syntax cond\textsuperscript{a}
  (syntax-rules ()
    ((_ (g y . . .) (g y' . . .))
      (lambda (a)
        (inc
          (if\textsuperscript{a} ((g y a) g . . .)
            ((g y a) g y' . . .)))))))

(define-syntax cond\textsuperscript{u}
  (syntax-rules ()
    ((_ (g y . . .) (g y' . . .))
      (lambda (a)
        (inc
          (if\textsuperscript{u} ((g y a) g . . .)
            ((g y a) g y' . . .)))))))

(define-syntax if\textsuperscript{a}
  (syntax-rules ()
    ((_ (e g . . .) b . . .)
      (let loop ((a e))
        (case\textsuperscript{a} a
          (())
          ((if\textsuperscript{a} b . . .))
          ((f) (inc (loop (f))))
          ((a) (bind\textsuperscript{a} a e g . . .))
          ((a f) (bind\textsuperscript{a} a e g . . .))))))

(define-syntax if\textsuperscript{u}
  (syntax-rules ()
    ((_ (e g . . .) b . . .)
      (let loop ((a e))
        (case\textsuperscript{u} a
          (())
          ((if\textsuperscript{u} b . . .))
          ((f) (inc (loop (f))))
          ((a) (bind\textsuperscript{u} a e g . . .))
          ((a f) (bind\textsuperscript{u} a e g . . .))))))

(define-syntax once\textsuperscript{a}
  (lambda (g) (cond\textsuperscript{a} (g))))

\end{verbatim}

C. Generalized Pattern Matcher

This appendix gives a definition of \texttt{dmatch} that is more general in several ways than Oleg Kiselyov's \texttt{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 \texttt{dmatch}, as in the use of \texttt{example} in \texttt{h} below. We get more generality by not handling \texttt{quote} specially, which allows for certain common patterns to be specified that were previously not possible. Finally, there is no \texttt{else} clause and the order of the clauses is arbitrary, but only one pattern (plus guard) can succeed for each invocation of \texttt{dmatch}. Here is an example of \texttt{dmatch} using guards.

\begin{verbatim}
(define h
  (lambda (x y)
    (dmatch '(x . y) example
      ((x . a b)
        (guard (number a) (number b)) (+ a b))
      ((x . a . b . c)
        (guard (number a) (number b) (number c)) (+ a b c))))

(list h 1 2) (apply h '(1 (3 4))) ⇒ (3 8)
\end{verbatim}

In this example, a dotted pair is matched against two different kinds of patterns. In the first pattern, the value of \texttt{x} is lexically bound to \texttt{a} and the value of \texttt{y} is lexically bound to \texttt{b}. Before the pattern match succeeds, however, an optional guard (no side-effects allowed in guards) is run within the scope of \texttt{a} and \texttt{b}. The guard succeeds only if \texttt{x} and \texttt{y} are numbers; if so, then the sum of \texttt{x} and \texttt{y} is returned.

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

The overall syntax of \texttt{dmatch} looks like this:

\begin{verbatim}
match := (dmatch exp clause ...)
  | (dmatch exp name clause ...)
clause := (pattern guard exp ...)
  | (pattern guard boolean-exp ...)
pattern := , | exp
  | (pattern1 pattern2 ...) | (pattern1 . pattern2)
\end{verbatim}

Now we examine the implementation of \texttt{dmatch}. The main \texttt{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 \texttt{run-a-thunk} procedure.

\begin{verbatim}
(define-syntax dmatch
  (syntax-rules ()
    ((_ v (e . . .) ...)
      (let ((pkgs (dmatch-remexp v (e . . .) ...)))
        (run-a-thunk 'v v pkgs))
      ((v name (e . . .) ...)
        (let ((pkgs (dmatch-remexp v (e . . .) ...)))
          (run-a-thunk 'v v name pkgs))))

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)))
\end{verbatim}

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

\begin{verbatim}
(define-syntax dmatch-remexp
  (syntax-rules ()
    ((_ (rator rand ...)) cls ...)
      (let ((v (rator rand ...))))
\end{verbatim}

\begin{verbatim}
(match := (dmatch exp clause ...)
  | (dmatch exp name clause ...)
clause := (pattern guard exp ...)
  | (pattern guard boolean-exp ...)
pattern := , | exp
  | (pattern1 pattern2 ...) | (pattern1 . pattern2)
\end{verbatim}
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 cs ...) (dmatch-aux v cs ...)))
    ((. v cs ...) (dmatch-aux v cs ...))))

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 `eq?` instead of `eq`.

(define-syntax ppat
  (syntax-rules (unquote)
    ((. v (unquote var) k t kf) (let ((var v) k t))
     (let ((. v x y) k t kf)
       (let ((v (car v)) (v y k t kf))
         (let ((v (cdr v)) (v y k t kf))
           (let ((v (eq? v (quote v) k t kf)))
             (let ((v (lit k t kf)) (if (eq? v (quote v) k t 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 name pkg)
    (cond
      (null? pkg)
      (no-matching-pattern name v-expr v))
      (null? (cdr pkg))))

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 termi-
nation proofs for the individual operators.

A note on typography: `+` is entered as `pluso`, `-` is entered as `minus`, `*` is entered as `*o`, and `/` is entered as `/o`.

D. A Relational Arithmetic System

There are two kinds of improvements that should be resolved
by the compiler. First, `x` and `y` are not needed, so they
should not get bindings. The lexical variable `a` and `b` could
have replaced `x` and `y`, respectively. Second, `a` and `b`
should be parallel `let` bindings.
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 \(x + y = z^2\)?’, and ‘for which natural numbers \(x\) and \(y\) does \(x + y = 24\) hold?’ which can be expressed in miniKanren as

\[
\text{(run}^n(q) \ (\text{fresh} \ (x \ y \ z) \\
(\equiv \ '(x \ . \ y \ . \ z) \ q)))
\]

and

\[
\text{(run}^n(q) \ (\text{fresh} \ (x \ y) \\
(\equiv \ '(x \ . \ y \ (\text{build-num} \ 24)) \ q)))
\]

respectively.

In order to understand the answers to these \textit{runs}, 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 \(0\).

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 \textit{run} expressions above:

\[
\begin{align*}
&\text{((\_\_ \ \_\_))} \\
&\text{((\_\_ \ . \ -\_))} \\
&\text{((\_\_ \ . \ -\_ \ . \ -\_))} \\
&\text{((\_\_ \ 0 \ . \ -\_ \ . \ -\_))} \\
&\text{((\_\_ \ 1 \ 1 \ 0 \ 1))}
\end{align*}
\]

and

\[
\begin{align*}
&\text{(((\_\_ \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))} \\
&\text{(((\_\_ \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1))}
\end{align*}
\]

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.

\text{define build-num}
\begin{align*}
\text{(lambda} \ (n) \\
\text{(cond} \\
\text{(\equiv} \ '(\text{odd}\ n) \\
\text{(cons} \ 1 \\
\text{(\equiv} \ '(\text{build-num} \ (\div \ (- \ n) \ 2))) \\
\text{(\equiv} \ '(\text{and} \ (\equiv \ (\text{zero}\ n) \ n)) \\
\text{(cons} \ 0 \\
\text{(\equiv} \ '(\text{build-num} \ (\div \ n) 2))) \\
\text{))}))
\end{align*}

\text{define zero^o}
\begin{align*}
\text{(lambda} \ (n) \\
\text{(\equiv} \ '(\text{\_\_}) \ n))
\end{align*}

\text{define pos^o}
\begin{align*}
\text{(lambda} \ (n) \\
\text{(fresh} \ (a \ d) \\
\text{(\equiv} \ '(\text{.} \ a \ . \ d) \ n)))
\end{align*}

\text{define >1^o}
\begin{align*}
\text{(lambda} \ (n) \\
\text{(fresh} \ (a \ ad \ dd) \\
\text{(\equiv} \ '(\text{.} \ a \ . \ ad \ . \ . \ . \ dd) \ n))))
\end{align*}

\text{define full-adder^o}
\begin{align*}
\text{(lambda} \ (b \ x \ y \ c) \\
\text{(cond} \\
\text{((\equiv} \ 0 \ b) \text{((\equiv} \ 0 \ x) \text{((\equiv} \ 0 \ y) \text{((\equiv} \ 0 \ r) \text{((\equiv} \ 0 \ c))} \\
\text{((\equiv} \ 1 \ b) \text{((\equiv} \ 0 \ x) \text{((\equiv} \ 0 \ y) \text{((\equiv} \ 1 \ r) \text{((\equiv} \ 1 \ c))} \\
\text{((\equiv} \ 0 \ b) \text{((\equiv} \ 1 \ x) \text{((\equiv} \ 0 \ y) \text{((\equiv} \ 1 \ r) \text{((\equiv} \ 0 \ c))} \\
\text{((\equiv} \ 1 \ b) \text{((\equiv} \ 0 \ x) \text{((\equiv} \ 1 \ y) \text{((\equiv} \ 0 \ r) \text{((\equiv} \ 1 \ c))} \\
\text{((\equiv} \ 0 \ b) \text{((\equiv} \ 1 \ x) \text{((\equiv} \ 1 \ y) \text{((\equiv} \ 0 \ r) \text{((\equiv} \ 1 \ c))} \\
\text{((\equiv} \ 1 \ b) \text{((\equiv} \ 1 \ x) \text{((\equiv} \ 1 \ y) \text{((\equiv} \ 1 \ r) \text{((\equiv} \ 1 \ c))})
\end{align*}

\text{define adder^o}
\begin{align*}
\text{(lambda} \ (d \ n \ m) \\
\text{(cond} \\
\text{((\equiv} \ 0 \ d) \text{((\equiv} \ 0 \ m) \text{((\equiv} \ 0 \ n) \text{((\equiv} \ 1 \ n) \text{((\equiv} \ 1 \ m) \text{((\equiv} \ 1 \ r) \text{((\equiv} \ 1 \ c))} \\
\text{))}))
\end{align*}

\text{define gen-adder^o}
\begin{align*}
\text{(lambda} \ (d \ n \ m) \\
\text{(fresh} \ (a \ c \ e \ x \ y \ z) \\
\text{(\equiv} \ '(\text{.} \ a \ . \ x) \ n) \\
\text{))})
\end{align*}
D.3 Additional Arithmetic Operators

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

(define odd-\textsuperscript{\textcircled{o}})
(lambda (n m)
  (cond
    ((= (\textquoteleft{\textcircled{o}}) n) (\textquoteleft{\textcircled{o}}) m)
    ((< (\textquoteleft{\textcircled{o}}) n m) (\textquoteleft{\textcircled{o}}) m))
  )
)

(define <\textsuperscript{\textcircled{o}}>
(lambda (n m)
  (cond
    ((= (\textquoteleft{\textcircled{o}}) n) (\textquoteleft{\textcircled{o}}) m)
    ((< (\textquoteleft{\textcircled{o}}) n m) (\textquoteleft{\textcircled{o}}) m))
  )
)

(define <\textsuperscript{\textcircled{o}}>
(lambda (n m)
  (cond
    ((= (\textquoteleft{\textcircled{o}}) n) (\textquoteleft{\textcircled{o}}) m)
    ((< (\textquoteleft{\textcircled{o}}) n m) (\textquoteleft{\textcircled{o}}) m))
  )
)

(define \textsuperscript{\textcircled{o}}\div (\textsuperscript{\textcircled{o}}) q r)
(lambda (n r l h)
  (cond
    ((= (\textquoteleft{\textcircled{o}}) n) (\textquoteleft{\textcircled{o}}) h) (\textquoteleft{\textcircled{o}}) l)
    ((fresh (h \textsuperscript{\textcircled{o}}) n))
    ((\textquoteleft{\textcircled{o}}) (\textquoteleft{\textcircled{o}}) h)
    (\textquoteleft{\textcircled{o}}) r)
  )
)
(define log^o)
(lambda (n b q r)
  (cond^e
   ((≡ '(1) n) (≡ pos^o b) (≡ () q) (≡ () r))
   ((≡ () q) (<^o r n) (+^o b '(1) n))
   ((≡ () q) (>^o b) (=^o r n) (+^o b n))
   ((≡ () q) (pos^o b) (pos^o q) (+^o r n))
   ((≡ () q) (pos^o q) (≡ r n))
   ((≡ () q) (≡ r n))
   (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))))
  (<^o b n)
  (fresh (bw1 bw nq1 q1 s)
    (exp2^o b () bw1)
    (+^o bw1 '(1) bw1)
    (<^o q n)
    (fresh (q1 bwq1)
      (+^o q1 (1) qq)
      (+^o bw qqq bwq1)
      (<^o nq1 bwq1)
      (exp2^o n () nq1)
      (+^o nq1 '(1) nq1)
      (≡ nq1 bwq1) (≡ nq1 q1 qq))
    (fresh (bq1 q1 s qdh qd)
      (repeated-mul^o b q1 bq1)
      (≡ nq1 bw1 q1 s)
      (+^o q1 qdh qn)
      (+^o q1 qd q)
      (<^o qd qdh)
      (fresh (bq1 dq1 b)
        (repeated-mul^o b dq1 b)
        (+^o dq1 bq1)
        (+^o bq r n)
        (<^o n bq1)))))

(define exp2^o)
(lambda (n b q)

(define repeated-mul^o)
(lambda (n q nq)
  (cond^e
   ((≡ pos^o n) (≡ () q) (≡ () nq))
   ((≡ () q) (≡ n nq))
   (>^o q)
   (fresh (q1 nq1)
     (+^o q1 (1) qq)
     (repeated-mul^o n q1 nq1)
     (<^o nq1 n nq1))))

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