A Unified Approach to Solving Seven Programming Problems (Functional Pearl)

WILLIAM E. BYRD, University of Utah, USA
MICHAEL BALLANTYNE, University of Utah, USA
GREGORY ROSENBLATT, Toronto, Ontario, Canada
MATTHEW MIGHT, University of Utah, USA

We present seven programming challenges in Racket, and an elegant, unified approach to solving them using constraint logic programming in miniKanren.

CCS Concepts: • **Software and its engineering → Functional languages**; **Constraint and logic languages**; **Automatic programming**;

Additional Key Words and Phrases: relational programming, program synthesis, miniKanren, Racket, Scheme

ACM Reference format:
William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (September 2017), 26 pages.
https://doi.org/10.1145/3110252

1 INTRODUCTION

This pearl shows how a single unusual technique can solve a variety of programming problems. Each problem is presented as a challenge to the reader, and we invite you to develop your own idea of how each problem might be solved before reading ours. If you'd like to consider the challenges with a frame of mind uncolored by our approach, feel free to jump ahead to Section 1. If you'd like a sneak peek, read on.

We solve each challenge by using an interpreter for a subset of Racket, or for a slight variant of Racket. These interpreters are written as relations in the constraint logic programming language miniKanren. Each interpreter relates two arguments: an expression *expr* to be evaluated, and the value *val* to which *expr* evaluates. These arguments may be only partially complete, containing logic variables representing unknown parts. Our miniKanren implementation will use constraint solving and search to fill in the logic variables with expressions and values consistent with the semantics of Racket. By placing logic variables representing unknowns in different positions within the *expr* and *val* arguments we can express a wide range of interesting queries, which provides the flexibility needed to solve the variety of challenges we pose.

We also show that this technique works especially well for reasoning about programs written in a functional programming language such as Racket.

2475-1421/2017/9-ART8
https://doi.org/10.1145/3110252

A note about the problems. The problem in section 3 is taken directly from Byrd et al. [2012], and the problem from section 2 is a simple variant of an example in that work. We include these problems to provide background and understanding for the problems in sections 4 through 8, which are all novel.

2 99,000 WAYS TO SAY (I LOVE YOU) IN RACKET

In his Valentine's Day blog post, "99 ways to say '(I love you) in Racket," one of the authors [Might 2013] writes:

> In spite of their simplicity, lists often confound new Racket programmers. After lists, the many forms for expressing a computation in Racket take time to discover and then master. To address both of these points, I've created 99 roughly tweet-sized expressions in Racket that all evaluate to the list '(I love you).[1]

These examples, such as the three below, introduce students to an assortment of Racket's features.

```
(cdr '(Hark! I love you))
((lambda (a c b) (list a b c)) 'I 'you 'love)
(match #t
  [#f '(Not me)]
  [#t '(I love you)])
```

Inspired by a love of Racket, and a love of the list (I love you), we want to go even further.

Challenge 1. Come up with 99,000 Racket expressions that evaluate to the list (I love you), demonstrating a variety of expression types in the spirit of the blog post.

Stop and think: How would you solve Challenge 1?

We could follow Might's approach, and write the 99,000 Racket expressions by hand. However, even we don't love the list (I love you) *that* much. There must be a better way. . .

The semantics of a programming language can be defined by an evaluation relation, which relates programs to their behaviors. For example, an evaluation relation for Racket relates the expression (**let** ((*x* 5)) (+ *x x*)) with the value 10. An interpreter for Racket expressions is an implementation of the evaluation relation for a special case: the expression to be evaluated is known, and the value is to be derived. What if instead of writing a normal interpreter, we implemented the evaluation relation in a constraint logic programming language, designed to query relations with any arguments unknown? We could solve our problem by simply leaving the expression argument as an unknown and specifying the value argument to be the list (I love you).

To understand this approach we must first learn a little about how to write relations in *miniKanren*, an embedded domain specific language for constraint logic programming [Byrd and Friedman 2006; Friedman et al. 2005].

[1] Racket prints the list value (I love you) as the quotation expression '(I love you), and Might follows this convention. In this paper we show just the value. To replicate our results in Racket, evaluate (*print-as-expression* #f) before the examples.

2.1 Relational Programming in miniKanren

Let's take list concatenation as our example. Here's how we might define *append* in Racket:

```
(define append
  (lambda (l s)
    (cond
      [(null? l) s]
      [else (cons (car l) (append (cdr l) s))])))
```

We name results of intermediate expressions, in order to make our next transformation more clear.

```
(define append
  (lambda (l s)
    (cond
      [(null? l) s]
      [else
       (let* ((a (car l))
              (d (cdr l))
              (res (append d s)))
         (cons a res))])))
```

This two-argument *append* function can now be translated into the three-argument $append^o$ relation in miniKanren:

```
(define append^o
  (lambda (l s ls)
    (cond^e
      [(≡ '() l) (≡ s ls)]
      [(fresh (a d res)
         (≡ `(,a . ,d) l)
         (≡ `(,a . ,res) ls)
         (append^o d s res))])))
```

We'll explain the miniKanren language forms used in this definition, but first let's take a look at what $append^o$ can do.

The original *append* function can be used to concatenate two lists:

```
(append '(a b c) '(d e)) ⇒ (a b c d e)
```

The $append^o$ relation can also be used the same way. In order to use the relation, we construct a *query* using the **run*** form. A query consists of *query variables* and calls to relations that will constrain those variables. Using search and constraint solving, miniKanren attempts to find values for the query variables that satisfy the constraints imposed by the relations. We emulate the behavior of *append* by calling $append^o$ with concrete lists for the first two arguments, and the query variable *q* representing an unknown third argument:

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

More excitingly, we can use $append^o$ to *infer* the list *q* that, when prepended to the list (d e), produces (a b c d e):[2]

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

[2]These examples treating list concatenation as a relation will be familiar to any Prolog programmer.

A query needn't produce only a single answer. We can also infer all pairs of lists x and y that when appended together produce (a b c d e):

(**run*** (x y) (*append*^{o} x y '(a b c d e)))

⇒

((() (a b c d e))
 ((a) (b c d e))
 ((a b) (c d e))
 ((a b c) (d e))
 ((a b c d) (e))
 ((a b c d e) ()))

Now that we've seen the added expressivity we get from writing *append*^{o} as a relation, let's consider its implementation. The definition of *append*^{o} uses the three core logical operators from miniKanren: ≡, **fresh**, and **cond**^{e}. These operators construct goals representing logical assertions, which may succeed or fail.

(≡ t_1 t_2) is a goal which asserts that t_1 and t_2 have the same value. For example, (≡ 5 5) *succeeds*, while (≡ 5 6) *fails*. ≡ is implemented using *first-order unification*, essentially a bidirectional form of pattern matching. Unification operates on *terms*. A miniKanren term is either the empty list, a Boolean constant, a number, a symbol, a *logic variable*, or a pair of terms. Logic variables start out *fresh*—that is, they initially have no value—and may later obtain a value through calls to ≡. For example, assuming x is a fresh logic variable, (≡ 5 x) succeeds, and associates x with the value 5. A subsequent call (≡ 6 x) would fail, since x would already be associated with 5.

As mentioned, we can apply ≡ to terms that are pairs—for example, if x and y are fresh variables, the goal (≡ (*cons* 3 4) (*cons* x y)) would succeed, and associate x with 3 and y with 4. For succinctness we often write pairs and lists using Racket's **quasiquote** and **unquote** syntax. The single characters ` and , are shorthand for these forms, respectively. All of these expressions are equivalent:

(*cons* 3 (*cons* (*cons* x (**quote** y)) (*cons* 4 (**quote** ()))))
(*list* 3 (*cons* x (**quote** y)) 4)
(*list* 3 (*cons* x 'y) 4)
(**quasiquote** (3 ((**unquote** x) . y) 4))
`(3 (,x . y) 4)

If the variable x were bound to 5, each expression above would evaluate to (3 (5 . y) 4).

(**fresh** (x^* …) g g^* …) introduces lexically scoped, fresh logic variables x^* …; **fresh** also performs logical conjunction ('and') over the goals in its body, g g^* …, forming a new goal asserting that conjunction. The **fresh** expression

(**fresh** (y z) (≡ 5 y) (≡ 6 z) (≡ y z))

first introduces logic variables y and z, then performs a conjunction of the three calls to ≡; (≡ 5 y) and (≡ 6 z) succeed, but (≡ y z) fails (since y and z are associated with 5 and 6, which differ), causing the entire **fresh** expression to fail.

(**cond**^{e} (g_0 g_0^* …) (g_1 g_1^* …) …) constructs a goal that performs logical disjunction ('or') over its clauses. Each clause acts as a conjunction over the goals it contains. A simple **cond**^{e} expression is

(**cond**^{e} ((≡ 3 x) (≡ 4 y)) ((≡ 5 x)))

with the two clauses ((≡ 3 x) (≡ 4 y)) and ((≡ 5 x)). The first clause is a conjunction of (≡ 3 x) and (≡ 4 y). In this case both clauses succeed: the first clause associates x with 3 and y with 4; the

second clause produces a distinct answer that associates x with 5 but doesn't constrain y. Since both clauses succeed, the entire **cond**^{e} expression succeeds twice, and can produce two answers.

Looking at the definition of *append*^{o}, we see its body is a **cond**^{e} expression with two clauses. The first clause represents the base case, corresponding to the first clause of the *append* function's **cond** expression. The goal (≡ '() l) corresponds to the *null?* check, while (≡ s ls) corresponds to returning the value of s, with ls representing the appended result of l and s. The second clause corresponds to the second clause of the *append* function's **cond**. Having named the intermediate results in *append*, we can see how the unifications in this clause correspond to taking the pair l apart, and constructing a return value using the result of a recursive invocation of *append*^{o}. Because logical assertions can be made in any order, we are able to place the recursive invocation last in this clause. This ordering is preferable, since encountering a failing unification early can make it unnecessary to perform the recursion.

Let's also revisit our query:

(**run*** (q) (*append*^{o} '(a b c) '(d e) q)) ⇒ (((a b c d e)))

Here **run*** introduces the query variable q, then evaluates its body as a conjunction of goals, returning a list of all the answers it produces. Under the hood, miniKanren uses a search that traverses the tree of conjunctions and disjunctions constructed by **fresh** and **cond**^{e}. Each answer consists of a list of values unified with query variables, followed by any additional side conditions introduced by lazy constraints. In this case, there is only a single answer unifying a single query variable to a value that happens to be a list, resulting in a triply-nested list. Here the single answer found unifies q with (a b c d e). If we wanted miniKanren to stop searching after producing the first answer rather than continuing until it can prove there are no more, we could use **run** 1 rather than **run***.

2.2 Loving Lists via a Relational Interpreter

As we just saw, by treating list concatenation as a relation instead of a function we get additional, useful behavior. This is also true of more complicated programs, such as interpreters.

Imagine an interpreter for a subset of Racket, written as a miniKanren relation. This interpreter—which we might call *eval*^{o}—would take two arguments: an expression e to be evaluated, and the value v of that expression. For example,

(**run** 1 (q) (*eval*^{o} '((lambda (x) x) 5) q))

would return ((5)), indicating the Racket expression ((**lambda** (x) x) 5) evaluates to the value 5.

The good news is that *eval*^{o} actually exists! We've written an interpreter for a small—but interesting and Turing-complete—subset of Racket, including first-class multi-argument and variadic functions, **letrec** for defining recursive definitions, and a simple pattern matcher.[3] Our *eval*^{o} is inspired by, but significantly extends, the relational interpreter given in Byrd et al. [2012].

The even better news is that we can use *eval*^{o} to generate expressions that evaluate to (I love you)!

Stop and think: Construct a miniKanren query that uses *eval*^{o} to solve Challenge 1.

Let's start off small, by generating just a single expression that evaluates to (I love you):

(**run** 1 (q) (*eval*^{o} q '(I love you))) ⇒ (('(I love you)))

[3]Other than the pattern matcher, the language supported by *eval*^{o} is also a subset of Scheme.

Success! The expression '(I love you) evaluates to the list (I love you). This expression isn't very exciting, though. Let's try to find more interesting answers. . .

Emulating the blog post, we can replace **run** 1 with **run** 99. Here are a few of the 99 expressions generated, all of which evaluate to (I love you):

```
(list 'I 'love 'you)
((lambda () '(I love you)))
(cons 'I '(love you))
(and '(I love you))
(car (list '(I love you)))
```

Here is a slightly more complex answer:

```
(((lambda _0 _0) 'I 'love 'you)
 (sym _0))
```

The sym side-condition tells us that $_0$ in the expression ((**lambda** $_0$ $_0$) 'I 'love 'you) can be replaced by any legal Racket identifier (represented in our interpreter as a Racket symbol). For example,

((**lambda** *x x*) 'I 'love 'you) ⇒ (I love you)

However, it's not actually necessary to replace $_0$ with another identifier, since $_0$ is itself a valid identifier in Racket:

((**lambda** $_0$ $_0$) 'I 'love 'you) ⇒ (I love you)

This answer is especially interesting because (**lambda** *x x*) is a *variadic* function—that is, a function that takes any number of arguments—which returns the arguments passed to it as a list. The application ((**lambda** *x x*) 'I 'love 'you) is equivalent to the expression (*list* 'I 'love 'you).

Here is one more interesting answer:

```
((if _0 '(I love you) _1)
 (num _0))
```

The side condition (num $_0$) indicates that $_0$ in the expression (**if** $_0$ '(I love you) $_1$) represents an arbitrary numeric constant. Replacing $_0$ with a specific number, such as 42, yields the expression (**if** 42 '(I love you) $_1$). Since 42 is considered a true value in Racket, the 'else' arm of the **if** expression is never evaluated. Therefore $_1$ may be replaced by *any* legal Racket expression, and the overall expression will still evaluate to (I love you).

There's something important happening in miniKanren's implementation to make these queries work. We can think of all the ways of constructing an expression as forming an infinite tree, branching at every point where a different kind of subexpression could be selected. The execution of the *eval*^{o} relational interpreter searches through this tree for expressions that evaluate to (I love you).

Stop and think: Does it matter what search strategy miniKanren uses?

In fact, the search strategy matters a great deal. Searching an infinite tree requires care: for example, depth-first search could get stuck considering an infinite subtree that doesn't contain answers to our query. Even when depth-first search does find answers in an infinite tree, it might find only one kind of answer rather than the variety of answers we're looking for. Our example programs might then only use *list* and *car*, but never **lambda**. Thus it is important that miniKanren

uses a complete search, which interleaves the processes of searching different parts of the tree [Kiselyov et al. 2005]. Our interleaving search also biases towards branches which have shown progress in the form of intermediate results, rewarding them with an increased share of the search effort. This bias allows miniKanren's search to investigate promising branches much more deeply than would unbiased complete searches like breadth-first and iterative deepening depth-first search.

Now it's time to shift our query into overdrive:

(**run** 99000 (*q*) (*eval*^{o} *q* '(I love you)))

Here are two artisanal, free-range (I love you) expressions, hand-curated from the 99,000 answers:

```
((lambda ()
   (((lambda () cons)) 'I
    ((lambda _0 '(love you)) list 42))))
```

```
(car ((lambda (_0) (_0 '(I love you) 42 list)) list))
```

Both answers include a num side-condition, which we've replaced with the number 42 so that the expressions will run without error in Racket.

3 QUINES, TWINES, AND THRINES! OH MY!

In his classic paper, 'A micro-manual for LISP – Not the whole truth,' John McCarthy describes the rules for a LISP evaluation function, *value*, and then offers this challenge [McCarthy 1978]:

> Difficult mathematical type exercise: Find a list e such that *value e* = *e*.

Challenge 2. Solve McCarthy's exercise by finding a list *e* that evaluates to itself in Racket.

Stop and think: How would you solve Challenge 2?

We could try to construct such a list by hand, either through trial-and-error or through cleverness. If we were of a more theoretical bent, we could use Kleene's second recursion theorem [Kleene 1952] to construct *e*.

We adopt a different, more direct approach, originally presented by Byrd et al. [2012]. We can once again use *eval*^{o}, the relational interpreter we describe in section 2.2!

Stop and think: Write a miniKanren query that uses *eval*^{o} to find an expression *e* that evaluates to itself.

It's actually very simple to write such a query—just use the query variable as both the first and second arguments to *eval*^{o} (representing the "input" expression and "output" value, respectively):

(**run** 1 (*e*) (*eval*^{o} *e e*))

And, indeed, this query produces an expression that evaluates to itself:

(_0 (num _0))

This answer represents any expression $\__0$ such that $\__0$ is a number (represented by the (num $\__0$) side-condition). That is, the answer represents *any* number.

Expressions that evaluate to themselves have been named *quines* by Hofstadter [1979]. The single answer returned by our query represents *infinitely many* individual quines, since any numeric literal is trivially a quine in Racket—for example, the expression 42 evaluates to the value 42.

We've found a trivial expression that evaluates to itself. However, the actual challenge is to find a *list* that evaluates to itself. If we replace the **run** 1 in our query with **run** 3, we get back two more trivial quines: #t and #f, the self-evaluating Boolean constants in Racket.

The fourth answer, produced with **run** 4, is more interesting:

```
(((lambda (_0) (list _0 (list 'quote _0)))
  '(lambda (_0) (list _0 (list 'quote _0))))
 (≠ ((_0 closure)) ((_0 list)) ((_0 prim)) ((_0 quote)))
 (sym _0))
```

The first part of this answer is indeed a list that evaluates to itself:

```
((lambda (_0) (list _0 (list 'quote _0)))
 '(lambda (_0) (list _0 (list 'quote _0))))
```

⇒

```
((lambda (_0) (list _0 (list 'quote _0)))
 '(lambda (_0) (list _0 (list 'quote _0))))
```

Fun fact: Though McCarthy describes finding quines as a "difficult mathematical type exercise," the quine-generating query is the *simplest* possible query involving $eval^o$, in that it contains the minimal number of distinct symbols.

The side-conditions in the second part of the answer tell us that $\__0$ can be any legal Racket identifier, other than *closure*, *list*, *prim*, or *quote*. For example, we could replace $\__0$ with the identifier x:

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

However, it would not be legal to replace $\__0$ with the identifier *list*, since this would shadow the *list* function used in the body of the **lambda** expression:

```
((lambda (list) (list list (list 'quote list)))
 '(lambda (list) (list list (list 'quote list))))
```

Internally $eval^o$ uses closure and prim as datatype tags, which is why these names cannot be used in expressions. This leaky abstraction could be mitigated by generating unique symbols for these tags. To truly plug up the leak, we would also have to tag all pairs, which would permit the unambiguous use of these tags as values.

We can produce our non-trivial quine using **run** 1 rather than a **run** 4 if we unify e with the pair $`(,a . ,d)$, forcing e to be a list:

```
(run 1 (e)
  (fresh (a d)
    (≡ `(,a . ,d) e))
  (eval^o e e))
```

We can also use *eval*^{o} to generate *twines*—two distinct expressions p and q such that p evaluates to q and q evaluates to p:

```
(run 1 (p q)
  (≠ p q)
  (eval^o p q)
  (eval^o q p))
```

Our query uses the *disequality constraint* operator ≠ to assert that p and q must be different.

The result of the query indicates p is the expression

```
'((lambda (_0) (list 'quote (list _0 (list 'quote _0))))
  '(lambda (_0) (list 'quote (list _0 (list 'quote _0)))))
```

and q is the expression

```
((lambda (_0) (list 'quote (list _0 (list 'quote _0))))
 '(lambda (_0) (list 'quote (list _0 (list 'quote _0)))))
```

where, once again, $_0$ can be any legal Racket identifier, other than *closure*, *list*, *prim*, or *quote*. Importantly, $_0$ must have the same value in both p and q.

Insight: "Lazy constraints"—such as the disequality constraint ≠ and the *number*^{o} constraint used to construct the num side-condition—are an important part of constraint logic programming [Jaffar and Lassez 1987] that we take advantage of here. These constraints are critical to the performance of *eval*^{o}. The abstract representation of a number allowed by the *number*^{o} constraint prevents the quine-generating query from enumerating infinitely many trivial quines. Disequality constraints provide an important form of negation. Using unification alone, disequality between terms must be expressed by enumerating all possible combinations of values that differ, making twine generation hopelessly inefficient.

Stop and think: Write a query that generates a *thrine*—that is, three distinct expressions p, q, and r such that p evaluates to q, q evaluates to r, and r evaluates to p.

Here is the thrine-generating query, which requires a disequality constraint for each pairing of p, q, and r:

```
(run 1 (p q r)
  (≠ p q)
  (≠ p r)
  (≠ q r)
  (eval^o p q)
  (eval^o q r)
  (eval^o r p))
```

Chaining additional uses of *eval*^{o}, going from quines to twines to thrines, results in the query taking more time and memory. The thrines-generating query may not find a thrine before running out of memory, unless some of the primitives and forms are removed from the *eval*^{o} relation, which reduces the branching factor of the search.

4 LEXICAL VS. DYNAMIC SCOPE

Challenge 3. Many students struggle with the concept of lexical scope versus dynamic scope. To help these students out, generate 100 expressions that evaluate to 42 under lexical scope, and 137 under dynamic scope. To help focus on issues of scope, use a restricted subset of Racket that corresponds to the call-by-value λ-calculus extended with numeric constants.

Stop and think: How would you solve Challenge 3?

We've seen that we can synthesize programs that evaluate to a particular value—such as (I love you)—using a relational interpreter. Our solution to this challenge evaluates an unknown expression simultaneously in *two* relational interpreters: one implementing lexical scope and one implementing dynamic scope.

We need only a few language forms to reveal distinctions between the two scoping disciplines: lambda, variable reference, function application, and numeric literals for the return values specified in the challenge. Figure 1 shows the big-step operational semantics for the lexically-scoped version of this minimal language, along with the corresponding relational interpreter in miniKanren.

$$\frac{}{\rho \vdash n \Rightarrow n}\ \textsc{const}$$

```
[(numbero expr) (≡ expr out)]
```

$$\frac{\text{lambda} \notin \rho}{\rho \vdash (\text{lambda } (x)\ e) \Rightarrow (\text{closure } x\ e\ \rho)}\ \textsc{abs}$$

```
[(fresh (x body)
   (≡ `(lambda (,x) ,body) expr)
   (symbolo x)
   (≡ `(closure ,x ,body ,env) out)
   (not-in-envo 'lambda env))]
```

$$\frac{\rho(x) = v}{\rho \vdash x \Rightarrow v}\ \textsc{ref}$$

```
[(symbolo expr) (lookupo expr env out)]
```

$$\frac{\begin{array}{l}\rho \vdash e_1 \Rightarrow (\text{closure } x\ e_c\ \rho_c) \\ \rho \vdash e_2 \Rightarrow v_2 \\ \rho_c, x \mapsto v_2 \vdash e_c \Rightarrow v_3\end{array}}{\rho \vdash (e_1\ e_2) \Rightarrow v_3}\ \textsc{app}$$

```
[(fresh (e1 e2 val x body cenv new-env)
   (≡ `(,e1 ,e2) expr)
   (eval-expr-lexo e1 env
     `(closure ,x ,body ,cenv))
   (eval-expr-lexo e2 env val)
   (ext-envo x val cenv new-env)
   (eval-expr-lexo body new-env out))]
```

```
(define (eval-expr-lexo expr env out)
  (conde
```

```
)))
```

Fig. 1. Correspondence between inference rules for the call-by-value λ-calculus (with lexical scope) and the evaluation relation.

In the semantics, $\rho \vdash e \Rightarrow v$ means the expression e evaluates to the value v in the environment ρ. The metavariable n represents any number, v any value (which includes numbers and closures), x any identifier, e any expression, and ρ any environment (a mapping from identifiers to values).

The evaluation judgement is implemented as the miniKanren relation *eval-expr-lex*^{o}. In our queries we'll use the relation *eval-lex*^{o} which just calls *eval-expr-lex*^{o} with the empty environment. We point out the interesting features of the miniKanren code corresponding to each inference rule:

CONST The lazy *number*^{o} constraint ensures the clause only applies when *expr* is a number.

ABS Shadowing is allowed, so the rule only applies when **lambda** isn't bound in the environment. The recursive *not-in-env*^{o} relation implements this restriction. Because miniKanren uses first-order unification, we must use data structural representations of expressions, environments, and closures. Expressions are represented as s-expressions, environments as association lists, and closures as lists containing the symbol 'closure, the code for the procedure, and the environment where the lambda was evaluated.

REF The recursive *lookup*^{o} relation implements variable lookup in the environment (represented as an association list). The lazy *symbol*^{o} constraint asserts that the expression is a symbol representing a lexical variable.

APP The *ext-env*^{o} relation extends the environment by unifying *new-env* with `` `((,x . ,val) . ,cenv) ``.

Insight: These semantics don't exactly correspond to a subset of Racket. We only care about the behavior of programs that terminate and don't produce errors. As such, we can cheat a little and get away with it! For example, our interpreter won't evaluate the remaining expressions in an application if the first expression, the operator, doesn't evaluate to a closure. In Racket, (5 ((**lambda** (*x*) (*x x*)) (**lambda** (*x*) (*x x*)))) loops infinitely, but its evaluation in *eval-lex*^{o} terminates with failure. Even better, our optimization makes the evaluation of the application (5 *e*), where *e* is an unknown expression, terminate with failure as well. Without this optimization, miniKanren would have to generate, and evaluate, each of the infinitely many possible expressions *e*.

Our "cheat" to optimize application is valid because our interpreter implements a purely functional language. Any successful, terminating evaluation must eventually check that the value of the operator expression is a closure. What might happen if we applied this optimization to an interpreter for a language with exceptions? For a language with user-defined exceptions that can be thrown, but not caught, we might treat throwing an exception as failure. Our optimization would remain correct. However, we might instead write an interpreter for a language where exceptions can be caught, or where we want to report details of the exception rather than fail. Then the interpreter must apply the optimization only when it is guaranteed that evaluation will eventually check that the value of the operator expression is a closure. First-class control operators like *call/cc* present a similar challenge.

Creating a variant of the interpreter for dynamic scope requires only a tiny change to the application rule: rather than extending *cenv*, the environment from the closure, we extend *env*, the environment at the call site. We'll refer to this variant as *eval-dyn*^{o}.

$$\frac{\rho \vdash e_1 \Rightarrow (\text{closure } x\ e_c\ \rho_c) \quad \rho \vdash e_2 \Rightarrow v_2 \quad \rho, x \mapsto v_2 \vdash e_c \Rightarrow v_3}{\rho \vdash (e_1\ e_2) \Rightarrow v_3}\ \text{APP-DYN}$$

```
[(fresh (e1 e2 val x body cenv new-env)
   (≡ `(,e1 ,e2) expr)
   (eval-expr-dyn^o e1 env
     `(closure ,x ,body ,cenv))
   (eval-expr-dyn^o e2 env val)
   (ext-env^o x val env new-env)
   (eval-expr-dyn^o body new-env out))]
```

We're ready to solve the challenge! Our query applies each of the two interpreters to the same unknown expression *expr*, asserting that in each interpreter *expr* evaluates to the right number.

```
(run 1 (expr)
  (eval-dyno expr 137)
  (eval-lexo expr 42))
⇒
((((lambda (_0)
     (((lambda (_0)
         (lambda (_1) _0))
       42)
      _2))
    137)
  (≠ ((_0 _1)) ((_0 lambda))) (num _2) (sym _0 _1)))
```

We can easily ask for many more programs; a **run** 100 query finds 100 in 1.7 seconds on an early 2013 MacBook Pro.

Curiously, the query is very slow if the *eval-dyno* call is reordered after the *eval-lexo* call. It turns out that this reordered query spends most of its time evaluating non-terminating programs!

Relatively simple programs in the language of *eval-dyno* can express non-terminating computations; some of these programs may terminate in *eval-lexo*. For example, here's a program that evaluates to 42 under lexical scope but does not terminate under dynamic scope:

```
((lambda (x)
   ((lambda (x) (x 1))
    (lambda (y) (x 2))))
 (lambda (z)
   42))
```

Stop and think: Why does the evaluation of this expression never terminate with dynamic scope?

After the outermost function application, the subexpression

```
((lambda (x) (x 1))
 (lambda (y) (x 2)))
```

is evaluated. The parameter of the first **lambda** expression binds *x* to the procedure resulting from the second **lambda** expression. The body of that procedure, (*x* 2), is evaluated while the binding is still present in the dynamic environment, making it a recursive call.

Program synthesis tools generally need to take special care to avoid non-terminating programs, and sometimes employ termination checking to rule them out. How is it that our relational interpreters don't get totally stuck when evaluating a non-terminating program?

Instead of exploring the alternative solutions represented by different **cond**^{e} clauses in sequence, the miniKanren search interleaves the work. A non-terminating computation in one clause will consume computational effort but cannot entirely halt work on other solutions. Even with an interleaving search strategy, however, we can still get in trouble. If many non-terminating computations stack up, they can starve work on other candidate solutions for resources. In many circumstances our search is clever enough to avoid that possibility, but here it fails.

In the reordered query with *eval-lex*^{o} first, the search quickly finds many partial programs that evaluate to 42 in lexical scope. Each of these programs are subsequently evaluated under dynamic scope, and as in the example above, some will never terminate. The implementation of conjunction in miniKanren assigns half of the computational effort to evaluating the first lexical scope solution under dynamic scope, a quarter to the second, an eighth to the third, and so on. That strategy works well when each evaluation quickly succeeds or fails. When those computations fail to terminate, however, later solutions are starved for time.

Why does the query work better with *eval-dyn*^{o} first? Our *eval-lex*^{o} language doesn't provide a form like **letrec** for direct recursion, so non-terminating programs require a more complex construction like the Z-combinator (the call-by-value Y-combinator). As a result, the programs that *eval-dyn*^{o} finds before solving our query happen to terminate in *eval-lex*^{o}.

5 RELATIONALITY FOR FREE

Challenge 4. We've seen the flexibility of the *append*^{o} relation in section 2.1. Is there a way to get the same flexibility, but using the standard Racket definition of the *append* function?

Stop and think: How would you solve Challenge 4?

One solution to this problem would be a Racket-to-miniKanren compiler, or a macro with the same functionality. However, if *eval*^{o} implements enough of Racket to run *append*, we can solve this problem directly: we just run the Racket definition of *append* inside a query to *eval*^{o}.

To keep the implementation of *eval*^{o} simple, we define *append* using **letrec** instead of **define**, and using **if** instead of **cond**. Naturally we can run *append* forwards:

```
(run 1 (q)
  (eval^o
    `(letrec ([append (lambda (l s)
                        (if (null? l)
                          s
                          (cons (car l) (append (cdr l) s))))])
       (append '(a b c) '(d e)))
    q))
⇒
(((a b c d e)))
```

Remember how we used *append*^{o} as a relation in section 2.1:

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

Because **run*** returned a single answer, miniKanren has proven that no other answers exist: only (a b c) can be prepended to (d e) to produce (a b c d e).

The *append*^{o} relation has three arguments, with the last one representing the result of appending the first two arguments. When evaluating a call to *append* in the relational interpreter, the second argument to *eval*^{o} serves a similar purpose, representing the return value of the *append* function.

This means we can use *append* as a relation! We pass the logic variable *q* as the first argument to *append*, using **unquote** (,) to escape the **quasiquote** (`) form wrapping the **letrec**.

```
(run 1 (q)
  (eval^o
    `(letrec ([append (lambda (l s)
                        (if (null? l)
                            s
                            (cons (car l) (append (cdr l) s))))])
       (append ,q '(d e)))
     '(a b c d e)))
⇒
(('(a b c)))
```

It seems to work. Let's try the same query with **run*** to see whether evaluating *append* in the interpreter can replicate *append^o*'s ability to prove that only one answer exists.

Alas, we'll be waiting a long time indeed; the **run*** query never returns a result. There are two possible causes: either there are infinitely many answers, or there are finitely many but miniKanren can't prove this. If there are many answers, a **run** 2 query should terminate with two answers. If there is only the one answer, the **run** 2 query should loop forever just like the **run*** query.

Stop and think: Will the **run** 2 query return two answers, or will it loop forever?

run 2 actually terminates with these two answers:

```
⇒
(('(a b c))
 (((lambda _0 '(a b c))) (≠ ((_0 quote))) (sym _0)))
```

What's **lambda** doing there?

Whereas *append^o* operates on lists, the relational interpreter operates on expressions. The first argument to *append* can be any of an infinite class of expressions that evaluate to the list (a b c)! And one such expression is ((**lambda** $_0$ '(a b c))). The fact that we're synthesizing expressions rather than values also explains a subtle change you may have noticed in the first answer: the query finds the quotation expression '(a b c) rather than the value (a b c).

Stop and think: How can we recover the behavior of *append^o*, where **run*** proves there is only one list *q* that when prepended to (d e) produces (a b c d e)?

The smallest of changes will do: just add a quote before the unquoted logic variable *q*. For each value, there is only one quotation expression that evaluates to that value.

```
(run* (q)
  (eval^o
    `(letrec ([append (lambda (l s)
                        (if (null? l)
                            s
                            (cons (car l) (append (cdr l) s))))])
       (append ',q '(d e)))
     '(a b c d e)))
⇒
(((a b c)))
```

We've found that *append* in the relational interpreter has all the power of the *append*^{o} relation implemented in miniKanren. Even better, the relational behavior of *append* and the interpreter's ability to generate expressions combine; we can now query for Racket expressions whose values satisfy a given call to *append*.

Let's go even further. Not only can we generate expressions in argument position of the call to *append*, we can generate expressions for the implementation of *append*. We can even run the (as yet partially unknown) *append* function backwards. We can do all of these things in a single query:

```
(run 1 (x y)
  (eval^o
    `(letrec ([append (lambda (l s)
                        (if (null? l)
                          s
                          ;; We use ,x here to leave part of the definition unknown.
                          (cons ,x (append (cdr l) s))))])
       (list (append ,y '(c d e)) (append '(f g h) '(i j))))
    '((a b c d e) (f g h i j))))
```

Let's discuss how this program is represented as a miniKanren term so that we can explain what is happening. A term can be either *fully instantiated* (containing no logic variables), *partially instantiated* (having some fixed structure, but containing one or more logic variables), or *fully uninstantiated* (a single logic variable with no associated value). In this case we've left part of the definition of *append* uninstantiated, representing the missing code with the logic variable x.

The program calls *append* twice: once with the output fully instantiated but part of the input unknown, represented by the logic variable y, and once with fully instantiated input and output. As the relational interpreter evaluates the first call to *append*, it synthesizes assignments of x and y such that (*append* y '(c d e)) evaluates to (a b c d e). The second call selects from those assignments a solution with an x such that (*append* '(f g h) '(i j))) evaluates to (f g h i j). The query instantiates x to the correct code snippet (car l), and y to the correct missing argument, '(a b). This query *simultaneously* performs code synthesis and bidirectional evaluation.

At this point, it's natural to wonder how much of *append* can be synthesized with this technique. With a straightforward, unoptimized implementation of *eval*^{o}, only small parts of *append* can be synthesized in a reasonable amount of time. Can we do better? By optimizing a few key portions of *eval*^{o}, we can in fact synthesize the entire body of *append* from just a few examples of its use! We'll briefly describe the optimizations that enable this degree of synthesis, but these details aren't necessary for an understanding of the rest of the paper.

The most significant optimization we implement orders goals carefully when performing procedure application. We evaluate the operator expression first to determine whether its value is a closure. If it's a closure, we next evaluate operand expressions that are at least partially instantiated, followed by the closure body, and then finally the fully uninstantiated operand expressions.

This order best takes advantage of known information. Evaluating partially instantiated operand expressions is assumed to be straightforward, and doing this first means that the values of corresponding variable references in the closure body do not need to be guessed blindly. Evaluating the closure's body before evaluating uninstantiated operand expressions reduces the number of branching computations between guessing and testing: guesses about the body can be immediately tested against the expected result of the application, whereas uninstantiated operand expressions cannot be immediately tested.

Consider this query:

```
(run 1 (e)
  (eval^o
    `(let ((x 7))
       ((lambda (a d)
          (cons a d))
        ,e x))
    '(5 . 6)))
```

Within the body of the **let** we would first evaluate the **lambda** expression to a closure, then x to 7, followed by the closure body (*cons a d*) where d is bound to 7 and a is still unknown. Only if those succeed would we start guessing terms for e. In this case the query fails before we make any guesses. Evaluating the closure body produces a pair whose *cdr* is 7, which cannot possibly fit our expected result of (5 . 6), whose *cdr* is 6.

Because we are implementing a purely functional language, we can perform these reorderings without changing a program's behavior. If evaluating e could produce side effects, such as mutating the value of x, we would have to be more careful.

Here are some other optimizations that help a great deal:

- To reduce uninformed guessing when encountering branches, we defer invoking the goals constructed by some uses of **cond**^{e} when it seems likely that more than one clause will succeed. Deferred goals are queued up to be retried later, after all informed guesses have been made, with the hope of having learned more useful information by the time we retry.
- Though we continue to support a complete search, we first try a depth-limited search, which prunes search branches that get very deep. These branches are often unpromising. If no answers are found, we fall back on the unlimited, complete search. The bias in our search described on page 6 is also important for the remaining challenges; a simple iterative deepening search won't work.
- To greatly reduce the expense of searching for useful conditional expressions, we restrict conditions to expressions that evaluate to Boolean constants #t and #f (as opposed to other "true" Racket values, such as '5').
- We also perform some other minor optimizations, such as hand-tweaking the weighting of some **cond**^{e} clauses in *eval*^{o}, favoring expression types that are more common.

Let's get back to synthesizing *append*. We'll start by providing a couple of example calls.

```
(run 1 (defn)
  (fresh (body)
    (≡ defn `(append (lambda (xs ys) ,body)))
    (eval^o
      `(letrec (,defn)
         (list (append '() '())
               (append '(1) '(2))))
      '(() (1 2)))))
⇒
(((append
  (lambda (xs ys)
    (if (null? ys)
        ys
        '(1 2))))))
```

This definition doesn't look right. Although it supports our specific query, it isn't general enough to append arbitrary lists. What went wrong?

Because the search simply looks for any definition that works, a simple way to satisfy the constraints is to simply return our example output verbatim. This is why we see our expected second result embedded directly in the definition. How do we get around this?

One solution is to prohibit the definition from mentioning our example data directly. The constraint ($absent^o$ t_1 t_2) prevents term t_1 from appearing anywhere in term t_2.[4] We can use $absent^o$ to prevent numbers from our examples from appearing in the definition we're synthesizing. We provide an $absent^o$ constraint for each number we mention.

```
(run 1 (defn)
  (fresh (body)
    (absent^o 1 defn) (absent^o 2 defn)
    (≡ defn `(append (lambda (xs ys) ,body)))
    (eval^o
      `(letrec (,defn)
         (list (append '() '())
               (append '(1) '(2))))
      '(() (1 2)))))
⇒
(((append
  (lambda (xs ys)
    (if (null? ys)
        ys
        (cons (car xs) ys))))))
```

Though our example result is no longer embedded in the definition, the definition still is not general enough to append arbitrary lists. It handles exactly the cases of the empty and single element lists. The definition can't handle larger lists because it's not recursive!

Stop and think: How can we synthesize a more general version of *append*?

Adding a couple more usage examples helps us find a more general definition.

```
(run 1 (defn)
  (fresh (body)
    (absent^o 1 defn) (absent^o 2 defn) (absent^o 3 defn) (absent^o 4 defn)
    (≡ defn `(append (lambda (xs ys) ,body)))
    (eval^o
      `(letrec (,defn)
         (list (append '() '())
               (append '(1) '())
               (append '(1) '(2))
               (append '(1 2) '(3 4))))
      '(() (1) (1 2) (1 2 3 4)))))
⇒
```

[4] The miniKanren implementation we use for this pearl restricts t_1 to be a literal symbol or number rather than a term. Other implementations can handle the general version of the constraint.

```
(((append
   (lambda (xs ys)
     (if (null? xs)
         ys
         (if (null? ys)
             xs
             (cons (car xs) (append (cdr xs) ys)))))))))
```

This query responds with a fully general definition of *append*, implemented as a recursive procedure. Why isn't the answer, once again, specific to exactly the examples provided? Given enough examples, the simplicity of the recursive solution makes it easier to find than the overly-specific solution!

Though we can synthesize *append* from scratch, what if a standard library were available? Would we be able to define *append* in terms of existing definitions? Let's try providing *fold-right*.

```
(run 1 (defn)
  (fresh (body)
    (absent^o 1 defn) (absent^o 2 defn) (absent^o 3 defn) (absent^o 4 defn)
    (≡ defn `(append (lambda (xs ys) ,body)))
    (eval^o
      `(letrec ((fold-right (lambda (f acc xs)
                              (if (null? xs) acc
                                  (f (car xs) (fold-right f acc (cdr xs))))))
                 ,defn)
         (list (append '() '())
               (append '(1) '())
               (append '(1) '(2))
               (append '(1 2) '(3 4))))
      '(() (1) (1 2) (1 2 3 4)))))
⇒
(((append
   (lambda (xs ys)
     (if (null? ys)
         xs
         (fold-right cons ys xs))))))
```

With *fold-right* available in the environment, *append* is quickly defined in terms of it!

While our optimized relational interpreter can both synthesize recursive functions and take advantage of higher-order functions, it struggles with larger programs. The search space grows exponentially with the size of the smallest correct definition of the program to be synthesized. Our current implementation can't synthesize complete recursive programs much larger than *append*.

6 TAKING TEST-DRIVEN DEVELOPMENT SERIOUSLY

Test-driven development asks us to write our test cases before we write our programs, so that we can receive immediate feedback on the code we've written. Say we're writing a program *remove* that removes all occurrences of a symbol x that occur as top-level elements in a list *ls*. Our first job is to describe *remove* with a set of tests:

```
(list (remove 'foo '())
      (remove 'foo '(foo))
      (remove 'foo '(1))
      (remove 'foo '(2 foo 3))
      (remove 'foo '((4 foo) foo (5 (foo 6 foo)) foo 7 foo (8))))
```

⇒

```
'(() () (1) (2 3) ((4 foo) (5 (foo 6 foo)) 7 (8)))
```

Let's pretend that after thinking over the cases that our function must handle, we have written the program skeleton below. Here *A*, *B*, and *C* represent "holes," which are expressions we've yet to fill in.

```
(define remove
  (lambda (x ls)
    (cond
      [(null? ls) '()]
      [(equal? (car ls) x) (cons A B)]
      .
      C)))
```

We've already made a mistake! Though the program isn't complete, there is no way to fill in the holes to form a definition of *remove* that matches our tests. Sadly, test-driven development in a typical programming environment doesn't help us: we can't run tests on an incomplete program.

Challenge 5. How can we determine that we've already made a mistake?

Stop and think: How would you solve Challenge 5?

Using our relational interpreter we actually *can* run the test suite with our incomplete program, by representing any holes with (unquoted) logic variables.

```
(run 1 (A B C)
  (eval^o
    `(letrec ([remove (lambda (x ls)
                        (cond
                          [(null? ls) '()]
                          [(equal? (car ls) x) (cons ,A ,B)]
                          .
                          ,C)) ])
       (list (remove 'foo '())
             (remove 'foo '(foo))
             (remove 'foo '(1))
             (remove 'foo '(2 foo 3))
             (remove 'foo '((4 foo) foo (5 (foo 6 foo)) foo 7 foo (8)))))
     '(() () (1) (2 3) ((4 foo) (5 (foo 6 foo)) 7 (8))))))
```

⇒

```
()
```

Our query responds almost immediately with the empty answer set. Because miniKanren uses a complete search that is guaranteed to find an answer if one exists, the empty answer set means there is no correct way to synthesize expressions to fill in the holes. We've proved that an incomplete program is inconsistent with its test cases! We call such a proof a *refutation* of the program.

It's worth pausing for a moment to discuss how refutation works, and how it can be so fast. We mentioned that the underlying system runs a complete search, and it may be tempting to assume the search uses blind, brute force. But this search isn't blind. By working top-down, connecting expected result values to the outer expression shapes being considered, the search can prune unpromising branches very early on. It's only after the outer expression shape is shown to be compatible with the desired result that sub-expressions are considered.

Stop and think: We know our program is wrong. How do we figure out *where* it went wrong?

By attempting to satisfy tests individually[5], we can get a hint at the problem. In this case, the only unsatisfiable test is:

(*remove* 'foo '(foo)) ⇒ ()

This failing test suggests *remove* does not work correctly when the first list element is the item to be removed. If we relax a portion of our program by forming more or larger expression holes, looking for the moment our test becomes satisfiable, we can narrow down the problem. In this case, we are able to satisfy our query by relaxing the right-hand-side expression of our equality clause.

```
(run 1 (A B C)
  (eval^o
    `(letrec ([remove (lambda (x ls)
                        (cond
                          [(null? ls) '()]
                          [(equal? (car ls) x) ,A]
                          .
                          ,C)) ])
       (remove 'foo '(foo)))
    '()))
⇒
(((('() _0 _1)))
```

The test now passes, suggesting it was a mistake to try (cons ,*A* ,*B*) in the equality clause; the clause should produce the empty list, not a pair. With this insight, we can continue implementing *remove*.

Let's try the related problem of removing a symbol from anywhere in an s-expression, even inside deeply nested lists. Our test suite should return slightly different answers:

```
(list (remove 'foo '())
      (remove 'foo '(foo))
      (remove 'foo '(1))
      (remove 'foo '(2 foo 3))
      (remove 'foo '((4 foo) foo (5 (foo 6 foo)) foo 7 foo (8))))
⇒
(() () (1) (2 3) ((4) (5 (6)) 7 (8)))
```

[5]In general, all test subsets may need to be checked since a subset may fail even if all its tests are individually satisfiable.

Using *eval*^{o} to run our program as we write it, we notice a refutation:

```
(run 1 (A B)
  (eval^o
    `(letrec ([remove (lambda (x ls)
                        (cond
                          [(null? ls) '()]
                          [(equal? (car ls) x) ,A]
                          [else (cons (car ls) ,B)]))])
       (list (remove 'foo '())
             (remove 'foo '(foo))
             (remove 'foo '(1))
             (remove 'foo '(2 foo 3))
             (remove 'foo '((4 foo) foo (5 (foo 6 foo)) foo 7 foo (8)))))
     '(() () (1) (2 3) ((4) (5 (6)) 7 (8)))))
⇒
()
```

Checking each test individually, we learn that our last test case is unsatisfiable.

Relaxing expressions that contain logic variables suggests a problem with the right-hand-side of our else clause. Using (cons (car ls) ,*B*) causes the last test to fail, while (cons ,*C* ,*B*) does not.

Stop and think: What is wrong with this partial definition of *remove*?

The problematic right-hand-side (cons (car ls) ,*B*) always constructs a pair containing the *car* of *ls* as the first element, even if the *car* contains the symbol we wish to remove. In other words, there is never a recursive call on the *car* of *ls*. If we look at the list passed to *remove* in the failing test case, we notice that its first element is itself a list containing the item we'd like to remove. If the first element of *ls* is a list, we need to process it before including it in our result. Adding a [(pair? (car ls)) ...] clause after the [(null? ls) ...] clause allows our test to be satisfied.

The debugging steps in this section are tedious to use by hand, but they could provide rich interactive feedback if automatically executed by an interactive development environment. It is interesting that we're able to refute programs without needing the support of a type system (though it is possible to combine a type inferencer, written as a relation, with the interpreter).

7 A TINY THEOREM PROVER

Here's a tiny proof-checker for propositional logic, written in Racket:

```
(define proof?
  (lambda (proof)
    (match proof
      [`(,A ,assms assumption ())
       (member? A assms)]
      [`(,B ,assms modus-ponens (((,A ⇒ ,B) ,assms ,r1 ,ants1)
                                 (,A ,assms ,r2 ,ants2)))
       (and (proof? `((,A ⇒ ,B) ,assms ,r1 ,ants1))
            (proof? `(,A ,assms ,r2 ,ants2)))]
      [`((,A ⇒ ,B) ,assms conditional ((,B (,A . ,assms) ,rule ,ants)))
       (proof? `(,B (,A . ,assms) ,rule ,ants))])))
```

A proof is represented as a list (*conclusion assumptions rule-name sub-proofs*) where:

- *conclusion* is a miniKanren term;
- *assumptions* is a list of terms;
- *rule-name* is a symbol;
- *sub-proofs* is a list of zero or more proofs.

This proof-checker has only three rules: assumptions, such as the proposition A or the implication $A \Rightarrow B$; *modus ponens*, which derives B from A and $A \Rightarrow B$; and conditional proof, which derives $A \Rightarrow B$ if we can derive B after assuming A. For example, we can check this proof that C holds, assuming A, $A \Rightarrow B$, and $B \Rightarrow C$:

```
(proof? '(C (A (A ⇒ B) (B ⇒ C)) modus-ponens
            (((B ⇒ C) (A (A ⇒ B) (B ⇒ C)) assumption ())
             (B (A (A ⇒ B) (B ⇒ C)) modus-ponens
                (((A ⇒ B) (A (A ⇒ B) (B ⇒ C)) assumption ())
                 (A (A (A ⇒ B) (B ⇒ C)) assumption ()))))))
⇒
#t
```

Challenge 6. Write a tiny theorem prover for the logic of the proof checker, *proof?*.

Stop and think: How would you solve Challenge 6?

How might we write our tiny prover? A simple (though inefficient) algorithm might use a goal-directed, breadth-first search, which is complete over infinite trees. But miniKanren *already* implements a complete, goal directed search! What happens if we just run the proof checker in the relational interpreter we've used for the previous challenges?

Just as we treated the function *append* as a relation by running it inside the relational interpreter, we can treat the *proof?* function as a relation that connects assumptions, proofs, and conclusions. Here we query the *proof?* function to generate proofs of a theorem, given a set of assumptions:

```
(run 1 (prf)
  (fresh (body)
    (≡ prf `(C (A (A ⇒ B) (B ⇒ C)) . ,body))  ;; prove C holds, given A, A ⇒ B, B ⇒ C
    (eval^o
      `(letrec ([member? (lambda (x ls)
                           (cond
                             ((null? ls) #f)
                             ((equal? (car ls) x) #t)
                             (else (member? x (cdr ls)))))]
                [proof? (lambda (proof)
                          (match proof
                            [`(,A ,assms assumption ()) (member? A assms)]
                            [`(,B ,assms modus-ponens (((,A ⇒ ,B) ,assms ,r1 ,ants1)
                                                       (,A ,assms ,r2 ,ants2)))
                             (and (proof? `((,A ⇒ ,B) ,assms ,r1 ,ants1))
                                  (proof? `(,A ,assms ,r2 ,ants2)))]
                            [`((,A ⇒ ,B) ,assms conditional ((,B (,A . ,assms) ,rule ,ants)))
                             (proof? `(,B (,A . ,assms) ,rule ,ants))]))])
         (proof? ',prf))
       #t)))
```

```
⇒
(((C (A (A ⇒ B) (B ⇒ C)) modus-ponens
    (((B ⇒ C) (A (A ⇒ B) (B ⇒ C)) assumption ())
     (B (A (A ⇒ B) (B ⇒ C)) modus-ponens
        (((A ⇒ B) (A (A ⇒ B) (B ⇒ C)) assumption ())
         (A (A (A ⇒ B) (B ⇒ C)) assumption ()))))))))
```

Sure enough, miniKanren finds the proof we checked previously.

By instantiating different parts of the query, we can use *proof?* in different ways. We can partially instantiate the proof and ask the query to fill in the missing parts, or provide assumptions and ask the query to search for all valid conclusions with corresponding proofs. This method is powerful enough to prove interesting theorems, such as transitivity of implication:

```
(run 1 (prf)
  (fresh (body)
    ;; prove (A ⇒ B) ⇒ (B ⇒ C) ⇒ (A ⇒ C) holds absolutely
    (≡ prf `(((A ⇒ B) ⇒ ((B ⇒ C) ⇒ (A ⇒ C))) () . ,body))
    (eval^o
      `(letrec ([member? (lambda (x ls)
                           (cond
                             ((null? ls) #f)
                             ((equal? (car ls) x) #t)
                             (else (member? x (cdr ls)))))]
                [proof? (lambda (proof)
                          (match proof
                            [`(,A ,assms assumption ()) (member? A assms)]
                            [`(,B ,assms modus-ponens (((,A ⇒ ,B) ,assms ,r1 ,ants1)
                                                       (,A ,assms ,r2 ,ants2)))
                             (and (proof? `((,A ⇒ ,B) ,assms ,r1 ,ants1))
                                  (proof? `(,A ,assms ,r2 ,ants2)))]
                            [`((,A ⇒ ,B) ,assms conditional ((,B (,A . ,assms) ,rule ,ants)))
                             (proof? `(,B (,A . ,assms) ,rule ,ants))]))])
        (proof? ',prf))
      #t)))
⇒
(((((A ⇒ B) ⇒ ((B ⇒ C) ⇒ (A ⇒ C))) () conditional
   ((((B ⇒ C) ⇒ (A ⇒ C)) ((A ⇒ B)) conditional
     (((A ⇒ C) ((B ⇒ C) (A ⇒ B)) conditional
       ((C (A (B ⇒ C) (A ⇒ B)) modus-ponens
          (((B ⇒ C) (A (B ⇒ C) (A ⇒ B)) assumption ())
           (B (A (B ⇒ C) (A ⇒ B)) modus-ponens
              (((A ⇒ B) (A (B ⇒ C) (A ⇒ B)) assumption ())
               (A (A (B ⇒ C) (A ⇒ B)) assumption ()))))))))))))
```

We can also prove commutativity of conjunction where conjunction is encoded with implication.

8 QUINES, REVISITED

Challenge 7. Generate a quine that uses **quasiquote** instead of *list* or *cons*.

The quines using *list* and *cons* generated in section 3 are a little verbose. Surely a quine can be more concise! Perhaps a quine could use **quasiquote** to construct the expression. Sadly, our *eval*^{o} relational interpreter doesn't include the **quasiquote** form. We could extend the interpreter to add it, but is there another way?

Stop and think: How would you solve Challenge 7 without modifying the relational interpreter?

Unlike *append*, **quasiquote** can't be implemented as a function, since it does not directly evaluate its argument, but rather interprets its argument to find **unquote** forms. Thus we can't simply define **quasiquote** in the interpreter with **letrec**. We need to go more meta! We'll use a meta-circular evaluator running *inside* the relational interpreter.

Let's write an interpreter for a small subset of Racket that includes **quasiquote**, in the subset of Racket supported by the relational interpreter. Our query asserts that applying the nested interpreter to the quoted s-expression of the quine should return that same s-expression.[6]

```
(run 1 (q)
  (eval^o
    `(letrec ([eval-quasi (lambda (q eval)
                            (match q
                              [(? symbol? x) x]
                              [`() '()]
                              [`(,"`unquote ,exp) (eval exp)]
                              [`(quasiquote ,datum) ('error)]
                              [`(,a . ,d) (cons (eval-quasi a eval) (eval-quasi d eval))]))])
       (letrec ([eval-expr (lambda (expr env)
                             (match expr
                               [`(quote ,datum) datum]
                               [`(lambda (,(? symbol? x)) ,body)
                                (lambda (a)
                                  (eval-expr body (lambda (y)
                                                    (if (equal? x y)
                                                        a
                                                        (env y)))))]
                               [(? symbol? x) (env x)]
                               [`(quasiquote ,datum)
                                (eval-quasi
                                  datum
                                  (lambda (exp) (eval-expr exp env)))]
                               [`(,rator ,rand)
                                ((eval-expr rator env) (eval-expr rand env))]))])
         (eval-expr ',q 'initial-env)))
    q))
```

[6]The unfortunate representation of the pattern for matching unquote expressions results from an oddity of Racket's quasiquotation support. In other Scheme implementations that line can be written [`(,`unquote ,exp) (eval exp)].

To keep the meta-circular evaluator concise, we don't implement nested quasiquotation. Instead, whenever the meta-circular evaluator encounters a nested **quasiquote** it produces an error by illegally attempting to call a symbol. The outer, relational interpreter handles such an error as failure, so the combination of interpreters won't generate programs using nested quasiquotation. We use the same technique to fail upon variable lookup in the empty environment, which we represent with the symbol initial-env.

A nested **letrec** is necessary because our standard version of *eval*^{o} does not support mutually-recursive definitions, though the optimized *eval*^{o} used earlier does support mutual recursion.

Insight: The outer, relational interpreter must use a first-order representation of environments and closures, as miniKanren only supports first-order terms. However, our Racket-in-Racket can use higher order representations thanks to its meta-circular use of that outer implementation. Environments are represented as functions that accept a symbol and return the value to which it is bound. When extending an environment, we wrap it in a new function that first checks a variable against the new binding. If the variable doesn't match, the new function delegates to the original environment by calling the function representing it.

The meta-circular evaluator uses pattern matching to recognize each form of expression. In the relational setting it's even more important than usual to specify programs unambiguously. Consider what might happen if rather than using *match*, we were to use this conditional expression to recognize the syntax for a **lambda** expression:

```
;; pattern used with match
`(lambda (,(? symbol? x)) ,body)

;; solution with conditional expression
(and (equal? 'lambda (car expr))
     (pair? (cadr expr))
     (symbol? (caaar expr))
     (pair? (cddr expr)))
```

When evaluating an instantiated expression with well-formed syntax, this condition should work. But when generating expressions, miniKanren can invent any expression that satisfies the condition. It might generate the identity function like this:

```
(lambda (x . _0) x . _1)
```

where $\__0$ and $\__1$ indicate that any term whatsoever may appear in those positions. Although the values in those positions don't affect the behavior of the expression in this evaluator, they both have to be the empty list in order for this function to behave the way we expect in Racket.

To ensure the expression assumes the desired syntax, the condition would need two more checks to fully specify the form: (*null?* (*cdadr expr*)) and (*null?* (*cdddr expr*)). In contrast, the natural way of writing the evaluator with *match* fully specifies the format.

Our **quasiquote** quine-generating query returns the answer:

```
((((lambda (_0) `(,_0 ',_0))
   '(lambda (_0) `(,_0 ',_0)))
  (≠ ((_0 closure)) ((_0 prim))) (sym _0)))
```

Finding quines through the meta-circular evaluator is much slower than the approach we used in section 3—this query takes 5 minutes with our current hardware—but it allows us to generate programs using a new language form without modifying our relational interpreter.

9 CONCLUSION

While solving these programming challenges we've shown that a relational interpreter can be used to solve myriad problems: example-based synthesis of recursive programs; refutation and bug-finding for incomplete programs; and generating program inputs with desired properties, using both a single interpreter and multiple interpreters with different semantics. Our success in solving the most difficult of these challenges—such as synthesis of recursive programs—is due to optimizations that use convenient properties of purely functional programming languages. The freedom to re-order evaluation, thanks to lack of side effects, is particularly important.

We also found that relational interpreters can confer relational capabilities on higher-order functional languages, even through multiple levels of interpretation. This pearl is a concrete demonstration that a functional program can have more (and surprising!) behaviors than just that for which it was originally written. In some sense every experienced functional programmer knows this to be true—for example, we write semantics as relations. The relational interpreter approach provides a technique for exercising the fully general relational meaning of a functional program.

ACKNOWLEDGMENTS

This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Stuart Halloway pointed out that the relational interpreter should be able to generate quines; thank you, Stuart! We thank Larry Moss and the members of the Indiana University logic seminar for their encouragement, and for challenging us to generate twines and thrines. Eric Holk and Dan Friedman worked with co-author Byrd on quine generation in miniKanren, which led to a paper [Byrd et al. 2012] from which Challenge 2 was taken.

We thank Dan Friedman, Nada Amin, Jason Hemann, Rob Zinkov, Tom Gilray, Lisa Zhang, Jon Smock, Dann Toliver, Annie Cherkaev, Guannan Wei, Dmitri Boulytchev, Evan Donahue, Steve Gilardi, and the anonymous reviewers for their many helpful comments.

REFERENCES

William E. Byrd and Daniel P. Friedman. 2006. From Variadic Functions to Variadic Relations: A miniKanren Perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop (University of Chicago Technical Report TR-2006-06)*, Robby Findler (Ed.). 105–117.

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

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

Douglas R. Hofstadter. 1979. *Gödel, Escher, Bach : an Eternal Golden Braid*. Basic, New York, NY, USA.

J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 111–119. https://doi.org/10.1145/41625.41635

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 192–203. https://doi.org/10.1145/1086365.1086390

Stephen Cole Kleene. 1952. *Introduction to Metamathematics*. North-Holland, Amsterdam.

John McCarthy. 1978. A micro-manual for LISP – Not the whole truth. *SIGPLAN Not.* 13, 8 (Aug. 1978), 215–216. https://doi.org/10.1145/960118.808386

Matt Might. 2013. 99 ways to say '(I love you) in Racket. http://matt.might.net/articles/i-love-you-in-racket/. (February 2013). Accessed: 2017-02-28.