Relational Processing for Fun and Diversity

Simulating a CPU relationally with miniKanren

GILMORE R. LUNDQUIST, University of Texas at Dallas, USA
UTSAV BHATT, University of Texas at Dallas, USA
KEVIN W. HAMLEN, University of Texas at Dallas, USA

Defining a central processing unit relationally using miniKanren is proposed as a new approach for realizing assembly code diversification. Software diversity has long been championed as a means of protecting digital ecosystems from widespread failures due to cyberattacks and faults, but is often difficult to achieve in practice. Using relational programming to simulate a processor allows large-scale automatic synthesis of assembly-level code. Early experiments with the technique indicate that such synthesis might lead to better automation of code diversification by breaking the synthesis problem into manageable chunks. An early prototype is presented, with some sample synthesis tasks and discussion of possible future applications.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; *Software safety*; Interpreters; Source code generation; • **Security and privacy →** *Software security engineering*;

Additional Key Words and Phrases: program synthesis, relational programming, miniKanren, software artificial diversity, malware polymorphism

1 INTRODUCTION

Software diversity has long been recognized as valuable for protecting digital ecosystems from widespread failures due to cyberattacks and faults [Cohen 1993]. Higher diversity of software implementations reduces the likelihood that a single, common flaw pervades all its deployments, and therefore that a single attack can compromise all members of the ecosystem or affect them all in the same way. Unfortunately, most software ecosystems today remain highly monocultural—all deployments of a given software product for a given architecture are almost identical, save for minute differences. This homogeneity often allows individual, low-cost cyberattacks to have a devastating impact on large numbers of computer systems. For example, the 2015 Stagefright 2.0 vulnerability left nearly all Android devices susceptible to remote compromise due to almost identical multimedia library implementations being used by nearly all apps on all the devices [Peters 2015].

One reason why software monoculture continues to abound despite its brittleness against attack is the uniformity of most present-day tools for developing software products. Compilers are typically designed to solve an optimization problem that translates a given source program into a single, efficient, equivalently behaved object program. Compiler design goals therefore typically include semantic transparency (behavioral preservation), runtime efficiency, and space efficiency, but not diversity. The source semantics of mainstream imperative languages (e.g., C/C++), the optimization stages of their compilers, and their backend code generation algorithms are all designed to search for a single, good solution to this optimization problem.

Authors' addresses: Gilmore R. Lundquist, University of Texas at Dallas, USA, gilmore.lundquist@utdallas.edu; Utsav Bhatt, University of Texas at Dallas, USA, utsav.bhatt@utdallas.edu; Kevin W. Hamlen, University of Texas at Dallas, USA, hamlen@utdallas.edu.

Our research seeks to shift the compiler optimization problem to pursue diversity as a design goal of software development. In particular, a diversifying software development methodology should attempt to yield a maximally dissimilar collection of object code implementations that are all semantically transparent to the original source code and that meet a certain baseline efficiency.

Although a few code diversification strategies have attained widespread deployment in practice, most provide only limited forms of diversity that continue to leave ecosystems vulnerable to exploitation. For example, *Address Space Layout Randomization* (ASLR) diversifies programs by randomly choosing the base addresses of libraries at load-time. While this does help mitigate some attacks, the resulting diversity is low, leaving the software vulnerable to *derandomization attacks* [Shacham et al. 2004]. We believe that changing implementations in a more fundamental way, such as modifying how values are computed, will protect against broader classes of attacks.

Historically, however, this type of diversity has been expensive to obtain and difficult to achieve in an automated fashion. Prior work [Lundquist et al. 2016] proposes merging the fields of artificial diversity of software with that of program synthesis, leveraging the natural diversity of search-style problems in order to create a plethora of program implementations. Because they spring from search problems, these implementations could be fundamentally different ways of solving a given computational problem.

One approach to program synthesis makes use of miniKanren[1] [Byrd 2009], a family of logic languages typically implemented as an embedded *Domain-Specific Language* (DSL). Since it is a relational language, miniKanren programmers write specifications that relate values, and users submit queries that yield sets of values within the relation. Since any value can queried by the system, values in computations that are traditionally thought of as "inputs" or "outputs" need not be. Querying for "inputs" that relate to a specified "output" effectively reverses the computation.

Prior work [Byrd et al. 2017, 2012] has proposed realizing program synthesis in miniKanren by implementing the relational specification of an interpreter that relates input code to the output values it produces. Reversing the computation by querying for inputs from given outputs then produces possible code that produces the desired output values.

Program synthesis in miniKanren seems particularly well suited to the goal of code diversity given miniKanren's natural ability to easily produce multiple answers for a query. The user simply specifies how many answers (at a maximum) the system is to search for, and a list of results is returned.

In the interest of being source code-agnostic with our diversification efforts, we would like to synthesize programs at the assembly level. This potentially has the advantages of breaking the synthesis problem into manageable chunks, as well as allowing existing code (of any origin) to be diversified in a largely automated fashion.

Since we wish to achieve program diversity at the assembly level, we propose writing a relational specification of a central processing unit. This miniKanren program interprets assembly code by relating that code to the states of the processor before and after code execution. A user can then query for assembly code that produces a desired output state.

2 MODES OF OPERATION

We envision the following ways of using an assembly language synthesis system, such as our miniKanren prototype:

2.1 For Diversification

Our main goal is to explore automated diversification of assembly-level code. Potential approaches include the following:

[1]See http://minikanren.org

- Use of code sketches:
  A standard approach to program synthesis is that of the *program sketch* [Solar-Lezama 2008, 2009]: a partial program with holes to be filled with synthesized code. In a logic programming environment, a sketch is a partial program with portions (here instruction mnemonics, instruction arguments, etc.) represented by logic variables. Those variables are then queried using a `run` command to determine possible values. To control which output is being computed, a sketch must also include a formal specification of what is to be computed. In assembly diversification, these specifications are constraints dictating the possible processor states after a computation runs. (We use sketches in this paper.)
  Constraints collectively define correctness of synthesized answers. Correct synthesis algorithms yield only answers that are correct with respect to the constraints. For code diversification, constraints must therefore also define what is meant by program equivalence. For example, a user may decide to specify result values for processor status flags, or leave them unconstrained. The latter option broadens the definition of equivalence to include relations on processor states. For example, diversification might be required to preserve register and memory values but be permitted to vary status flag values.
  When diversifying existing code, either the program to be diversified already has a formal (mathematical) specification, or (more commonly) there will *a priori* be no formal notion of correctness. In the ideal former case, we need only convert the specification into miniKanren constraints. In the latter case, quality assurance and testing processes must be applied to check program correctness or equivalence; however, for diversity to be tractable, this assurance process should be (semi-)automated to easily apply it to the multitude of variants generated.
- Basic blocks bracketed by control flow instructions:
  Generating proper control-flow using program synthesis is known to be a hard problem [Solar-Lezama 2008]. To avoid this, sketches can explicitly specify most or all control-flow transfer instructions [Solar-Lezama 2008]. Instructions in the basic block(s) between control-flow transfers are synthesized in entirety or in part. Program constraints include loop invariants to dictate what is synthesized. (The GCD examples in the following section illustrate this approach.)
- Use of effect traces:
  In addition to dividing code up into basic blocks, another approach is to generate code based on a known effect trace, synthesizing portions between calls to other functions, system routines, or other side effects not otherwise modeled by our processor relation. The format and construction of arguments to these external routines are determined by constraints provided in the sketch. Since trace equivalence is a common way to define program equivalence, this approach provides an intuitive starting point for solving the equivalence problem mentioned above.
- Diversification of existing code:
  One of our primary goals for this project is the automated diversification of existing code. To achieve this, processor states could be generated by running existing assembly code forwards in our system. Enhanced by constraints describing known properties of how the code should function, these processor states become the end goals for new code to satisfy.
- Gadget-oriented program composition:
  A *gadget* can be defined as any arbitrary string of assembly language instructions, usually one found in pre-existing code. As shown in prior work [Lundquist et al. 2016; Mohan and Hamlen 2012] (and as shown by Return-Oriented Programming in general [Schwartz et al. 2011]), programs can be constructed by stringing chains of gadgets together in an order determined by the synthesis engine as a means of reaching a particular goal. Return-Oriented Programming (ROP) is an exploit technique that repurposes gadgets found in existing benign code to implement attack payloads. While gadget-oriented programming was

originally used for malicious purposes, we speculate that this technique could be adapted for more general synthesis tasks.

The overall theme of the techniques described above is that of breaking the problem of synthesis into small, manageable chunks at a low level. Rather than trying to synthesize an entire program at once, a program is broken down into sub-pieces, which are then synthesized. This avoids traditional problems associated with synthesizing large or complex code fragments, and mitigates state-space explosion.

2.2 For Use by a Compiler

Since programming using processor states and constraint sets is likely to be difficult, we envision this system as a target of higher level tools, such as compilers or interpreters. Replacing the back-end code generation of a compiler with a synthesis system allows for automatic diversification while keeping programming tasks manageable. Compilers are potentially in a unique position to know what constraints must be preserved from higher-level source in the generated assembly. Further, allowing compilers and other code-generation tools to programmatically break the result into known intermediate states allows for smaller and more frequent synthesis tasks, once again keeping with the theme of reducing synthesis problems into smaller sub-problems. Sub-problems can be divided at whatever level is appropriate for the tool based on its knowledge and analysis of the code being generated, or based on what size tasks the synthesis engine is capable of handling efficiently.

We plan to use our system to explore each of the above approaches in future work.

3 IMPLEMENTATION

To explore assembly language synthesis, we have developed a prototype relational assembly interpreter in miniKanren.[2] Our prototype uses a subset of Intel x86 assembly language instructions with the limited set of 32-bit general purpose registers available for that architecture. The implementation uses a modified version of faster-miniKanren[3] written in Racket.

A full scale system must support enough instructions and processor features to synthesize programs that work; but like existing compiler back-ends (which emit only a subset of the available instructions), this does not necessitate supporting the entire instruction set architecture (ISA). Supporting more of the ISA allows for more diversity in the resulting programs, potentially at the cost of synthesis time. However, prior work [Mohan and Hamlen 2012] has found that only a small portion of the architecture is needed to achieve high code diversity.

3.1 The x86^o Relation

Our processor relation x86^o relates assembly code, an input processor state, and an output processor state. Processor state is modeled as a set of association lists, one mapping registers to values and another mapping memory locations to values. The memory mapping is a partial function, only containing those values that have been updated by the program. Addresses read from an uninitialized address return a default value, typically 0.

The interpreter models assembly code as a list of instructions, using a fall-through approach to execute (or synthesize) a basic block. Since we do not yet represent code addresses in our model, execution always necessarily continues to the subsequent instruction in the list. Each instruction is a list containing an instruction mnemonic and the appropriate number of operands for the instruction. Each operand describes a register, an immediate value, or a memory address.

[2]Code is available for download at https://www.utdallas.edu/~hamlen/lundquist-miniKanren19.zip.
[3]Obtained from https://github.com/gregr/tutorial-relational-interpreters. This version contains modifications by Greg Rosenblatt to speed up the eval^o relational interpreter. The original faster-miniKanren can be obtained from https://github.com/michaelballantyne/faster-miniKanren/blob/master/README.md.

The interpreter calls sub-relations to decode arguments (which relate operands to their values with respect to some processor state) and then interprets the instruction by operating on those values and creating an updated processor state. A fragment of the relation with a few instructions is shown below:

```
;; The x86 processor relation.
(define (x86^o code rstore new-rstore)
  (conde
    [(≡ '() code) (≡ rstore new-rstore)]

    [(fresh (opcode arglist morecode op1 op2 res rstore1)
       (≡ code `((,opcode . ,arglist) . ,morecode))
       (conde

         ;; add
         [(≡ opcode 'add)
          (decode-2args^o arglist op1 op2 rstore)
          (plus^o op1 op2 res)
          (update-arg^o arglist res rstore rstore1)
          (x86^o morecode rstore1 new-rstore)]
...
         ;; div
         ;;  uses implicit arguments -
         ;;    dividend is edx:eax,
         ;;    divisor is op1,
         ;;    destination (result quotient) is eax,
         ;;    (result) remainder is edx
         [(≡ opcode 'div)
          (decode-1arg^o arglist op1 rstore)
          (fresh (edx eax n rem rstore2)
            (lookup^o 'R_EDX rstore eax)
            (lookup^o 'R_EAX rstore edx)
            (append^o eax edx n) ; n=dividend: eax=low order bits, edx=high order bits
            (div^o n op1 res rem)
            (update^o 'R_EAX res rstore rstore1)
            (update^o 'R_EDX rem rstore1 rstore2)
            (x86^o morecode rstore2 new-rstore))]
...
         ;; xor
         [(≡ opcode 'xor)
          (decode-2args^o arglist op1 op2 rstore)
          (xor^o op1 op2 res)
          (update-arg^o arglist res rstore rstore1)
          (x86^o morecode rstore1 new-rstore)]
...
```

Numeric values in our system are represented as *Oleg numerals*—little-endian lists of binary digits that encode the base-2 representation of the number. Most arithmetic operations are the definitions[4] found in *The Reasoned Schemer* [Friedman et al. 2018], with a few of our own added to implement missing operators (e.g., logical operations) needed for the instructions we've included. For example, here is our implementation of integer division and remainder:

```
; integer division with remainder
(define (divᵒ a b q r)
  (conde
    [(≡ q '()) (≡ r a) (<lᵒ a b)]
    [(fresh (p)
       (<=lᵒ b a)
       (<lᵒ r b)
       (plusᵒ p r a)
       (*ᵒ b q p))]))
```

3.2 Example Queries

3.2.1 Synthesis 1: Generating Some Assembly. Our first example interaction demonstrates a query for generating lots of assembly code quickly. The goal of each synthesized program is to place the result value 1875 (Oleg numeral (1 1 0 0 1 0 1 0 1 1 1)) into register EAX.

We begin by providing the following assembly program sketch:

```
mov ECX ?ₓ
mov EDX ?_y
mov EAX ?_p
mov EBX ?_q
mov EDI ?_s
mov ESI ?_t

?_g ECX ESI
?_f EDX EDI
?_d EAX ECX
?_e EBX EDX
?_c EAX EBX
```

The question marks denote holes to be filled in by the synthesizer, subscripted with the name of the logic variable used to denote the hole. The first six instructions denote mov instructions with a particular destination operand (one for each general-purpose register) and a hole to be filled in for the source operand. The remaining five instructions give concrete destination and source operands but leave the choice of instruction open to be synthesized.

Constraints are then added to further limit what will be synthesized. The first set of constraints prevent any of the final instructions from being additional mov instructions. The second set prevents some source operands from

[4]Obtained from https://github.com/miniKanren/CodeFromTheReasonedSchemer2ndEd.

being the same as some of the other source operands. The third set of constraints disallow source operand holes from containing 1875, which prevents the goal value from being moved directly into a register. Finally, source operands must be positive (non-zero).

We use our processor relation to relate the sketch code, an initial empty processor state, and a final processor state. Constraining this final state by the value in EAX forces our desired goal to be met. We then query for the fully synthesized code (in variable `v1`) and the resulting processor state (in variable `z`).

For our queries, we make use of Racket's `time` operator, which reports (in milliseconds) CPU time, real time, and garbage collection time spent for expression evaluation. The full query and first 2 (of 200) results are as follows:

```
> (time
    (run 200 (v1 z)
      (fresh (d x p e q y c s t f g)
         (≡ v1 `((mov R_ECX ,x)
                 (mov R_EDX ,y)
                 (mov R_EAX ,p)
                 (mov R_EBX ,q)
                 (mov R_EDI ,s)
                 (mov R_ESI ,t)

                 (,g R_ECX R_ESI)
                 (,f R_EDX R_EDI)
                 (,d R_EAX R_ECX)
                 (,e R_EBX R_EDX)
                 (,c R_EAX R_EBX)))
         (≢ c 'mov) (≢ d 'mov) (≢ e 'mov) (≢ f 'mov) (≢ g 'mov)
         (≢ p q) (≢ x q) (≢ y p) (≢ y q) (≢ x p) (≢ x y)
         (≢ x '(1 1 0 0 1 0 1 0 1 1 1))
         (≢ y '(1 1 0 0 1 0 1 0 1 1 1))
         (≢ q '(1 1 0 0 1 0 1 0 1 1 1))
         (≢ p '(1 1 0 0 1 0 1 0 1 1 1))
         (≢ s '(1 1 0 0 1 0 1 0 1 1 1))
         (≢ t '(1 1 0 0 1 0 1 0 1 1 1))
         (pos^o x) (pos^o y) (pos^o p) (pos^o q) (pos^o t) (pos^o s)
         (x86^o v1 initial-store z)
         (lookup^o 'R_EAX z '(1 1 0 0 1 0 1 0 1 1 1))
  )))
cpu time: 21109 real time: 21481 gc time: 9204
'(((((mov R_ECX (_.0 . _.1))
     (mov R_EDX (_.2 . _.3))
     (mov R_EAX (1))
     (mov R_EBX (0 1 0 0 1 0 1 0 1 1 1))
     (mov R_EDI (_.2 . _.3))
     (mov R_ESI (_.0 . _.1))
     (sub R_ECX R_ESI)
     (sub R_EDX R_EDI)
     (add R_EAX R_ECX)
     (add R_EBX R_EDX)
     (add R_EAX R_EBX))
    ((R_EAX 1 1 0 0 1 0 1 0 1 1 1) (R_EBX 0 1 0 0 1 0 1 0 1 1 1) (R_ECX) (R_EDX)
     (R_EDI _.2 . _.3) (R_ESI _.0 . _.1)))
   (=/=
    ((_.0 0) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 ()))
    ((_.0 _.2) (_.1 _.3))
    ((_.2 0) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 ()))))
  ((((mov R_ECX (_.0 . _.1))
     (mov R_EDX (_.2 . _.3))
     (mov R_EAX (0 1 0 0 1 0 1 0 1 1 1))
     (mov R_EBX (1))
     (mov R_EDI (_.2 . _.3))
     (mov R_ESI (_.0 . _.1))
     (sub R_ECX R_ESI)
     (sub R_EDX R_EDI)
     (add R_EAX R_ECX)
     (add R_EBX R_EDX)
     (add R_EAX R_EBX))
   ((R_EAX 1 1 0 0 1 0 1 0 1 1 1) (R_EBX 1) (R_ECX) (R_EDX) (R_EDI _.2 . _.3) (R_ESI _.0 . _.1)))
   (=/=
    ((_.0 0) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 ()))
    ((_.0 _.2) (_.1 _.3))
    ((_.2 0) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 ()))))
...
```

3.2.2 Synthesis 2: Reverse ALU. Next we give an example of *Angelic Execution*—determining which sets of inputs successfully result in a particular output [Bodik et al. 2010; Chandra et al. 2011]. Here the output value 65,535 must be assigned to register EAX after computing the following assembly fragment:

```
  mul EAX EBX
  or ECX EDX
  add ESI EDI
  dec EAX
  xor ESI ECX
  inc ECX
  xor EAX ECX
```

Each synthesized output contains a set of possible positive input values, one for each of the six general-purpose registers. Each input set successfully results in the desired output.

The full query and first 5 (of 200) results are shown below:

```
> (time
    (run 200 (v1 v2 v3 v4 v5 v6)
      (fresh (a b c x y z str)
        (pos^o a) (pos^o b) (pos^o c) (pos^o x) (pos^o y) (pos^o z)
        (x86^o `((mov R_EAX ,x)
                (mov R_EBX ,y)
                (mov R_ECX ,z)
                (mov R_EDX ,a)
                (mov R_ESI ,b)
                (mov R_EDI ,c)
                (mul R_EAX R_EBX)
                (or R_ECX R_EDX)
                (add R_ESI R_EDI)
                (dec R_EAX)
                (xor R_ESI R_ECX)
                (inc R_ECX)
                (xor R_EAX R_ECX))
              initial-store
              str)
        (lookup^o 'R_EAX str '(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
        (≡ `(R_EAX ,x) v1)
        (≡ `(R_EBX ,y) v2)
        (≡ `(R_ECX ,z) v3)
        (≡ `(R_EDX ,a) v4)
        (≡ `(R_ESI ,b) v5)
        (≡ `(R_EDI ,c) v6)
  )))
cpu time: 14062 real time: 14129 gc time: 4384
'(((R_EAX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (1)) (R_EBX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (0 _.0 . _.1)))
  ((R_EAX (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (0 1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (0 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_ESI (1)) (R_EDI (1)))

...
```

3.2.3 Specifying Loop Constraints: Euclid's GCD Algorithm. For our final example, we use a slightly higher-level code fragment to demonstrate how we handle looping or stitching together multiple code blocks. Our prototype currently only handles straight-line code with no jumps. To specify code with loops or more complicated control flow, one can specify how the blocks are to be sequenced using miniKanren.

As our example, we use Euclid's algorithm for finding the Greatest Common Divisor (GCD) of two integers a and b. Recall that the algorithm proceeds as follows:

(1) Find the remainder r when dividing a and b. (Equivalently, find integers q and r such that $a = qb + r$).
(2) If $r = 0$ then b already divides both a and b, and is the largest integer that divides both a and b; return b.
(3) Otherwise, any number that divides a and b must also divide b and r. Repeat from step (1) to find the GCD of b and r.

To implement this algorithm, first we adopt the convention that the values for a and b are held in registers EAX and EBX, respectively. We then write a miniKanren relation `gcd_tester`^{o} to run the loop. Given an assembly code body and input values, the code is run in a processor environment with values in their proper registers. It then checks the remainder result, and either associates the output with the final value (if the remainder is zero) or recursively calls itself on the resulting b and r values.

```
; recursive loop which calls the basic block [the loop body]
(define (gcd_testerᵒ code a b gcd)
  (fresh (s s+ stmp)
    (updateᵒ 'R_EAX a initial-store stmp)
    (updateᵒ 'R_EBX b stmp s)
    (gcd_block_codeᵒ code s s+)
    (conde
      [(lookupᵒ 'R_EBX s+ '() )  ; final remainder is 0
       (lookupᵒ 'R_EAX s+ gcd)]  ; result is in EAX
      [(fresh (r)
         (lookupᵒ 'R_EBX s+ r)
         (≢ r '()) ; r is non-0
         (gcd_testerᵒ code b r gcd))])))  ; otherwise loop
```

Relation gcd_block_code^o specifies a loop invariant for our algorithm. It runs our x86 simulator on the given code, assuming the above register convention and specifying mathematical constraints for outputs that a correct run should produce. In this case, the constraints require that after the code runs, the new a value is the original b value and the new b value is the remainder of a and b.

```
; spec for gcd loop invariant
(define (gcd_block_codeᵒ code s s+)
  (fresh (old_eax old_ebx new_eax new_ebx q r)
    (lookupᵒ 'R_EAX s old_eax)  ; a
    (lookupᵒ 'R_EBX s old_ebx)  ; b
    (x86ᵒ code s s+)
    (lookupᵒ 'R_EAX s+ new_eax)
    (lookupᵒ 'R_EBX s+ new_ebx)
    (≡ new_eax old_ebx)  ; a ← b
    (divᵒ old_eax old_ebx q r)  ; a = qb + r,  r < b
    (≡ new_ebx r)))
```

Using this infrastructure, we now run an algorithm on our processor and verify the answers computed are correct.

```
; euclid's gcd algorithm
(define euclid '(
  (xor R_EDX R_EDX)
  (div R_EBX)
  (mov R_EAX R_EBX)
  (mov  R_EBX R_EDX)))

; test euclid's algorithm and get an answer
> (time (run* (d) (gcd_testerᵒ euclid (build-num 12) (build-num 9) d)))
cpu time: 78 real time: 80 gc time: 0
'((1 1))

> (time (run* (d) (gcd_testerᵒ euclid (build-num 42) (build-num 30) d)))
cpu time: 203 real time: 213 gc time: 15
'((0 1 1))
```

We see that we get the correct answers of 3 and 6, resepectively (in Oleg numeral representation). As with any miniKanren relation, we can query for multiple answers at once; here we simultaneously obtain the gcd of 12 and all values of b up to 12, and verify that the answers are correct:

```
> (time (run* (b d) (<=ᵒ b (build-num 12)) (gcd_testerᵒ euclid (build-num 12) b d)))
cpu time: 1329 real time: 1336 gc time: 937
'(((0 0 1 1) (0 0 1 1))
  ((1) (1))
  ((0 1) (0 1))
  ((1 1) (1 1))
  ((0 0 1) (0 0 1))
  ((0 1 1) (0 1 1))
  ((1 1 0 1) (1))
  ((1 0 1) (1))
  ((0 1 0 1) (0 1))
  ((0 0 0 1) (0 0 1))
  ((1 0 0 1) (1 1)))
```

Interestingly, we see three different classes of answers to this query, generated in order: the first six are those for which $b = d$, namely the factors of 12 (12, 1, 2, 3, 4, and 6). The next group (11 and 5) are those that are *relatively prime* with 12, for which the only common divisor is 1. The remaining three answers (10, 8, and 9) are those that are not factors of 12, yet still have common factors (2, 4, and 3, respectively) with 12.

3.2.4 Synthesis 3: Synthesizing Euclid's Algorithm with a Sketch. We now synthesize a similar algorithm by querying for the code. An initial, naïve approach attempts to synthesize the code directly using the relations we have so far:

```
> (time (run 1 (c) (gcd_testerᵒ c (build-num 12) (build-num 9) (build-num 3))))
. . .
    ; [Fails with out-of-memory error]
```

The out of memory error occurs because this naïve query is too vague, resulting in a state space explosion. To narrow the search space, we employ a program sketch. Using the intuition that only a division operation and moving some data around should be sufficient, we create a program sketch that only allows `div` and `mov` instructions, with a specified program length. We take advantage of relational programming to auto-generate a suitable sketch:

```
(define (gen-mov-div-listᵒ depth out)
  (conde
    [(≡ depth '()) (≡ out '())]
    [(≢ depth '())
     (fresh (n l)
        (minusᵒ depth '(1) n)
        (conde
          [(fresh (op1 op2) (≡ out `((mov ,op1 ,op2) . ,l) ))]
          [(fresh (op)      (≡ out `((div ,op)       . ,l) ))]
        )
        (gen-mov-div-listᵒ n l) )]))

> (time (run 1 (c) (gen-mov-div-listᵒ (build-num 4) c)
                   (gcd_testerᵒ c (build-num 12) (build-num 9) (build-num 3))))
cpu time: 361204 real time: 362308 gc time: 84412
'(((mov R_ECX ()) (div R_EBX) (mov R_EAX R_EBX) (mov R_EBX R_EDX)))
```

Here we have limited possible sketches to programs of length 4 with only `div` or `mov` instructions, and the system is now able to produce an answer. Upon running the test queries below, we see that this new code produces the same correct GCD answers as the original code above.

```
(define new-code '((mov R_ECX ()) (div R_EBX) (mov R_EAX R_EBX) (mov R_EBX R_EDX)))
> (time (run* (d) (gcd_testerᵒ new-code (build-num 12) (build-num 9) d)))
cpu time: 110 real time: 107 gc time: 46
'((1 1))
> (time (run* (d) (gcd_testerᵒ new-code (build-num 42) (build-num 30) d)))
cpu time: 265 real time: 262 gc time: 31
'((0 1 1))
> (time (run* (b d) (<=ᵒ b (build-num 12)) (gcd_testerᵒ new-code (build-num 12) b d)))
cpu time: 422 real time: 437 gc time: 76
'(((0 0 1 1) (0 0 1 1))
  ((1) (1))
  ((0 1) (0 1))
  ((1 1) (1 1))
  ((0 0 1) (0 0 1))
  ((0 1 1) (0 1 1))
  ((1 1 0 1) (1))
  ((1 0 1) (1))
  ((0 1 0 1) (0 1))
  ((0 0 0 1) (0 0 1))
  ((1 0 0 1) (1 1)))
```

4 RELATED WORK

Rosette [Torlak and Bodik 2013] is another DSL embedded in Racket capable of program synthesis and angelic execution. Instead of backtracking search, Rosette uses a *satisfiability modulo theories* (SMT) solver to find solutions to synthesis and constraint problems. This gives it the potential to be more efficient than miniKanren in finding solutions that are arithmetic in nature. While any sufficiently general program synthesis system should be able to synthesize assembly code in a similar fashion to our approach, we found the Rosette system to be focused on returning a single optimal answer to queries. This makes it more cumbersome to achieve large-scale diversity with Rosette than with miniKanren in our experience.

Minimips[5] is a miniKanren implementation of a MIPS architecture assembler/dissassembler. Its relation converts between MIPS assembly language programs and their binary encodings. Minimips contains a full syntactic description for MIPS instructions, but doesn't appear to have an interpreter. It therefore has no semantic description of the instructions, or the ability to synthesize code that satisfies a formal specification. We chose to use the x86 architecture rather than a RISC architecture such as MIPS because a CISC architecture naturally allows for more diversity of implementations. As future work we plan to implement a similar relational assembler/disassembler for our x86 system.

Automated generation of assembly code is a common task of compilers and other similar tools. As noted in Section 1, these tools do not have diversity of implementation as a goal. Some systems (e.g., [Hong and Gerber 1993; Pu et al. 1988]) do synthesize assembly code for specific tasks using algorithms unique to the task. However, these systems do not synthesize general-purpose assembly code from arbitrary program constraints, or for the purpose of implementation diversity.

5 CONCLUSION

This paper proposed the implementation of an assembly-level interpreter in miniKanren for the purpose of synthesizing assembly code, motivated by the need for increased software diversity. This allows synthesis problems to be broken up into small, manageable pieces. Such problems can be subdivided using specific sketches, basic blocks, or effect traces; and can be driven by various inputs, including effect traces, processor states obtained from existing code, availability of particular gadgets, or compiler-driven information. Our working prototype implements an assembly interpreter for a small subset of x86, and experiments demonstrate its use for synthesizing code in both straight-line and looping programs. Synthesis examples show the potential to synthesize large numbers of diverse implementations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and William Byrd for their helpful comments on this paper. We would also like to thank the committee chair Nada Amin and all of the miniKanren workshop organizers. The research reported herein was supported in part by NSF Award #1513704, ONR Award N00014-17-1-2995, and an endowment from the Eugene McDermott family. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the above supporters.

REFERENCES

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 339–352.

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University, Bloomington, Indiana.

[5]https://github.com/orchid-hybrid/minimips

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of the ACM on Programming Languages (PACMPL)* 1 (2017), 8:1–8:26.

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 Annual Workshop on Scheme and Functional Programming*. 8–29.

Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 121–130.

Fred B. Cohen. 1993. Operating System Protection Through Program Evolution. *Computers & Security* 12, 6 (1993), 565–584.

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (second ed.). MIT Press.

Seongsoo Hong and Richard Gerber. 1993. Compiling Real-time Programs into Schedulable Code. In *Proceedings of the 14th ACM Conference on Programming Language Design and Implementation (PLDI)*, Vol. 28. 166–176.

Gilmore R. Lundquist, Vishwath Mohan, and Kevin W. Hamlen. 2016. Searching for Software Diversity: Attaining Artificial Diversity through Program Synthesis. In *Proceedings of the New Security Paradigms Workshop (NSPW)*. 80–91.

Vishwath Mohan and Kevin W. Hamlen. 2012. Frankenstein: Stitching Malware From Benign Binaries. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*. 77–84.

Sara Peters. 2015. Stagefright 2.0 Vuln Affects Nearly All Android Devices. *DARKReading* (October 2015).

Calton Pu, Henry Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1, 1 (1988), 11–32.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium*. 25–41.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. 298–307.

Armando Solar-Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. The University of California, Berkeley.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS)*. 4–13.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 3rd ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)* 135–152.