Some Novel miniKanren Synthesis Tasks

JASON HEMANN, Northeastern University
DANIEL P FRIEDMAN, Indiana University

Synthesizing quines with a relational interpreter in miniKanren is a relatively simple but not infrequently arresting example. We exhibit several related, novel synthesis tasks—either pre-existing programming challenges we newly solve with miniKanren synthesis, or newly formulated challenges of our own. In doing so we exhibit a relational interpreter for a "mirrored" language. These examples demonstrate miniKanren's potential and versatility as a platform or substrate for experimenting on executable program specifications. As with quines, these examples are themselves interesting from a theoretical perspective, may also suggest other, more practical future applications.

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

Additional Key Words and Phrases: relational, program synthesis, quine, miniKanren, logic programming

ACM Reference Format:
Jason Hemann and Daniel P Friedman. 2020. Some Novel miniKanren Synthesis Tasks. 1, 1 (July 2020), 14 pages.

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

John McCarthy, *A micro-manual for LISP-not the whole truth*

1 INTRODUCTION

Constraint-logic and relational programming languages such as miniKanren have found application in program synthesis tasks [7]. Although Kanren languages are also applicable to many of the standard logic programming tasks, such synthesis applications have arguably become Kanren languages' foremost specialty.

Generating *quines* is an early, not-infrequently arresting example of relational programming and of relational program-synthesis in miniKanren [8]. In his book *Gödel, Escher, Bach*, Hofstadter coined "quining" to mean "the operation of preceding some phrase by its quotation". This self-reference guarantees a quine—a program that evaluates to its own source code—in any sufficient programming language (e.g., the language of Listing 1). The existence of quines follows from Kleene's second recursion theorem (see e.g., Moschovakis [22] for details). Lee [16] and Thatcher [29] exhibit early examples of Turing machines that print their own descriptions. Bilar and Filiol [3] credit Hamisch Dewar with producing the earliest known self-replicating program, written in Atlas Autocode at the University of Edinburgh in the 1960s. The traditional, textbook proofs of quines' existence (e.g., Yasuhara [31, p. 102]) uses Gödel encodings of Turing machines, and is a bit afield from the programmer who wants to experiment.

Authors' addresses: Jason Hemann, jhemann@northeastern.edu, Northeastern University; Daniel P Friedman, dfried@indiana.edu, Indiana University.

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/

```
> (run 1 (q) (evalo q q))
'((((λ (_0) (list _0 (list 'quote _0)))
   '(λ (_0) (list _0 (list 'quote _0))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Listing 1. Querying against a relational interpreter for a quine

This same demonstration is more straightforwardly expressed in miniKanren with the query in Listing 1. We remind the reader that when miniKanren constrains the synthesized programs' variables, these constraints are most often to exclude distinguished tags on defunctionalized closures, to prevent shadowing special forms in the environment, or to restrict a logic variable to a sub-sort of the domain (e.g., the symbols). The language of Listing 1's interpreter, `evalo`, can produce only *proper quines*. Briefly, a proper quine has distinguishable and separable data and code components (disqualifying blank files or self-evaluating literals) and that does not reflect on its source code, e.g., through the file-system.[1] No interpreter in this paper can produce an improper quine, and so we use "quine" as a shorthand without ambiguity.

miniKanren can produce still more complicated examples with similar techniques. Byrd et al. [8] show how to implement "twines" (twin-quines) and "thrines" as well. The relational interpreter makes producing examples of such programs trivial with similar, slightly more sophisticated queries. Pagiamtzis [25] defines an *oscillating quine with initial transients*, by analogy to electrical engineering, as a sequence of distinct programs $p_0, \ldots, p_{k-1}, p_k, \ldots, p_n$, where each p_i evaluates to p_{i+1}, and where p_n evaluates to p_k. More precisely, this sequence of programs forms an $n-k$-quine cycle with a $k-1$ initial transient. Listing 2 demonstrates the query for such a program. The auxiliary `all-diffo` relation imposes disequality constraints between every two programs p_i and p_j, $i \neq j$ in that list. From left to right, the first two evaluations in the query describe the quine cycle, and the remaining three evaluations describe the antecedent transient programs, in reverse order. miniKanren uses an interleaving depth-first search (see Rozplokhas et al. [27]) biased toward earlier successful branches of the search. In this order, miniKanren spends more of its search time looking for a twine, and with a twine in hand miniKanren can quickly synthesize the antecedent programs.

Self-replicating programs do make occasional appearances in applied research, especially in security [30]. Writing quines, however, is usually a hacker's pastime (or even art [5]). Generating quines is not in and of itself an especially practical example of program synthesis. Byrd et al. [7] demonstrate that generating quines can be a first step toward more complex, useful synthesis tasks. We find this paper's novel miniKanren synthesis tasks interesting for similar reasons. To more practically-oriented readers we suggest that—like the earlier quine generation results—even if the programs we synthesize here are not themselves practically interesting, they may suggest related programs with practical applications and future uses of similar techniques to construct such related practical programs. Some of these tasks (e.g., Section 6) are novel in that the authors have found no earlier references that address the questions we raise and have not seen their specifications beyond our own work. We find intrinsically motivating how relational synthesis techniques suggest simple, novel, and (subjectively) interesting program specifications. Our other synthesis examples' specifications are not in and of themselves novel, but usually presentations offer these tasks as coding challenges for programmers; the authors are unaware of previous miniKanren syntheses of such programs. These examples also explore the space of programs and demonstrate how to experiment with a relational interpreter.

[1] See Madore [18] or the Quine contest rules at codegolf.meta.stackexchange.com/questions/4877 for more.

```
> (run 1 (v t u q p)
    (all-diffo '(v t u q p))
    (evalo q p) (evalo p q) (evalo u q) (evalo t u) (evalo v t))
'(((''''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
        '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   '''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
       '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   ''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
      '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   '((λ (_0) (list 'quote (list _0 (list 'quote _0))))
     '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   ((λ (_0) (list 'quote (list _0 (list 'quote _0))))
    '(λ (_0) (list 'quote (list _0 (list 'quote _0))))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Listing 2. Successful query for a binary iterating quine with a three-step transient.

```
(define-relation (tsil-reporpo pxe vars env lav)
  (conde
    [(== `(tsil) pxe)
     (== `() lav)]
    [(fresh (a d t-a t-d)
       (== `(,a . ,d) pxe)
       (== `(,t-a . ,t-d) lav)
       (fo-lavo a vars env t-a)
       (tsil-reporpo d vars env t-d))]))
```

Listing 3. The `tsil-reporpo` help relation for the `fo-lavo` of Listing 4

In Section 2 we introduce a version of Byrd et al.'s [8] relational interpreter with a reversed syntax and explain its implementation's unique bits. Then in Section 3 we remind the reader of polyglots and demonstrate a successful query for valid structurally-palindromic programs. In Section 4, we describe a technique for generating quine-relays. In Section 5, we synthesize programs that bootstrap quines. In Section 6, we specify a kind of quasi-quine cycles, exhibit both trivial and non-trivial programs that meet the specification and generalize this result to broader classes of such programs. In Section 7 we describe related synthesis work, and in Section 8 we conclude. The examples of Sections 2, 3 and 6 were previously published in Hemann [13, chap. 4]. At github.com/jasonhemann/novel-miniKanren-synthesis-tasks we provide the source code for our examples as well as instructions to execute it. In places we have slightly reformatted the output for spacing and clarity.

2 MIRRORED LANGUAGE INTERPRETER

Many of the following synthesis problems involve executing programs across multiple languages. To that end, we implement a relational interpreter like that of Byrd et al., but with the language's syntax reversed. That is, a program `((λ (arg) (list arg)) (quote cat))` would be `((cat etouq) ((arg tsil) (arg) adbmal))` in this reversed language. The two languages have the same semantics, only the concrete syntax has changed.

```
(define-relation (laveo pxe lav)
  (fo-lavo pxe '() '() lav))

(define-relation (fo-lavo pxe vars env lav)
  (conde
    [(fresh (v)
       (== `(,v etouq) pxe)
       (absento 'etouq vars)
       (absento 'closure v)
       (== v lav))]
    [(absento 'tsil vars)
     (absento 'closure pxe)
     (tsil-reporpo pxe vars env lav)]
    [(symbolo pxe) (lookupo pxe vars env lav)]
    [(fresh (rotar dnar x ydob vars^ env^ a)
       (== `(,dnar ,rotar) pxe)
       (fo-lavo rotar vars env `(closure ,x ,ydob ,vars^ ,env^))
       (fo-lavo dnar vars env a)
       (fo-lavo ydob `(,x . ,vars^) `(,a . ,env^) lav))]
    [(fresh (x ydob)
       (== `(,ydob (,x) adbmal) pxe)
       (== `(closure ,x ,ydob ,vars ,env) lav)
       (symbolo x)
       (absento 'adbmal env))]))
```

Listing 4. The `fo-lavo` evaluation relation with a split environment and the `laveo` relation front end

Listing 4 contains the mirrored language interpreter. While the syntax of the language is mirrored, the internal representations of closures and environments are not. Naturally enough, we named this relation `laveo`, which calls to `fo-lavo`. The one interesting difference is that testing for `tsil` expressions now requires a help relation. Unification can immediately separate the first element of a list from the rest, however, matching against the last element of a list requires arbitrary recursion. To this end we implemented the help relation `tsil-reporpo` in Listing 3, a mirrored "proper list" relation that describes reversed proper lists.

That the two languages have the same semantics (they are isomorphic) is inconsequential in the following. It is important, however, that we have relational interpreters for two languages with overlapping syntaxes. More precisely still, for these results we need programs in the language of one interpreter to produce, as data, certain classes of programs in the language of the second interpreter.

3 PALINDROME PROGRAMS

Polyglots are programs valid in more than one language. In Listing 5, we query for programs that are well-formed in both the language of the standard relational interpreter and also Listing 4's interpreter's mirrored language. We query for a single program `q` that evaluates in both interpreters. We do not require that the two values be equal; the problem statement only demands that the programs have a value under each interpretation.

```
> (run 3 (q) (fresh (a b) (evalo q a) (laveo q b)))
'('etouq
  ((λ (_0) adbmal) (sym _0))
  ((λ (adbmal) adbmal) (λ (λ) adbmal)))
```

Listing 5. "Mirrored programs" that evaluate in both languages

```
> (run 1 (q) (fresh (p) (laveo p q) (evalo q p)))
'((((λ (_0) (list (list _0 (list 'quote _0)) 'etouq))
    '(λ (_0) (list (list _0 (list 'quote _0)) 'etouq)))
   (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
   (sym _0)))
```

Listing 6. A relational query for the first entry of a 2-cycle quine relay

For the special case of these two relational interpreters, whose languages are mirror images of each other, these polyglot programs are also palindromes. The programs are not literally, character for character palindromes. Instead, they are palindromic in that their syntax trees are mirror images. Racket's default printer slightly obscures this, since it uses abbreviations for **quote** and **lambda**. We could, however, overcome this imbalance by installing a printer that either removes these abbreviations or provides analogous abbreviations for their mirrored counterparts.

4 QUINE RELAYS

Quine relays (or "ouroboros programs") are n-cycles of programs where each program p_i in the cycle evaluates to the program $p_{i+1} \pmod{n}$. To preclude trivial quine relays, the problem statement usually requires a cycle of *distinct* programs, and further that each program be written in a different programming language.

In Listing 6 we query miniKanren using the same two relational interpreters to synthesize a program `p` that in the language of the first interpreter evaluates to a program `q` in the language of the second interpreter, which itself then evaluates to `p`. The calls to the two different interpreters guarantee that the resulting pairs of programs will be in different languages; we do not have to overtly stipulate that the languages be different or write predicates to recognize each language's programs. If we had instead merged the two evaluation relations into one, we would then need to recursively specify the programs of each language to properly state our query. Further, because the only overlap in the two interpreters' languages are quoted (`etouq`'d) expressions, we do not need to explicitly stipulate that the programs be different with a disequality constraint.

miniKanren's first solution is not especially revealing. Rather than the self-quoting behavior often seen in miniKanren twines (e.g., Listing 2), miniKanren has generated a self-`etouq`ing expression. The quine cycles example does demonstrate an interesting facet of miniKanren's behavior. In queries for twines, the order of the calls to the interpreter make little to no difference, since both calls are to the same interpreter. For quine cycles though, reordering the calls can make a significant difference. miniKanren takes significantly longer to answer Listing 7's version of this query. Further, the answer largely reverses the roles of the two interpreters. The query's result is a self-quoting (that is, with the real **quote** of the standard interpreter) program, largely written in the language of the mirrored interpreter of Section 2. The result program is similar to, but not precisely, a mirrored version of the more traditional-looking result program from Listing 6. Reversing the order of the two

```
> (run 1 (q) (fresh (p) (evalo q p) (laveo p q)))
'(('((('etouq
      (_0
       (((_0 (etouq etouq) tsil) (_1 etouq) tsil)
        ((_1) etouq)
        (adbmal etouq)
        tsil)
       tsil)
      tsil)
     (_0)
     adbmal)
    ((((('etouq
         (_0
          (((_0 (etouq etouq) tsil) (_1 etouq) tsil)
           ((_1) etouq)
           (adbmal etouq)
           tsil)
          tsil)
         tsil)
        (_0)
        adbmal)
       etouq)
      _1)
     (_1)
     adbmal))
  (=/= ((_0 closure)) ((_0 etouq)) ((_0 tsil)) ((_1 closure)) ((_1 etouq)))
  (sym _0 _1)))
```

Listing 7. A second relational query for a 2-cycle quine relay, reordering the evaluations

evaluations biases miniKanren's search toward the already-partially successful branches of quoted programs when evaluating in `laveo`. `tsil` expressions cost more to evaluate than `list` expressions, and we believe this difference explains some of the overall performance disparity.

5 BOOTSTRAPPING, RECOVERING QUINES

David Madore, author of the unlambda language, on his *Quines* page [18] inadvertently describes yet another application of synthesis by relational interpreter. Madore describes how, by executing a near-quine program whose "source code" portion is damaged, mangled, or otherwise modified, we might recover the original quine. Some quines are then, to a degree, self-healing programs: for certain damage, executing the damaged program *bootstraps* the quine, and recovers the original, undamaged source. In Listings 8 to 11 we query for three such near-quine "damaged-code" programs that, when invoked, produce a quine. The query stipulates via `(absento p s)`, an additional constraint, that we will exclude somewhat trivial cases like nullary damage (we can view quines as trivially, vacuously bootstrapping themselves) or programs simply containing the quine itself as data. Without these restrictions, in fact, the bootstrap programs are precisely the initial transients of oscillating 1-cycle quines with a single initial transient.

```
> (run 3 (q)
    (fresh (p r s)
      (== q `(,s ,r))
      (absento p s)
      (evalo `(,p ,r) `(,p ,r))
      (evalo `(,s ,r) `(,p ,r))))
```

Listing 8. Query bootstrapping quines from “damaged” source code.

```
'((((λ (_0) (list _0 (list 'quote _0)))
    '(λ (_1) (list _1 (list 'quote _1))))
   (=/= ((_0 _1)) ((_0 closure)) ((_0 list)) ((_0 quote))
        ((_1 closure)) ((_1 list)) ((_1 quote)))
   (sym _0 _1))
```

Listing 9. First result for the query in Listing 8

```
((((λ (_0) (λ (_1) (list _1 (list 'quote _1)))) '_2)
  '(λ (_3) (list _3 (list 'quote _3))))
 (=/= ((_0 list)) ((_0 quote)) ((_0 λ))
      ((_1 _3)) ((_1 closure)) ((_1 list)) ((_1 quote))
      ((_3 closure)) ((_3 list)) ((_3 quote)))
 (sym _0 _1 _3)
 (absento ((λ (_3) (list _3 (list 'quote _3))) _2) (closure _2)))
```

Listing 10. Second result for the query in Listing 8

We break up the list of results over Listings 9 to 11. The first program miniKanren returns is α-equivalent to a quine. The bootstrap program differs from the quine it produces only in the names of code portion's lambda expressions' variables. miniKanren constrains the variables `_0` and `_1` in the answer with a disequality, maintaining the query's requirement that `p` is absent from `s`. These subsequent results are also interesting.

For this second result, two constraints on the answer bear particular mention. First, _1 must not equal _3, since by the query's `absento` constraint the resulting quine's operator, `(λ (_3) (list _3 (list 'quote _3)))`, must not be present in the operator of the bootstrap program. That same `absento` induces the restriction on _2 in the presented answer.

We choose to include Listing 11 and the third result because it showcases a good example of miniKanren generating a non-trivial bootstrap program. This bootstrap program, when executed, constructs the subsequent quine's operator **`lambda`** expression from quoted pieces. Taken together, these results may appear to the reader as “strategically” damaged source code, reverse-engineered. Bootstrapping from damaged quine code may feel more realistic in a lower-level language closer to the machine's representation. The specification we describe via our queries, however, would not change, and from miniKanren's complete search we know every answer occurs at some point in the sequence of results.

```
(((λ (_0)
    (list (list 'λ '(_1) '(list _1 (list 'quote _1))) (list 'quote _0)))
  '(λ (_1) (list _1 (list 'quote _1))))
 (=/= ((_0 closure)) ((_0 list)) ((_0 quote))
      ((_1 closure)) ((_1 list)) ((_1 quote)))
 (sym _0 _1)))
```

Listing 11. Third result for the query in Listing 8

```
> (run 2 (q) (evalo `(,q ,q) q))
```

Listing 12. Query for trivial and non-trivial quasi-quines q

```
'(quote
  (((λ (_0) (λ (_1) (list _0 (list 'quote _0))))
    '(λ (_0) (λ (_1) (list _0 (list 'quote _0)))))
   (=/= ((_0 _1)) ((_0 closure)) ((_0 list)) ((_0 quote)) ((_0 λ))
        ((_1 closure)) ((_1 list)) ((_1 quote)))
   (sym _0 _1)))
```

Listing 13. Query for non-trivial quasi-quines q

6 QUASI-QUINES AND QUASI-QUINE CYCLES

The query in Listing 12 asks not for quines, but instead a sub-expression `q` that, when applied to itself, returns itself as the result. This question arose while we were experimenting with a relational interpreter. We invite the reader to stop a moment and consider—does something like that exist? What would one look like? Absent the relational interpreter approach, how would you construct one?

miniKanren quickly finds a first answer: **`quote`**. In the language of the interpreter, as in Scheme, `(quote quote)` evaluates to **`quote`**. The first result of this query is trivially satisfiable. We present the first and second answers in Listing 13, where for readability we expand the first **`quote`** result from Racket's printer. The second answer is non-trivial, and looks closer to the usual quine answers, but parameterized additionally in both code and data by a "don't care" variable. The disequality constraint between `_0` and `_1` differs from either of the two superficially similar examples we discussed in Listings 9 and 10. Here, the disequality constraint is necessary to prevent shadowing, lest the "don't care" variable actually matter to the result.

Now suppose we wish to construct a triplet of distinct expressions `a`, `b`, and `c` so that the expressions `(a b)`, `(b c)`, and `(c a)` evaluate respectively to `c`, `a`, and `b`. Such programs are not the thrines discussed in Section 1. As far as we know, there is no canonical name for the program cycles with this property; we've nicknamed them "quasi-quine cycles", or "stuttered-quine cycles". We find even our clearest prose specifications for such programs wordier and more opaque than simply reading this program triplet's specification directly from the miniKanren query itself.

Listings 14 and 15 contain this augmented query and the first program triplet that miniKanren returns. The answers, again, are non-trivial. The authors were not certain what to expect when initially formulating these queries. We had not predicted the trivial answers. After first discovering the trivial results, we still were not

```
(run 1 (a b c)
  (fresh (t)
    (== t `((,a ,b) (,b ,c) (,c ,a)))
    (=/= a b) (=/= b c) (=/= c a)
    (evalo `(,a ,b) c)
    (evalo `(,b ,c) a)
    (evalo `(,c ,a) b)))
```

Listing 14. Query for the triplet of expressions a, b, c that form a stuttered quine 3-cycle

```
'((((λ (_0)
      '(λ (_1)
         (list
          'λ
          '(_2)
          (list
           'quote
           (list 'λ '(_0) (list 'quote (_1 '_3)))))))
    (λ (_2)
      '(λ (_0)
         '(λ (_1)
            (list
             'λ
             '(_2)
             (list
              'quote
              (list 'λ '(_0) (list 'quote (_1 '_3))))))))
    (λ (_1)
      (list
       'λ
       '(_2)
       (list 'quote (list 'λ '(_0) (list 'quote (_1 '_3)))))))
   (=/= ((_0 closure)) ((_0 quote)) ((_1 closure)) ((_1 list))
        ((_1 quote)) ((_2 closure)) ((_2 quote)))
   (sym _0 _1 _2)
   (absento (closure _3))))
```

Listing 15. Result of query in Listing 14

sure that miniKanren could produce more interesting answers. One of the facets that makes miniKanren a worthwhile synthesis platform is the ease with which we can formulate specifications as executable miniKanren queries and run experiments.

7 RELATED QUINE-LIKE SPECIFICATIONS AND MINIKANREN SYNTHESIS TASKS

Self-reference is a broad, general phenomenon well-studied across many different contexts [28]. While of wide theoretical interest, the contortions required to express self-reference in many systems make it generally unusable as a practical technique. Self-reference is significantly easier to express, though, in systems with quotation. Little [17] and Polonsky [26] formalize the properties of quotation and syntactic self-reference and suggests directions for constructing languages with facilities for practically useful self-reference. Even just defining a class of non-trivial self-replicating systems has proved somewhat difficult, and recent efforts point to a non-binary sliding scale [1]. Quines are generally regarded among the more trivial kinds of self-replication; the examples we present here are of similar complexity.

Myhill [23] describes the general theory of self-reproduction, and mentions producing identical offspring as the simplest variety. His discussion includes varieties of automata that take no input, and also posits infinitely-descending chains of automata-generating automata. Myhill also suggests self-improving automata, and even automata that are the mirror-image of their parent. Case [9] studies machine lineages and the problem of deciding if a particular machine starts an infinite sequence, and if so, whether that sequence is periodic. "Periodic chains" is another description for those bootstrapped n-cycles. Kraus's [15] recently rediscovered MS thesis is something of a bridge between this earlier work and the application to computer virology. In it he also describes a wide variety of self-replicating adjacent programs, several of which we have synthesized here. Where quines do appear in security and computer virology, it's frequently in some deep and interesting places, such as in Thompson's [30] "Reflections on Trusting Trust." Gratzer and Naccache [12] offer a more recent example, showing how quines can help secure software systems, rather than undermine them.

Program synthesis is a broad research area with interdisciplinary connections to programming languages, machine learning, artificial intelligence and regular conferences in its own right. Unsurprisingly it also has deep connections to logic and specification. In its early form, synthesis research dates back to Church [10] in 1957. Manna and Waldinger [19] developed an influential model for synthesizing from a logical statement of the specification programs guaranteed-correct with respect to that specification. More recently, the *syntax-guided synthesis* [2] approach has gained traction. In syntax-guided synthesis, the user also provides partial template(s) as additional synthesis aids. Kanren-style relational interpreters take advantage of partial program templates. Their problem space, however, is much more general than the relatively straightforward synthesis problems we address here.

Byrd et al. [7] solve a problem nearly the inverse of polyglots. Instead, they evaluate the same concrete syntax under distinct semantics, in their case lexical and dynamic scope. These are orthogonal directions, and we could certainly mix and match distinct semantics and different (but not disjoint) syntaxes for even more interesting queries.

8 CONCLUSIONS

We have presented several instances of more general classes of well-known problems, sometimes presented as challenges or contests. Through surveying these myriad exemplary problems and their uniform solutions, we have demonstrated the power, simplicity, and convenience of miniKanren as a language for solving these quine and quine-like synthesis problems. We have used relational programming techniques to synthesize curious programs across multiple languages, including polyglot "palindrome programs", quine relays, and some stuttered quine-like programs. Our straightforward, declarative solutions to instances of so many diverse pre-existing problems should evidence miniKanren expressiveness. That these aren't artificially created problems, but largely known and pre-existing challenges further buttresses this claim.

These problems can play the role of "litmus tests" to assess, express, and checkpoint the power of relational language design and implementation. We think it also important that we consider here concrete synthesis problems in a broader context of computability theory. Distillation research [24] is itself a meritorious aim, and to the degree we fulfill this role, that too is a research contribution.

We cannot claim the utmost immediate application. This is not white paper research driven to, say, show improved load balancing on a particular vendor's cloud computing infrastructure. It is little surprise our work less immediately, practically applicable than other research motivated by an intended practical application. We reflect that applications often present themselves where foundational research exists to be applied. The world would be significantly impoverished had the last 2,000 years progress of science and mathematics been limited to that which could be put to immediate use.

Although not as well known as quines themselves, producing polyglot programs and quine relays are also time-worn coding challenges. Challenges like these are usually solved by hand. Other stipulations are sometimes added in such contests for extra complexity, e.g., "the shortest such program that … ", "the longest cycle of such programs that … ". Margenstern and Rogozhin [20], for instance, describe significantly shrinking self-describing Turing machines.

Multi-quine synthesis would be an interesting challenge for the future. A "multi-quine" is a set of programs that print their own source code, except that on particular, given inputs, they instead produce another program in the set. Madore describes virology applications for these results, and even demonstrates an impressive encrypted multi-quine proof-of-concept for this behavior. These would require at least some portions of Byrd et al.'s more powerful interpreters, as by definition multi-quines requires variable-arity functions. Presumably this would also necessitate their techniques for taming the explosion of the search space. Along these same lines we suggest self-analyzing or self-describing programs as other synthesis targets. These programs are not *necessarily* self-referential, since, of course, there are many other programs that begin with the same character, have the same length, or what have you. Nor are they strictly speaking quines, since many varieties of these problems require input. They too might suggest other future applications.

The answers we synthesized are simple compared with record-holding solutions. The codegolf community is a repository for such challenges and impressive solutions. Endoh [11] constructed a 128-language quine relay. Also impressive are long-lasting group efforts producing 280+-language polyglot programs.[2] The authors do not anticipate dethroning these records with our techniques anytime soon. We remind the reader, however, that these records are not mechanically synthesized in seconds or minutes, but instead laboriously hand-crafted; in the latter case constructed communally by scores over years. Bratley and Millo [6] in 1972 anticipate the enduring fascination for such exercises, explaining

> there is something about this particular problem which excites keen programmers, so that research in self-reproducing program design is now under way for practically every compiler on our system.

We too could not resist the temptation to play.

[2]See the current record and discussion at codegolf.stackexchange.com/questions/102370.

```
(define-relation
  (plop acc whole res data)
  (conde
   ((fresh
     (s)
     (== data `(,s))
     (== acc `(,s))
     (conde
      ((fresh
        (a d)
        (== s `(,a unquote d))
        (=/= d '())
        (fresh (m) (== res `(,m)) (plop s whole m s))))
      ((== 'q s) (== res `(q ',whole))))))
   ((fresh
     (a d)
     (=/= d '())
     (== `(,a unquote d) data)
     (fresh
      (p)
      (== res `(,a unquote p))
      (fresh (q) (== acc `(,a unquote q)) (plop q whole p d)))))))
 (run*
  (q)
  (fresh
   (x)
   (plop
    x
    x
    q
    '((define-relation
        (plop acc whole res data)
        (conde
         ((fresh
           (s)
           (== data `(,s))
           (== acc `(,s))
           (conde
            ((fresh
              (a d)
              (== s `(,a unquote d))
              (=/= d '())
              (fresh (m) (== res `(,m)) (plop s whole m s))))
            ((== 'q s) (== res `(q ',whole))))))
         ((fresh
           (a d)
           (=/= d '())
           (== `(,a unquote d) data)
           (fresh
            (p)
            (== res `(,a unquote p))
            (fresh (q) (== acc `(,a unquote q)) (plop q whole p d)))))))
      (run* (q) (fresh (x) (plop x x q)))))))
```

REFERENCES

[1] Bryant Adams and Hod Lipson. 2003. A universal framework for self-replication. In *Advances in Artificial Life, 7th European Conference, ECAL 2003, Dortmund, Germany, September 14-17, 2003, Proceedings* (Lecture Notes in Computer Science). Wolfgang Banzhaf et al., (Eds.) Vol. 2801. Springer, 1–9. DOI: 10.1007/978-3-540-39432-7_1. https://doi.org/10.1007/978-3-540-39432-7_1.

[2] Rajeev Alur et al. 2013. Syntax-guided synthesis. In *Proceedings of Formal Methods in Computer Aided Design, FMCAD 2013.* Barbara Jobstmann and Sandip Ray, (Eds.) IEEE, 1–17.

[3] Daniel Bilar and Eric Filiol. 2009. Editors/translators foreword. Daniel Bilar and Eric Filiol, (Eds.) (2009). DOI: 10.1007/s11416-008-0116-y. link.springer.com/journal/11416/5/1.

[4] Daniel Bilar and Eric Filiol, (Eds.) 2009. Journal in Computer Virology 5, 1 (2009): *On Self-Reproducing Computer Programs.* link.springer.com/journal/11416/5/1.

[5] Gregory W. Bond. 2005. Software as art. *Commun. ACM*, 48, 8, (08/2005), 118–124. DOI: 10.1145/1076211.1076215. https://doi.org/10.1145/1076211.1076215.

[6] Paul Bratley and Jean Millo. 1972. Computer recreations: self-reproducing programs. *Software: Practice and Experience*, 2, 4, 397–400. DOI: 10.1002/spe.4380020411. onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380020411.

[7] 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, (08/2017), 8:1–8:26. DOI: 10.1145/3110252. http://doi.acm.org/10.1145/3110252.

[8] 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, Copenhagen, Denmark, 8–29. DOI: 10.1145/2661103.2661105. http://doi.acm.org/10.1145/2661103.2661105.

[9] John Case. 1974. Periodicity in generations of automata. *Mathematical systems theory*, 8, 1, 15–32. DOI: 10.1007/BF01761704. https://doi.org/10.1007/BF01761704.

[10] Alonzo Church. 1960. Application of recursive arithmetic to the problem of circuit synthesis. In *Summaries of talks presented at the Summer Institute for Symbolic Logic—Cornell University, 1957.* Vol. 28. Communications Research Division, Institute for Defense Analyses, Princeton, New Jersey, 3–50.

[11] Yusuke Endoh. 2020. 128-language uroboros quine relay. https://github.com/mame/quine-relay.

[12] Vanessa Gratzer and David Naccache. 2006. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Advances in Cryptology - EUROCRYPT 2006.* Serge Vaudenay, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 48–58.

[13] Jason Hemann. 2020. *Constraint microKanren in the CLP Scheme.* Ph.D. Dissertation. Indiana University. http://hdl.handle.net/2022/25183.

[14] Douglas Hofstadter. 1979. *Gödel, Escher, Bach : an eternal golden braid.* Basic Books, New York.

[15] Jürgen Kraus. 2009. On self-reproducing computer programs. Trans. by Daniel Bilar and Eric Filiol. *Journal in Computer Virology*, 5, 1, 9–87. *On Self-Reproducing Computer Programs.* Daniel Bilar and Eric Filiol, (Eds.) DOI: 10.1007/s11416-008-0115-z. link.springer.com/journal/11416/5/1.

[16] C. Y. Lee. 1963. A Turing machine which prints its own code script. In *Proc. of the Symposium on Mathematical Theory of Automata*, 155–164.

[17] C. M. W. Little. 2002. Parametrized quotation and self-reference. *Electronic Notes in Theoretical Computer Science*, 74, 69–88. MFCSIT 2002, Second Irish Conf. on the Mathematical Foundations of Computer Science and Information Technology. DOI: 10.1016/S1571-0661(04)80766-7. doi.org/10.1016/S1571-0661(04)80766-7.

[18] David Madore. 2020. Quines (self-replicating programs). Last accessed 5 May 2020. madore.org/~david/computers/quine.html.
[19] Zohar Manna and Richard Waldinger. 1975. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6, 2, 175–208. DOI: https://doi.org/10.1016/0004-3702(75)90008-9. http://www.sciencedirect.com/science/article/pii/0004370275900089.
[20] Maurice Margenstern and Yurii Rogozhin. 2002. Self-describing Turing machines. *Fundamenta Informaticae*, 50, 3-4, 285–303.
[21] John McCarthy. 1978. A micro-manual for LISP -not the whole truth. *ACM Sigplan Notices*, 13, 8, 215–216.
[22] Yiannis N. Moschovakis. 2010. Kleene's amazing second recursion theorem. *The Bulletin of Symbolic Logic*, 16, 2, 189–239. DOI: 10.2178/bsl/1286889124.
[23] John Myhill. 1964. The abstract theory of self-reproduction. *Views on general systems theory*, 106–118.
[24] Chris Olah and Shan Carter. 2017. Research debt. *Distill*. https://distill.pub/2017/research-debt. DOI: 10.23915/distill.00005.
[25] Kostas Pagiamtzis. 2010. Oscillating (iterating) quine with initial transient. Last accessed 5 May 2020. pagiamtzis.com/articles/iterating-quine-with-transient/.
[26] Andrew Polonsky. 2011. Axiomatizing the Quote. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings* (LIPIcs). Marc Bezem, (Ed.) Vol. 12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 458–469. DOI: 10.4230/LIPIcs.CSL.2011.458. https://doi.org/10.4230/LIPIcs.CSL.2011.458.
[27] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2019. Certified semantics for minikanren. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop* number TR-02-19. Cambridge, Massachusetts, 80–98. dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.
[28] Raymond M Smullyan. 1994. *Diagonalization and Self-Reference*. Clarendon Press, Oxford New York.
[29] James W Thatcher. 1963. The construction of a self-describing Turing machine. In *Proc. of the Symposium on Mathematical Theory of Automata*, 165–171.
[30] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM*, 27, 8, (08/1984), 761–763. DOI: 10.1145/358198.358210. https://doi.org/10.1145/358198.358210.
[31] Ann Yasuhara. 1971. *Recursive Function Theory and Logic*. Academic Press, New York.