Program Synthesis with Algebraic Library Specifications

BENJAMIN MARIANO, University of Maryland, College Park, USA
JOSH REESE, University of Maryland, College Park, USA
SIYUAN XU, Purdue University, USA
THANHVU NGUYEN, University of Nebraska, Lincoln, USA
XIAOKANG QIU, Purdue University, USA
JEFFREY S. FOSTER, Tufts University, USA
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology, USA

A key challenge in program synthesis is synthesizing programs that use libraries, which most real-world software does. The current state of the art is to model libraries with *mock* library implementations that perform the same function in a simpler way. However, mocks may still be large and complex, and must include many implementation details, both of which could limit synthesis performance. To address this problem, we introduce JLibSketch, a Java program synthesis tool that allows library behavior to be described with algebraic specifications, which are rewrite rules for sequences of method calls, e.g., encryption followed by decryption (with the same key) is the identity. JLibSketch implements rewrite rules by compiling JLibSketch problems into problems for the Sketch program synthesis tool. More specifically, after compilation, library calls are represented by abstract data types (ADTs), and rewrite rules manipulate those ADTs. We formalize compilation and prove it sound and complete if the rewrite rules are ordered and non-unifiable. We evaluated JLibSketch by using it to synthesize nine programs that use libraries from three domains: data structures, cryptography, and file systems. We found that algebraic specifications are, on average, about half the size of mocks. We also found that algebraic specifications perform better than mocks on seven of the nine programs, sometimes significantly so, and perform equally well on the last two programs. Thus, we believe that JLibSketch takes an important step toward synthesis of programs that use libraries.

CCS Concepts: • **Software and its engineering** → Automatic programming; Source code generation.

Additional Key Words and Phrases: Sketch-based Program Synthesis, Algebraic Specification, Term Rewriting, Java

ACM Reference Format:
Benjamin Mariano, Josh Reese, Siyuan Xu, ThanhVu Nguyen, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2019. Program Synthesis with Algebraic Library Specifications. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 132 (October 2019), 25 pages. https://doi.org/10.1145/3360558

1 INTRODUCTION

In recent years, there has been tremendous progress on *sketch-based program synthesis*, which, given a partial program (or *sketch*) containing some unknowns, solves for those unknowns so the resulting program satisfies its assertions [Solar-Lezama 2013; Solar-Lezama et al. 2006].

Authors' addresses: Benjamin Mariano, University of Maryland, College Park, USA; Josh Reese, University of Maryland, College Park, USA; Siyuan Xu, Purdue University, USA; ThanhVu Nguyen, University of Nebraska, Lincoln, USA; Xiaokang Qiu, Purdue University, USA; Jeffrey S. Foster, Tufts University, USA; Armando Solar-Lezama, Massachusetts Institute of Technology, USA.

2475-1421/2019/10-ART132
https://doi.org/10.1145/3360558

Sketch-based synthesis allows users to leverage knowledge about the structure of the desired program to reduce the program search space. This technique has been used effectively for a wide variety of synthesis problems, including synthesizing Java framework models [Jeon et al. 2016], generating automated feedback for introductory Python assignments [Singh et al. 2013], and providing social media recommendations [Cheung et al. 2012] among others [Bornholt and Torlak 2017; Bornholt et al. 2016; Cardelli et al. 2017; D'Antoni et al. 2016; Hua and Khurshid 2017; Raabe and Bodik 2009; Solar-Lezama et al. 2007, 2008, 2006; Torlak and Bodik 2014; Wang et al. 2017].

While sketch-based synthesis has come a long way, a major remaining challenge is synthesizing programs that use *libraries*, which is common in most real-world software. One possible approach is to inline the library source code into the sketch, but this is untenable in practice: library code is large and complex—yielding synthesis problems that are too big to solve—and may include native code, which the synthesizer cannot reason about.

Instead, a more effective approach is to develop *mock libraries* (or just *mocks*), written in the sketch language, that implement essential library functionality in a simpler way [Jeon et al. 2016; van der Merwe et al. 2015]. However, in practice mocks can be sizable, and they include many implementation details the synthesizer must reason about. As a result, good synthesis performance can be hard to achieve with mocks.

To address this problem, we introduce JLibSketch [1], a novel Java program synthesis tool in which libraries can be described with *algebraic specifications*. We express these specifications as *rewrite rules* of the form *pattern* $\Rightarrow$ *result*. For example, we can partially specify a cryptography library with the rule $\mathsf{decrypt}(\mathsf{encrypt}(m, k), k) \Rightarrow m$, meaning encryption followed by decryption with the same key is the identity. Notice this specification is both dramatically simpler than actual encryption and decryption routines and easy to reason about.

Thus, the JLibSketch synthesis problem is to find a program that is correct for any library satisfying the algebraic specifications and any program input. To solve such a problem, JLibSketch builds on top of JSketch [Jeon et al. 2015b], a compiler from Java code with unknowns to the input language of the Sketch [Solar-Lezama 2016] synthesis system. JLibSketch extends JSketch so that the compiled code models library methods as constructing *algebraic data types* (ADTs) [Inala et al. 2017]. JLibSketch then compiles rewrite rules as Sketch code that manipulates those ADTs. In this way, JLibSketch implements rewrite rules without requiring any modification to the solver. Although our particular implementation is for Java, we believe the same approach can be used to model libraries in other existing synthesis frameworks. (Section 2 gives an overview of JLibSketch.)

We formalize JLibSketch as a compilation from $\text{Sketch}^{+\text{lib}}_{\text{core}}$, a core Sketch language with built-in support for algebraic specifications, to $\text{Sketch}_{\text{core}}$, a core Sketch language without such support. We prove that, if the rewriting rules are ordered and non-unifiable, then soundness and completeness hold, i.e., the $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program has a solution if and only if its compilation has a solution in Sketch. (Section 3 presents our formalism.)

Our JLibSketch implementation extends JSketch's surface syntax to include Java-like notation for algebraic specifications. The implementation embeds its ADT-based representation of library method calls inside JSketch's object representation, so the two may be interchanged. Because JSketch distinguishes primitives and arrays from objects (the former to match Java, the latter for performance), JLibSketch also includes boxing and unboxing annotations to support library methods that might evaluate to a primitive/array or might be represented with an ADT. (Section 4 discusses our implementation.)

We evaluated JLibSketch by algebraically specifying libraries from three domains: data structures, cryptography, and file systems. We then used those specifications to synthesize three library client

[1] https://github.com/bmarwritescode/oopsla-19-artifact

programs from each domain, for a total of nine programs. We found that, in comparison to mocks for the same libraries, algebraic specifications were, on average, 53% of the lines of code of mocks. We also found that, for seven of the nine benchmarks, synthesis ran from 2× to more than 81× faster with algebraic specifications than with mocks. In the other two cases, mocks and algebraic specifications performed about the same. (Section 5 discusses our experiments.)

Thus, our results suggest that algebraic specifications provide a new, more succinct way of describing libraries, and they can provide significant performance benefits compared to mocks. We believe that JLibSketch takes an important step forward in making synthesis of programs that use libraries more practical.

2 OVERVIEW

Background and Motivation. JLibSketch is built on top of JSketch [Jeon et al. 2015b], which brings Sketch-based program synthesis to Java. Figure 1 illustrates some key features of JSketch, which mimic corresponding Sketch features. Each JSketch program contains one or more *harnesses*, which are partial programs that can contain three kinds of unknowns: *holes*, denoted by ??, which represent integer or boolean constants; *expression generators*, denoted by $\{|\ e_1, e_2, \ldots\ |\}$, which represent a choice among the expressions described by the e_i; and *function generators*, described shortly. The goal of JSketch is to find an instantiation of the unknowns such that for all choices of the harness's parameters, its assertions are satisfied.

For example, Figure 1 shows a sample JSketch problem. It begins with a function generator, lin, that describes linear functions of its arguments x and y. More specifically, the if statement (line 2) is guarded by a hole, and thus the synthesizer can pick from one of two returns. The first (line 2) uses an expression generator to select among one of the arguments. The second (line 3) returns the sum of two recursive calls to lin and an unknown constant (i.e., a hole). When this generator is called, either from elsewhere or recursively, JSketch replaces the call with a fresh copy of its body containing fresh holes and generators. Thus, for example, a call of lin could expand to x, x+y+0, y+y+4, etc.

```
1 generator int lin(int x, int y){
2   if(??){ return {| x , y |}; }
3   else{ return lin(x,y)+lin(x,y)+??; }
4 }
5 harness void main(int i, int j){
6   int f = lin(i,j), g = lin(i,j);
7   if (i == 1 && j == 1) { assert f==g; }
8   else { assert f ≠ g; }
9 }
```

Fig. 1. Simple JSketch problem.

Next, the figure shows the harness main, which calls lin twice, generating two linear functions f and g. The first assertion (line 7) checks that f(1,1) =g(1,1). The second assertion (line 8) checks that f(a,b) ≠ g(a,b) for all (a,b) ≠ (1,1). Thus, when given this program, JSketch finds two instantiations of lin (i.e., two linear functions) that are only equivalent on the input (1,1), such as f(x,y)=3x+4y+16 and g(x,y)=2y+21. For more details of JSketch, see Jeon et al. [2015b].

While JSketch is promising, it is difficult to use it to synthesize programs that use libraries. As a running example, in the remainder of this section we will consider one of our benchmarks, *SuffixArray*. Part of this benchmark is to synthesize a method lrs, shown in Figure 2d. We will discuss the details of this figure shortly. For now, we just observe that the code uses TreeSet (denoted TS in the figure), a data structure from the Java standard library. Hence, the synthesizer needs to know the semantics of TreeSet's methods to solve this problem.

The most straightforward solution is to concatenate the TreeSet source code to the sketch, but unfortunately that does not work. The standard TreeSet implementation is built on the TreeMap library, which is over 3,000 lines of code and has dozens of dependencies—which would make the problem too large to be solved—and includes native code—which JSketch does not understand. Instead, we need to build a *model* of the library—a representation that is smaller, simpler, and amenable to synthesis.

One approach to modeling is to write a *mock* implementation of TreeSet that avoids native code and has fewer dependencies. While this can be effective, mocks could also be lengthy, reducing synthesis performance, and writing mocks requires making a range of implementation choices that could have unexpected consequences. For example, if we mocked TreeSet using an object array rather than TreeMap, we would either have to bound the array size—limiting the applicability of the mock—or implement dynamic resizing—which introduces extra complexity that can slow synthesis performance (see Section 5.1 for a more detailed discussion of mocking TreeSet).

Algebraic Specifications using Rewrite Rules. JLibSketch aims to address this issue by extending JSketch to support *algebraic specifications* [Henkel and Diwan 2003] for modelling libraries. For example, Figure 2a gives a partial specification describing relationships among several methods of TreeSet (abbreviated as TS), grouped inside an @rewriteClass (line 1). The annotations in this code, beginning with @, are part of JLibSketch. The specification begins with type signatures, labeled with @alg, for the methods (lines 2-5). Since contains and size do not mutate the receiver, we further annotate them as @pure. Next, we define six rewrite rules, which pattern match combinations of method calls and return a new expression to which that pattern should be rewritten. For compactness, the rules in the figure omit type annotations, which are straightforward but verbose.

The first three rules describe the behavior of size. The first rule (line 7) states that the size of an empty TreeSet is zero. Here the pattern is TreeSet(), which is the nullary constructor for the class, i.e., this rule says (new TreeSet()).size() == 0. Notice that in these specifications, we write the receiver as the first method argument, whereas in Java the receiver is written to the left of the method name and is referred to in the method body as this.

Similarly, the second rewrite rule states that the size of a cleared TreeSet is 0 (line 9). The third rewrite rule (lines 11-14) states that adding an element to a TreeSet increases the size by one if that element is not already in the set. Since add mutates its receiver (it is not @pure), JLibSketch creates two pattern constructors for the method. The constructor add represents the normal return value, and the constructor add! represents the mutated TreeSet after the call.

The last three rewrite rules describe the behavior of contains. The first rule (lines 16-18) states that no element is contained in an empty TreeSet. The second rule (lines 20-22) states that an element is not contained in a TreeSet that was just cleared. The last rule (lines 23-26) states that an element e1 is contained in a TreeSet that just added an element e2 if e1 is equal to e2 or if e1 is contained somewhere in the rest of the TreeSet.

Synthesis with Algebraic Specifications. To see how these rules work in practice, consider the sketch in Figure 2b for the lrs method mentioned earlier. (Section 5 details our benchmark creation process.) This method (line 30) returns a TreeSet of the longest repeated substrings. The input to lrs is the fields n, t, lcp, and sa (line 28), which are initialized elsewhere (code not shown) and contain, respectively, the target string's length, contents as integers, longest common prefixes, and a suffix array of the string.

The body of lrs begins with local variable initialization, followed by some loops and conditionals. We discuss our procedure for developing this synthesis problem in detail in Section 5.2. For now, it is sufficient to know that lrs uses two generators: guard, which generates a boolean for use in a conditional guard, and stmts, which generates a sequence of statements. Note that, for clarity, the code in this example has been simplified from the actual benchmark.

The guard generator (line 44) uses JSketch's {| ... |} form, which selects among a set of expressions. Here there are various comparisons of holes to ints[0], among others not shown. Conceptually, ints[0] is just a local variable of lrs, but we store it in an array to make it easier to assign to in a generator, as we will see shortly.

```
@rewriteClass class TS<E> {
 @alg boolean add(E e);
 @alg void clear();
 @alg @pure boolean contains(Object e);
 @alg @pure int size();

 rewrite size(TS()) { return 0; }

 rewrite size(clear!(s)) { return 0; }

 rewrite size(add!(s, e)) {
  return contains(s, e) ? size(s)
                        : size(s)+1;
 }

 rewrite contains(TS(), e) {
  return false;
 }

 rewrite contains(clear!(s), e) {
  return false;
 }
 rewrite contains(add!(s, a), b) {
  return a.equals(b) ? true
                     : contains(s, b);
}}
```

(a) JLibSketch algebraic specifications for TreeSet. Type annotations in rules omitted for clarity.

```
class SuffixArray {
 int n; int[] t, lcp, sa;

 TS<String> lrs() {
  int[] ints = ...; ints[0] = 0;
  Object[] objs =...; objs[0]=new TS();
   ... // Initialize other local vars
  for (int i=0; i<n; i++)
   stmts(ints, objs);
  for (int i=0; i<n; i++) {
    if (guard(ints, objs)) {
     if (guard(ints, objs))
      stmts(ints, objs);
     stmts(ints, objs);
  } }
  return (TS<String>) objs[0];
 }
 generator boolean guard(...) {
  return {| ints[0]<??, ints[0]>?? ,...|};
 }
 generator void stmts(...) {
  if(??) ints[0]=... objs[0].size()...;
  if(??) ... objs[0].add(...)...;
   ... // additional statements
  if(??) stmts(ints, objs);
}}
```

(b) JSketch code (simplified).

```
harness public void testLRS() {
 String s = "abccdd";
 SuffixArray sa= new SuffixArray(s);
 TreeSet<String> lrss = sa.lrs();
 assert lrss.size() == 2;
}

/* lrss = add!(add!("c",[]),"d")
    lrss.size() =
     size(add!(add!("c",[]),"d"))
    ⇒ (contains(add!("c",[]),"d") ?
       size(add!("c",[])) :
       size(add!("c",[]))+1)
    ⇒* size(add!([], "c"))+1
    ⇒ (contains([],"c") ? size([]) :
       size([])+1)+1
    ⇒* size([])+1+1
    ⇒ 2 */
```

(c) TreeSet test harness.

```
class SuffixArray {
 int n; int[] t, lcp, sa;

 TS<String> lrs() {
  int max_len = 0;
  TS<String> lrss = new TS<>();
  char[] tmp = new char[n];
  for (int i=0; i<n; i++) {
    tmp[i] = (char) t[i]; }
  for (int i=0; i<n; i++) {
   if (lcp[i]>0 && lcp[i]≥max_len){
    if (lcp[i]>max_len) lrss.clear();
     max_len = lcp[i];
     lrss.add(new String(tmp,sa[i],
                 max_len));
  }}
  return lrss;
}}
```

(d) Synthesis solution (simplified).

Fig. 2. JLibSketch example from *SuffixArray* (Table 1). TS is short for Java's TreeSet class.

The stmts generator (line 47) uses holes to guard each possible statement it could generate. Thus, if a hole is synthesized as true, the statement it guards will be included, and otherwise it will not. We show two example statements. If enabled, line 48 assigns the result of calling TreeSet's size method (on objs[0], also conceptually a local variable of lrs) to integer ints[0]. Similarly, if enabled, line 49 calls TreeSet's add method. Further, stmts is recursive, so it can generate arbitrary sequences of statements, e.g., a sequence involving two calls to add.

Finally, to specify the correct behavior of lrs, we use the test suite that accompanied the original code. For example, Figure 2c gives a harness (a test case) testLRS, which creates a SuffixArray from a string, calls lrs, and asserts that the resulting set of longest repeated substrings has two elements. (The full harness contains more tests and assertions about lrs.) Hence, the synthesis problem is to find instantiations of guard and stmts—note each occurrence of a generator could be instantiated differently—such that the synthesized code passes the test case.

When JLibSketch performs synthesis on this example, it uses the rules in Figure 2a to give semantics to the TreeSet methods. For example, the comment at the bottom of Figure 2c shows the rewriting steps that occur for testLRS, assuming for simplicity that JLibSketch has reached the correct solution. Note that in reality, JLibSketch performs rewriting during synthesis, and thus the rewriting steps taken are control-dependent on holes and generators, and the data values in rewritten terms may also include holes and generators.

In Figure 2c, the call to lrs in the harness returns a TreeSet containing "c" and "d". That TreeSet is represented (line 60) as a term modeling the sequence of receiver-mutating method calls that would construct it. Then, when checking the assertion (line 57), the call to size rewrites to a conditional that checks if "d" was already added to the remainder of the TreeSet, per the third rewrite rule. Since it was not, this is rewritten to size(...)+1. Then this call to size is rewritten again to a conditional, and by the same process the term becomes size([])+1+1. This is simplified to 2 using the first rewrite rule.

The final synthesis solution is shown in Figure 2d. For clarity, we have replaced the local variable arrays with separate local variables with meaningful names.

In this particular example, using algebraic specifications has significant benefits. The algebraic specifications are about half the number of lines of code of mocks, and synthesis runs at least twice as fast with algebraic specifications compared to mocks. In our experiments, we found this trend holds in other cases as well. Another potential benefit is that using algebraic specifications to describe relationships among methods might be simpler than describing the methods themselves. For example, as mentioned in the introduction, we can write an algebraic specification such as $\mathsf{decrypt}(\mathsf{encrypt}(m, k), k) \Rightarrow m$ without needing to provide an actual implementation of encrypt or decrypt.

3 COMPILING ALGEBRAIC SPECIFICATIONS TO SKETCH

JLibSketch follows the same approach as JSketch, which works by compiling a JSketch problem into a Sketch problem, running Sketch on the result, and mapping Sketch's solution to the original JSketch input [Jeon et al. 2015b]. JLibSketch extends JSketch's compilation to also encode the rewriting rules for algebraic specifications.

To ease formal reasoning, we conceptually break JLibSketch's compilation process into two steps. The first step encodes JLibSketch into $\text{Sketch}^{+\text{lib}}_{\text{core}}$, a core language we introduce that is Sketch-like and has built-in support for algebraic specifications. More precisely, a $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program has the form $\mathcal{A}; \mathcal{H}; \mathcal{E}$, where $\mathcal{A}$ is a set of algebraic data type (ADT) declarations, $\mathcal{H}$ is a set of harness functions with holes and $\mathcal{E}$ is a set of rewrite rules. We omit the translation as it is similar to JSketch [Jeon et al. 2015b].

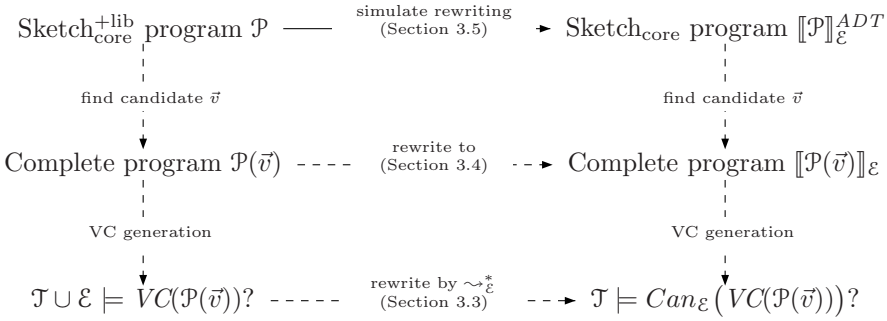

Fig. 3. Compiling Sketch$^{+\text{lib}}_{\text{core}}$ to Sketch$_{\text{core}}$.

The second step, which we focus on in this section, compiles a Sketch$^{+\text{lib}}_{\text{core}}$ program to Sketch$_{\text{core}}$, a core sublanguage of Sketch. Formally, we describe this step as a compilation $[\![\mathcal{P}]\!]^{ADT}_{\mathcal{E}}$, where $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$ is the original Sketch$^{+\text{lib}}_{\text{core}}$ program. We also simply denote it as $[\![\mathcal{H}]\!]^{ADT}_{\mathcal{E}}$ when the ADT declarations $\mathcal{A}$ is fixed. The superscript *ADT* indicates that the compiled program models algebraic specifications by rewriting Sketch ADTs [Inala et al. 2017], which are constructed terms that can be destructed by pattern matching. We present the major steps of this second compilation step and prove its soundness and completeness, which means we can solve Sketch$^{+\text{lib}}_{\text{core}}$ problems by compiling them to Sketch.

3.1 Sketch$^{+\text{lib}}_{\text{core}}$ to Sketch$_{\text{core}}$ Proof Overview

Figure 3 gives an overview of our argument that we can solve Sketch$^{+\text{lib}}_{\text{core}}$ problems by compiling them to Sketch$_{\text{core}}$ problems. The left-hand side of the figure shows a direct method for Sketch$^{+\text{lib}}_{\text{core}}$ synthesis, with no compilation. Given a Sketch$^{+\text{lib}}_{\text{core}}$ program $\mathcal{P}$ with holes (top left), we generate a candidate solution $\vec{v}$ for the holes of $\mathcal{P}$ (middle left). We then check whether the Sketch$^{+\text{lib}}_{\text{core}}$ program with no holes, $\mathcal{P}(\vec{v})$, is correct by generating a verification condition (VC) and checking it under the theory of integers $\mathcal{T}$ and the rewrite rules $\mathcal{E}$ (bottom left). If the VC holds, then $\mathcal{P}(\vec{v})$ is a valid solution (Theorem 3.3). Thus, we have a guess-and-check approach for Sketch$^{+\text{lib}}_{\text{core}}$ synthesis, which is the core of either a brute-force algorithm or counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama 2013]. However, this algorithm requires reasoning with the rewrite rules as part of VC checking.

The right-hand side of Figure 3 shows a parallel procedure in which the Sketch$^{+\text{lib}}_{\text{core}}$ program has been compiled to a pure Sketch$_{\text{core}}$ program, i.e., the solver need not reason about the rewrite rules separately. We show that this procedure—and therefore the compilation into Sketch$_{\text{core}}$—is correct. We proceed from the bottom right to the top right. First, in Section 3.3 (bottom right), we show that checking the synthesis condition under $\mathcal{T} \cup \mathcal{E}$ is equivalent to rewriting according to $\mathcal{E}$ and then checking the rewritten synthesis condition ($Can_{\mathcal{E}}(VC(\mathcal{P}(\bar{v})))$) under just $\mathcal{T}$. Next (middle row), in Section 3.4 we show that rewriting according to $\mathcal{E}$ can be lifted to the program level for a candidate solution $\mathcal{P}(\vec{v})$. Finally (top row), in Section 3.5 we show that rewriting according to $\mathcal{E}$ for a Sketch$^{+\text{lib}}_{\text{core}}$ program with holes can be simulated using abstract data type (ADT) operations in Sketch$_{\text{core}}$.

Section 3.2 begins with a formal description of both Sketch$^{+\text{lib}}_{\text{core}}$ and Sketch$_{\text{core}}$.

u : library value x: library variable f: library function
n : primitive value z : primitive variable
A : algebraic data type C : ADT constructor d : ADT variable

$$
\begin{array}{rcl}
\mathcal{P} & ::= & \mathcal{A}; \mathcal{H}; \mathcal{E} \\
\mathcal{A} & ::= & \text{adt } A \ = \ \text{C } \bar{A} \mid \mathcal{A}; \mathcal{A} \\
\mathcal{H} & ::= & \text{harness } (\bar{x}; \bar{z})\ S \mid \mathcal{H}; \mathcal{H} \\
S & ::= & \text{skip} \mid z := E \mid \text{assume}(B) \mid \text{assert}(B) \mid d := R \mid \bar{d} := \text{match}(d, R) \mid S; S \\
 & & \mid z := f(\bar{V}) \mid x := f(\bar{V}) \\
E & ::= & n \mid ?? \mid z \mid E + E \mid E - E \\
B & ::= & \text{true} \mid \text{false} \mid z < z \mid x = x \mid B \wedge B \mid B \vee B \mid \neg B \\
R & ::= & d \mid (\text{C } \bar{R}) \\
V & ::= & z \mid x \\
v & ::= & n \mid u \\
\mathcal{E} & ::= & \text{skip} \mid \text{rewrite } T \Rightarrow T' \mid \mathcal{E}; \mathcal{E} \\
T, T' & ::= & v \mid V \mid f(\bar{T}) \mid T + T \mid T - T
\end{array}
$$

Fig. 4. Syntax of $\text{Sketch}^{+\text{lib}}_{\text{core}}$ (the full syntax) and $\text{Sketch}_{\text{core}}$ (the non-highlighted part only).

3.2 $\text{Sketch}_{\text{core}}$ and $\text{Sketch}^{+\text{lib}}_{\text{core}}$ Syntax and Semantics

Figure 4 gives the syntax for our core languages. The highlighted portions are part of $\text{Sketch}^{+\text{lib}}_{\text{core}}$ only, which we discuss below. For simplicity, both languages have primitive values n that include only integers with a few operations (+, −, and <), but our approach extends in a straightforward way to more operators and a richer set of primitives.

In $\text{Sketch}_{\text{core}}$, a program $\mathcal{P}$ consists of a set of algebraic data type declarations $\mathcal{A}$, a set of harnesses $\mathcal{H}$, and an empty set of rewrite rules $\mathcal{E}$. The body of a harness is a sequence of statements. Assignments $z := E$ bind primitive variables z to expressions, which are either primitive values, holes ?? to be solved for, primitive variables, or sums or differences of expressions. The assume and assert statements behave as expected given a boolean expression with the usual connectives. Assignments $d := R$ bind ADT variables d to ADT terms. In statement $\bar{d} := \text{match}(d, R)$, d is an ADT variable and R is a pattern, i.e., an ADT term with a set of variables. If the term represented by d can be matched with the pattern R, every variable in R is mapped to a subterm of d. These subterms can be ordered by their positions in the pattern R (from left to right), and bound to a vector of ADT variables $\bar{d}$.

Note that $\text{Sketch}_{\text{core}}$ does not include branching or looping. Assuming all loops are bounded (which is the case in Sketch), these forms can be split into multiple straight-line harnesses using assume.

The synthesis goal [Solar-Lezama 2016] is to find a solution for the holes such that, for any values assigned to the harness parameters $\bar{x}$ and $\bar{z}$ (i.e., they are universally quantified), running the harness will not trigger any assertion failures. We will make this precise shortly.

Adding Algebraic Specifications. To extend $\text{Sketch}_{\text{core}}$ to $\text{Sketch}^{+\text{lib}}_{\text{core}}$, we introduce library functions f. Such functions may return either primitive values or library values u. Note that the latter never appear in the surface syntax, because they can only be created by calling library functions. For simplicity, we assume there is only a single *library type* returned by library functions.

We syntactically distinguish library variables x, which can hold values of the library type, from primitive variables, and we allow equality tests among library variables in conditional positions.

$$\textsc{Harness}\ \frac{Env' = [\bar{x} \mapsto \bar{u}; \bar{z} \mapsto \bar{n}]}{\langle \text{harness } (\bar{x}; \bar{z})\ , S\rangle \xrightarrow{Orcl} \langle Env', S\rangle} \qquad \textsc{Call1}\ \frac{\langle Env, \bar{V}\rangle \rightarrow \bar{v} \quad Env' = Env[z \mapsto [\![f(\bar{v})_{Orcl}]\!]]}{\langle Env, z := f(\bar{V})\rangle \xrightarrow{Orcl} \langle Env', \text{skip}\rangle}$$

$$\textsc{AssertT}\ \frac{\langle Env, B\rangle \rightarrow \text{true}}{\langle Env, \text{assert}(B)\rangle \xrightarrow{Orcl} \langle Env, \text{skip}\rangle} \qquad \textsc{AssertF}\ \frac{\langle Env, B\rangle \rightarrow \text{false}}{\langle Env, \text{assert}(B)\rangle \xrightarrow{Orcl} \mathit{ERROR}}$$

Fig. 5. Operational semantics of $\text{Sketch}^{+\text{lib}}_{\text{core}}$ (partial).

We then extend our statement language to include two assignments for library calls: $z := f(\bar{V})$ and $x := f(\bar{V})$, which bind primitive and library variables, respectively, to the result of a library call. The arguments to such calls are sequences of variables; values can be passed by first assigning them to a variable. Notice that we assume the programmer knows which calls return primitives and which return the library type. For example, in Figure 2, add returns the library type, and size returns a primitive. (We discuss this issue further in Section 4.)

Finally, we introduce rewrite rules of the form rewrite $T \Rightarrow T'$, where *terms* T and T' are patterns composed of primitive/library values, primitive/library variables, library calls, and primitive operations. We write $Vars(T)$ for the free variables of T, and we assume that if rewrite $T \Rightarrow T'$ is a rule, then $Vars(T') \subseteq Vars(T)$. We interpret such a rule to mean that any expression instantiating T is equivalent to the corresponding (simplified) expression instantiating T'.

To define the synthesis problem, we first need to describe how to run programs with rewrite rules. To keep our semantics simple, we assume the existence of an *oracle Orcl* that models library calls. More specifically, for every function $f(\bar{v})$, the oracle *Orcl* maps arguments $\bar{v}$ to the return value $[\![f(\bar{v})]\!]_{Orcl}$. The oracle must also be consistent with the rewrite rules:

Definition 3.1 ($\mathcal{E}$-model). An oracle *Orcl* is an $\mathcal{E}$-model if $[\![t]\!]_{Orcl} = [\![t']\!]_{Orcl}$ for all t and t' such that t can be rewritten to t' using rules in $\mathcal{E}$.

Now we can define a semantics for $\text{Sketch}^{+\text{lib}}_{\text{core}}$, part of which is shown in Figure 5. For harness bodies, our semantics proves judgments of the form $\langle Env, S\rangle \xrightarrow{Orcl} \langle Env', S'\rangle$, meaning in initial environment Env (a binding of variables to values), executing statement S with oracle *Orcl* yields environment Env' with remaining statement S' to be executed, without any assertion violation.[2] For example, Call1 uses the oracle to reduce a library call, and AssertT evaluates a valid assertion to skip. Assertion violations reduce to the special error term in AssertF. Finally (and with a slightly different judgment form), in Harness, evaluating a harness evaluates its body with arbitrary bindings for its arguments. A complete semantics can be found in Appendix A.

Given this semantics, we can now define the synthesis problem: given a $\text{Sketch}^{+\text{lib}}_{\text{core}}$, find a solution for the holes such that there are no assertion violations in any executions. Formally:

Definition 3.2 (Validity). A $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$ is *valid* if every hole in $\mathcal{H}$ can be assigned a value to form a harness $\mathcal{H}'$ with no holes such that for any $\mathcal{E}$-model *Orcl* it is the case that $\langle \epsilon, \mathcal{H}'\rangle$ does not reduce to *ERROR*, where ϵ is the initial, empty environment.

The synthesis problem, then, is to determine whether a program is valid according to the above definition by finding an assignment to its holes.

[2]We assume the statements are all well typed with respect to the ADT declarations $\mathcal{A}$, i.e., all ADT terms/patterns can be associated with an appropriate ADT type defined in $\mathcal{A}$.

Synthesis Condition. As a last step before proceeding with the proof, we argue that we can reason about validity by reasoning about VCs. Formally, let $\mathcal{H} = \mathsf{harness}\ (\bar{x}; \bar{z})\ S$ be a harness with holes $\bar{h}$.[3] From here on, we write such a harness as $S[\bar{x}, \bar{z}, \bar{h}]$ for compactness. Then for a $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$, we define the *synthesis condition* to be $\exists \bar{h}. VC(\mathcal{P}(\bar{h}))$, where $\bar{h}$ is the set of holes in $\mathcal{H}$ and $VC(\mathcal{P}(\bar{h}))$ is the verification condition

$$VC(\mathcal{P}(\bar{h})) \equiv \bigwedge_{S[\bar{x},\bar{z},\bar{h}] \in \mathcal{H}} \left(\forall \bar{x} \forall \bar{z}. \text{wp}(S[\bar{x}, \bar{z}, \bar{h}], \text{true})\right)$$

Here, $\text{wp}(S[\bar{x}, \bar{z}, \bar{h}], \varphi)$ is the (standard) weakest precondition of S with respect to φ. (See Figure 11 in Appendix A.) Also, when plugging values for $\mathcal{P}$'s holes into S we implicitly ignore values for any holes not in S.

We can show that satisfying the synthesis condition is equivalent to validity:

Theorem 3.3. *Let $\mathcal{T}$ be the theory of integers (the primitives in Sketch_{core}). For any $\text{Sketch}^{+lib}_{core}$ program $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$, the program $\mathcal{P}$ is valid if and only if $\mathcal{T} \cup \mathcal{E} \models \exists \bar{h}. VC(\mathcal{P}(\bar{h}))$.*

Here $\mathcal{T} \cup \mathcal{E} \models \phi$ means that ϕ holds assuming the theory $\mathcal{T}$ and the rewriting rules in $\mathcal{E}$.

3.3 Rewriting the Verification Condition

We now show that checking the VC with $\mathcal{E}$ is equivalent to checking the VC without $\mathcal{E}$ after applying rewriting (under some assumptions).

The rewriting rules in $\mathcal{E}$ operate on terms T. We can lift these rules in a straightforward way to verification conditions by simply applying the rules to the terms inside of a verification condition in the obvious way. Let $\varphi \rightsquigarrow_{\mathcal{E}} \psi$ be the one-step rewriting relation derived in this way that applies a single rewrite. Let $\varphi \rightsquigarrow^*_{\mathcal{E}} \psi$ be the reflexive, transitive closure of $\rightsquigarrow_{\mathcal{E}}$.

In general, applying these rewrite rules could be arbitrarily complex or could take arbitrarily long. But we found that the rules needed for our experiments (Section 5) satisfy two key properties:

Definition 3.4 (Termination). A set of rewriting rules $\mathcal{E}$ is terminating if there is no unbounded sequence $\varphi_1 \rightsquigarrow_{\mathcal{E}} \varphi_2 \rightsquigarrow_{\mathcal{E}} \varphi_3 \rightsquigarrow_{\mathcal{E}} \ldots$.

Definition 3.5 (Confluence). A set of rewriting rules $\mathcal{E}$ is confluent iff whenever $\varphi \rightsquigarrow^*_{\mathcal{E}} \psi$ and $\varphi \rightsquigarrow^*_{\mathcal{E}} \psi'$, there exists θ such that $\psi \rightsquigarrow^*_{\mathcal{E}} \theta$ and $\psi' \rightsquigarrow^*_{\mathcal{E}} \theta$.

There are many ways to show these properties hold. For example, termination and confluence is guaranteed for *ordered and non-unifiable* rewriting systems. An ordered rewriting system has a *Lexicographic Path Ordering* (LPO), an ordering over the rewritten terms such that every rewriting rule rewrites from a higher ordered term to a lower ordered term. A unifiable rewriting system intuitively contains two rewriting rules such that applying one of them may potentially disable the application of the other rule. The formal definitions of LPO and unifiability can be found in Appendix B. Both orderedness and non-unifiability could be verified automatically, the former via enumeration and the latter via the Maude Church-Rosser tool [Durán and Meseguer 2010]. In our experiments, we found that manual verification of these properties was sufficient.

Theorem 3.6. *An ordered and non-unifiable rewriting system is terminating and confluent.*

Proof. See Appendix B. □

When a rewriting system is both terminating and confluent, exhaustively rewriting a formula deterministically produces a canonical form:

[3]We assume the surface syntax ?? has been converted into hole variables $\bar{h}$ drawn from a new alphabet.

Definition 3.7 (Canonical Form). Given terminating and confluent rewriting rules $\mathcal{E}$ for any formula φ, there is a unique formula ψ such that $\varphi \rightsquigarrow_{\mathcal{E}}^{*} \psi$ and ψ cannot be rewritten anymore. We call ψ the canonical form of φ and denote it as $Can_{\mathcal{E}}(\varphi)$.

Finally, we can transform away reasoning about library functions by using the canonical form, which is unique and deterministically computable:

Theorem 3.8 (soundness and completeness of formula rewriting). *Let $\mathcal{E}$ be a terminating and confluent set of rewriting rules, then for any φ, $\mathcal{T} \cup \mathcal{E} \models \varphi$ if and only if $\mathcal{T} \models Can_{\mathcal{E}}(\varphi)$.*

Proof. See Appendix C. □

3.4 Compiling Complete $\text{Sketch}^{+\text{lib}}_{\text{core}}$ To $\text{Sketch}_{\text{core}}$

In Section 3.3 we showed that canonical rewriting allows for checking of the VC without $\mathcal{E}$. Next we lift canonical rewriting to the program level via compilation for *complete* $\text{Sketch}^{+\text{lib}}_{\text{core}}$ programs, i.e., $\text{Sketch}^{+\text{lib}}_{\text{core}}$ programs with no holes.

We define a compiler $[\![\_]\!]_{\mathcal{E}}$ from $\text{Sketch}^{+\text{lib}}_{\text{core}}$ to $\text{Sketch}_{\text{core}}$ that takes the following steps:

(1) Normalize $\mathcal{P}$ to a Single Static Assignment (SSA) form, i.e., for each assignment in the program, introduce a fresh new variable for each update to a variable, and rename all uses of this variable accordingly.
(2) For each statement assume(φ) or assert(φ), repeatedly replace every non-input variable with the right-hand side of the unique assignment for that variable, until all variables are input variables.
(3) For each remaining statement assume(φ) or assert(φ), replace it with assume($Can_{\mathcal{E}}(\varphi)$) or assert($Can_{\mathcal{E}}(\varphi)$).

We first prove the following lemma, which shows that this compilation process preserves the verification condition:

Lemma 3.9. *Let S be a complete harness in a $\text{Sketch}^{+lib}_{core}$ program and $[\![S]\!]_{\mathcal{E}}$ be the harness compiled from S. Then $Can_{\mathcal{E}}(wp(S, true)) \equiv wp([\![S]\!]_{\mathcal{E}}, true)$.*

Proof. After the first two steps, any φ in a assume/assert statement does not contain any non-input variable, and hence is not affected by any regular statement in the program. In other words, wp(S, true) is a boolean combination of formulas from assume/assert statements. As the third step has rewritten every assume/assert condition to its canonical form, wp($[\![S]\!]_{\mathcal{E}}$, true) is just $Can_{\mathcal{E}}(\text{wp}(S, \text{true}))$. □

More importantly, when the $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program involves unknowns, the compilation also preserves solutions to the synthesis problem, i.e., if $\vec{v}$ is a solution to $\mathcal{P}$, it is also a solution to $[\![\mathcal{P}]\!]_{\mathcal{E}}$:

Theorem 3.10 (preserved solution). *Let $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$ be a $\text{Sketch}^{+lib}_{core}$ program with holes $\bar{h}$. Then for any candidate solution $\vec{v}$, it is the case that if $\mathcal{A}; [\![\mathcal{H}(\vec{v})]\!]_{\mathcal{E}}$; skip is valid, so is $\mathcal{A}; \mathcal{H}(\vec{v}); \mathcal{E}$, and vice versa if $\mathcal{E}$ is ordered and non-unifiable.*

Proof. Assuming $\mathcal{E}$ is ordered and non-unifiable, with the synthesis condition, $\mathcal{A}; \mathcal{H}(\vec{v}); \mathcal{E}$ is valid

$$\Leftrightarrow \mathcal{T} \cup \mathcal{E} \models \bigwedge_{S[\bar{x},\bar{z},\vec{v}] \in \mathcal{H}} \left(\forall \bar{x} \forall \bar{z}. \text{wp}(S[\bar{x}, \bar{z}, \vec{v}], \text{true})\right) \text{ (by Theorem 3.3)}$$

$$\Leftrightarrow \mathcal{T} \models \bigwedge_{S[\bar{x},\bar{z},\vec{v}] \in \mathcal{H}} \left(\forall \bar{x} \forall \bar{z}. Can_{\mathcal{E}}(\text{wp}(S[\bar{x}, \bar{z}, \vec{v}], \text{true}))\right) \text{ (by Theorem 3.8)}$$

$\Leftrightarrow \mathcal{T} \models \bigwedge_{S[\bar{x},\bar{z},\vec{v}] \in \mathcal{H}} \left(\forall \bar{x} \forall \bar{z}.\mathrm{wp}(\llbracket S[\bar{x},\bar{z},\vec{v}] \rrbracket_{\mathcal{E}}, \mathrm{true})\right)$ (by Lemma 3.9)

$\Leftrightarrow \llbracket \mathcal{H}(\vec{v}) \rrbracket_{\mathcal{E}}; \mathsf{skip}$ is valid (by Theorem 3.3). □

3.5 Simulating Rewriting Through ADT Operations

The compiler just described assumes we can compute the canonical form during compilation, but this is in general not possible if execution depends on unknowns. For example, let φ be the assertion $contains(add!(s,'a'), ??)$. If ?? is assigned $'a'$, then (using the rules in Figure 2a) $Can_{\mathcal{E}}(\varphi)$ is true. But if ?? is assigned $'b'$, then $Can_{\mathcal{E}}(\varphi)$ is $contains(s,'b')$.

We solve this problem by compiling the computation of $Can_{\mathcal{E}}(\varphi)$ as term rewriting that happens at synthesis time in $\text{Sketch}_{\text{core}}$. We do this by representing terms as ADTs and then simulating term rewriting by operations over ADTs. To do so we need a slightly more expansive version of $\text{Sketch}_{\text{core}}$ that includes ADTs, pattern matching, and conditionals. We omitted these from Figure 4 to keep the language compact; all these constructs have the expected semantics.

Figure 6 shows some of the generated ADT code. At the top of the figure, we create an ADT F representing library call terms that return non-primitive values. For the primitive type integer we create an ADT I for library calls that return integers, and similarly for booleans with ADT B. Notice that the latter two ADTs have extra constructors to simulate built-in connectives from integer arithmetic, as these connectives may also occur in the term to be rewritten. We denote these extra ADT declarations as $\mathcal{A}_{\mathcal{E}}$.

The bottom of the figure shows the rewrite function that encodes the rewrite rules. It works by performing a case analysis, with each case trying to match the pattern of the left hand side of a rule and replace it with the corresponding right hand side. The first set of tests handle rewrites to the top level of the term, and the switch statement handles nested rewrites. This particular rewrite function handles the cases that return the library type; rewriting the other types is similar (not shown), with the added caveat that they could return I and B ADTs or actual primitives, which we discuss further in Section 4.

This ADT embedding allows us to compile a $\text{Sketch}^{+\text{lib}}_{\text{core}}$ program $\mathcal{P}$ to a $\text{Sketch}_{\text{core}}$ program $\llbracket \mathcal{P} \rrbracket^{ADT}_{\mathcal{E}}$. The compilation is similar to the one we present in Section 3.4. The only difference is that in the last step, φ is compiled to a function call rewrite(φ) instead of $Can_{\mathcal{E}}(\varphi)$. We can then prove our final theorem, that compiling rewriting rules into ADT manipulations is sound and complete:

Theorem 3.11 (soundness and completeness of library synthesis). *Let $\mathcal{P} = \mathcal{A}; \mathcal{H}; \mathcal{E}$ be a $\text{Sketch}^{+lib}_{core}$ program with holes $\bar{h}$. Then for any candidate solution $\vec{v}$, if $\mathcal{A} \uplus \mathcal{A}_{\mathcal{E}}; \llbracket \mathcal{H} \rrbracket^{ADT}_{\mathcal{E}}(\vec{v})$; skip is valid, so is $\mathcal{H}(\vec{v}); \mathcal{E}$, and vice versa if $\mathcal{E}$ is ordered and non-unifiable.*

3.6 Supporting More Rewriting Rules

Recall that the completeness of our approach relies on the assumption that the rewriting rules are terminating. This assumption can be easily violated if a library function is associative and/or commutative. For example, the string concatenation function is associative: $concat(a, concat(b, c)) = concat(concat(a, b), c)$. If we add both directions as rewrite rules, rewriting will not terminate because the two terms can be rewritten back and forth ad infinitum.

Although we did not need to do so in our experiments, we can handle this case by treating two kinds of binary functions differently. Let Σ^{AI} be the binary functions that are associative and have an identity (there is a 0_f such that $f(0_f, a) = f(a, 0_f) = a$ for any a). Let Σ^{ACI} be the binary functions that are associative, have an identity, and are commutative (which can also lead to infinite rewriting through symmetry).

```
1  adt F = f1 τ1^1 ... τn^1
2        | f2 τ1^2 ... τn^2
3        | ... //for every library function f_i : τ1^i × ··· × τn^i → F
4  adt I = f_j τ1^i ... τn^i //for every library function or standard integer function f_j : ··· → I
5  adt B = f_k τ1^i ... τn^i //for every library function or standard boolean function f_k : ··· → B
6
7  F rewrite(F t) {
8      F[] V = match(t, α1(x̄)); //for every rule α_i(x̄) ⇒ β_i(x̄)
9      if (V ≠ ⊥) return rewrite(β1(V)); //rewrite α_i(V) to β_i(V); then continue rewriting recursively
10     F[] V = match(t, α2(x̄));
11     if (V ≠ ⊥) return ...;
12      ...
13     switch (t) {
14         case f1(x1, ..., xn): //for every composite case f (...)
15             return rewrite(rewrite(x1), . . ., rewrite(xn));
16         case f2:
17             return ...
18         default: return t;
19 }   }  ...
```

Fig. 6. Definition of the ADTs and the rewrite function.

To avoid infinite rewriting, we stipulate that different kinds of functions take different forms of arguments. A function $g \in \Sigma^{AI}$ takes a *string* of terms T as argument, denoted as $g(T)$. A function $h \in \Sigma^{ACI}$ takes a *multiset* of terms S as argument, denoted as $h[S]$. These special signatures flatten multiple, nested calls to the function to a single canonical method call, and avoid non-terminating rewriting due to the associativity or commutativity. The ADT encoding we presented in Figure 6 can then be straightforwardly extended to support these functions.

4 IMPLEMENTATION

JLibSketch is built on top of JSketch, which is written in Python and comprises roughly 6.9K lines of code (excluding the parser). JLibSketch is comprised of an additional 2.9K lines of code (again excluding the changes to the parser). We next discuss two key challenges in developing JLibSketch: implementing the rewrite function from Section 3 inside of Sketch, and introducing boxing to allow Java primitives and objects to mix where necessary.

Implementing JLibSketch Rewriting in Sketch. As presented in Section 3, JLibSketch works by encoding @rewriteClass methods using Sketch ADTs and simulating term rewriting by ADT rewriting. We illustrate the key ideas in Figure 7, which shows part of the Sketch code produced by JLibSketch for the example from Section 2. Note we have simplified the Sketch output for readability by using a simpler naming convention and eliding (object-oriented) dynamic dispatch.

For each @rewriteClass, JLibSketch adds a field to struct Object (line 10) —in this case _ts—to store instances of the @rewriteClass as a Sketch ADT. Lines 11–16 define the ADT for our example, giving three example constructors: TS, representing an empty TreeSet, and Add and Addb, representing the regular return value of a call to add and the mutated receiver after such a call, respectively.

JSketch translates each method into a Sketch function, where the first function argument is the receiver. Calls to @rewriteClass methods are translated into calls to JLibSketch-generated wrappers that construct new ADTs, apply rewriting rules, and then return the result.

For example, lines 29–31 show the function generated for size. This function calls rewrite_sz, part of which is shown on lines 33–39. rewrite_sz mirrors the rewrite function in Figure 6 by first

```
10 struct Object{int __cid ;...; TS _ts ;}
11 adt TS {
12  TS   { }
13  Add {TS self ; Object e ;}
14  Addb {TS self ; Object e ;}
15   ...
16 }
17
18 bit rewrite_contains (TS s, Object e1){
19  switch(s) {
20   case TS: return false ;
21   case Addb(s2, e2 ):
22    return equals(e1, e2) ? true :
23     rewrite_contains (s2, e1 );
24    ...}}
```

```
25 void addb(Object self , Object e) {
26  self . _ts=new Addb(self._ts , e );
27 }
28
29 int size (Object self ) {
30  return rewrite_sz ( self . _ts );
31 }
32
33 int rewrite_sz (TS s) {
34  switch(s) {
35   case TS: return 0;
36   case Addb(s2, e ):
37    return rewrite_contains (s2 ,e) ?
38     rewrite_sz (s2 ): rewrite_sz (s2 )+1;
39    ...}}
```

Fig. 7. (Partial) Compiled Sketch encoding of Figure 2.

performing case analysis on its argument to implement pattern matching. For example, if the argument is Addb(s2, e), then rewrite returns a value depending on whether rewrite_contains(s2,e), i.e., it implements the third rewriting rule in Figure 2a. Notice we recursively apply the appropriate rewrite function to any ADTs in the return value.

rewrite_sz is simpler than rewrite, as we make certain assumptions about our rewrite rules. For example, we are able to elide the switch statement from Figure 6 (lines 13-19) because JLibSketch performs eager rewriting, so subterms will already have been rewritten, and no rules in our experiments rewrite subterms as part of the rule. Similarly, we are able to elide the primitive ADTs B and I from Figure 6 as we assume rewriting of any ADT representing a primitive type produces an actual primitive.

@rewriteClass methods with side effects (those that are not @pure) are handled similarly, except each source-level call to such a method is translated into two Sketch calls, one to a function returning the method's return value and the other returning the mutated receiver. For example, the call to lrss.add(...) in Figure 2d, line 84 would be translated into a call to add and a call to the receiver-mutating function addb (line 25 in Figure 7), which updates the self parameter's _ts field to contain a new ADT representing the modified object. We need not call rewrite here, as no rewrites of addb were defined in the @rewriteClass. This encoding lets us use ADTs, which are immutable, to represent imperative computations.

Boxing Primitives and Arrays. In JLibSketch, algebraically specified library methods that return values could either produce some actual value by rewriting, or they could evaluate to an ADT until later combined with other method calls. Since ADTs are stored in the Object type (Figure 7), this works fine for library methods that return objects. However, in Java, primitives are not objects, and, for performance reasons, in JSketch, arrays are also not objects. Thus, there is no way in JLibSketch to declare a library method as returning either an ADT or a primitive/array depending on the call. This is similar

```
1 @rewriteClass
2 class Cipher {
3   ... @alg @boxedRet @boxedArg(1)
4  byte[] doFinal(byte[] text );
5
6  rewrite byte[] doFinal(Cipher c1,
7   doFinal(Cipher c2, byte[] text )){
8    return text ;
9 }}
```

Fig. 8. Simplified manual boxing example from PasswordManager.

Table 1. Client program summaries with LoC comparison for mocks vs. rewrites.

Program	LoC	Summary	Spec LoC	Mock LoC	(S/M) LoC
SuffixArray[1]	349	Sorted array of suffixes for a given string used for indexing of substring patterns.	225	399	0.56
HashMap1[2]	337	Custom hash map using bucketing.	33	110	0.30
HashMap2[3]	221	Different custom bucketed hash map.	41	122	0.34
PasswordManager[4]	384	Cryptographic password manager.	118	284	0.42
CipherFactory[5]	458	Crypto backend for web app.	120	329	0.36
Kafka[6]	309	Backend cryptographic component for tool that brings NuCypher to Apache Kafka.	166	323	0.51
EasyCSV[7]	484	Library for parsing comma-separated files.	89	152	0.58
RomList[8]	482	Backend configuration parser for game.	106	173	0.61
Comparator[9]	202	Line-by-line comparator for files.	247	307	0.80

[1,2] https://github.com/williamfiset/Algorithms
[3] https://github.com/mohamednabil00000/Hashing-Table
[4] https://github.com/azmy92/Crypto-password-manager
[5,8] No longer publicly available
[6] https://github.com/nucypher/kafka-as-module-oss
[7] https://github.com/dangeabunea/EasyCsv
[9] https://github.com/malgorzatazawojek/ComparatorOfTwoLists

to the problem faced by languages that support lazy evaluation, where thunks representing function application could appear anywhere [Ingerman 1961].

To address this issue, JLibSketch requires the user to annotate both positions that must be boxed so that primitives/arrays and Objects can co-occur, and positions that should be unboxed, allowing access to values prior to use. In practice, we found that many examples require no manual boxing, and even those that do often require only a few annotations. Moreover, Sketch's type system will complain if insufficient boxing is used, so requiring manual boxing creates no risk to correctness. We leave autoboxing to future work [Lindholm et al. 2016].

Figure 8 illustrates boxing using a simplified snippet of one of our crypto benchmarks. In this library doFinal is used both for encryption and for decryption. Thus—eliding details of keys and some other arguments—one call to doFinal returns an ADT, and two nested calls to doFinal returns a byte array (lines 6-9). Therefore, we add annotations @boxedRet and @boxedArg(1) to indicate the argument and return of doFinal should be boxed. Then in code where doFinal is called (not shown), we insert annotations to add and remove boxing, including @isBoxed for positions boxed values flow to, and @unbox to unbox boxed values.

5 EXPERIMENTS

We evaluated JLibSketch by comparing synthesis using algebraic specifications with synthesis using mocks on a set of Java programs. Our benchmark suite is comprised of nine client programs from GitHub, from three application domains: data structures, cryptography, and file manipulation. We modified each benchmark to turn it into a synthesis problem by introducing holes and generators (Section 5.2). The first three columns of Table 1 give the size, in terms of lines of code (LoC), and a brief description of each benchmark. LoC were determined using SLOCCount [Wheeler 2009] where LoC are for the JSketch/JLibSketch code, rather than the resulting Sketch encoding. We chose well-commented, well-organized benchmarks that ranged from 200–800 lines of code—to yield a reasonably sized synthesis problem—and that used standard libraries from our target domains.

5.1 Mocking and Algebraically Specifying Libraries

We created both mocks and algebraic specifications for the libraries used by our benchmarks. We restricted development to just those methods called by our benchmarks. For successful synthesis, the mocks and specifications need not be complete with respect to the library behavior, as long as

```
10 E[] set; int cap, sz, rs_mult;
11 boolean add(E e) {
12   if (contains(e)|| e == null) {
13     return false
14   } else {
15     set[sz]=e; sz++; check_sz();
16     return true;
17 }}
```

```
18 void check_sz() {
19   if(sz≥cap) resize ();
20 }
21 void resize () {
22   E[] ns = new E[cap*rs_mult];
23   for(int i=0; i<cap; i++) { ns[i]=set[i]; }
24   set = ns; cap *= rs_mult;
25   }
```

Fig. 9. (Partial) TreeSet model code for adding an element.

they are complete with respect to the requirements of the synthesized program. In general, one might not know which methods are sufficient for synthesis a priori, however this can be learned by incrementally including methods until synthesis succeeds. We were careful to ensure that mocks and algebraic specifications implemented the same API functionality.

We developed algebraic specifications based on the methods' documentation, which was quite clear for our target libraries. We have already seen some example algebraic specifications in Figure 2a.

To verify rewriting termination, we confirmed the rules are ordered, i.e., there is no way to rewrite a term to itself by applying a sequence of rules and checked that the rules are non-unifiable, i.e., there are no cases in which either more than one rule could be triggered or in which one rule's pattern could overlap another. (Recall that these two conditions, orderedness and non-unifiability, are sufficient to prove termination, as shown in Theorem 3.6.) For example, in Figure 2a, the sixth rule matches contains(add!(...)). There are no rules with add!(...) on the outside, and no rules with contains on the inside (nested in another constructor), so applying the former rule cannot disable any other rule. Thus, the rules are non-unifiable.

To develop mocks, we looked at the same documentation and wrote simple, straightforward implementations that were sufficient (i.e., the program can successfully synthesize) and efficient (i.e., we spent some time making sure the mock performed well). For example, Figure 9 gives a mock for TreeSet's add method (line 11). The class has four fields (line 10): set, an array containing the elements, stored contiguously starting from index 0; sz, the number of elements stored in the array; cap, the maximum possible size array; and rs_mult, the resizing multiplier. When adding an element, duplicates are skipped (line 12), and if the array fills to capacity (line 18) then it is reallocated (line 21). Notice that while this code is straightforward, we were forced to make some implementation decisions (using an array) that introduce complexity (resizing when the array is full).

For test harnesses, we used the test suite for each benchmark. Recall these test harnesses serve as our specification; that is, any program synthesized is guaranteed to be correct with respect to the test suite. Some examples had incomplete or lacking test suites, so we added additional harnesses to cover more methods and functionality. In practice, we found the additional harnesses were simple extensions of existing tests.

Size Comparison. Since we co-developed JLibSketch with the algebraic specifications and mocks, we cannot make a direct comparison of effort required for each approach. However, we do observe a significant difference in size, which may be a proxy for effort. The last three columns of Table 1 give the LoC for the specifications and mocks required for each program, along with their ratio. From the table, we see that the algebraic specifications require fewer LoC for every example. The median difference across all examples is over 80 lines. In the worst case (*Comparator*), the algebraic

specifications are only 20% shorter, while in the best case (*HashMap1*), they are 70% shorter. For the *Comparator*, the relatively small difference is due to the difficulty of encoding sorting over ArrayList using rewrite rules; this is a case when rewrite rules are not a natural specification. In contrast, the large difference in LoC for *HashMap1* is due to complicated mocked ArrayList methods such as a resizing method (similar to Figure 9); such implementation choices are not required with algebraic specifications.

5.2 Synthesis Evaluation

To develop our benchmarks, we systematically turned the programs in Table 1 into synthesis problems by applying a predetermined set of rules. The goal of this process is to yield challenging synthesis problems that are not tailored to suit a specific library modelling approach. The rules are as follows:

(1) Create an array for each local variable type. For example, the local variables max_len and lrss in Figure 2d (lines 75-77) are translated into the arrays ints and objs in Figure 2b. Recall that using arrays makes it easy for generators to assign to those variables.
(2) Create a function generator stmts that represents possible assignments to local variables and calls to void functions with side effects. We use type information to determine which assignments are possible.
(3) Create a generator guard that represents possible boolean values for conditional guards. The guard generator can return local booleans, the results of calling boolean-valued functions, comparisons, and conjunctions, disjunctions, and negations of such terms.
(4) Replace each block of non-control, non-return statements with a call to the stmts generator and each conditional guard with a call to the guard generator.
(5) Replace each non-void return statement with a return of a generator for the appropriate type. For example, return 0 is replaced by return genInt(ints, objs), where genInt operates similarly to guard for integers instead of booleans.

For some of our benchmarks, this process creates very large synthesis problems that are well beyond the power of JLibSketch. To keep the problems tractable, we applied simplifications as needed to reduce the search space. Our simplifications mostly consisted of removing recursive generator calls. For instance, we first incrementally replaced function call argument generators with actual values. If more simplification was needed, we incrementally replaced generators for method call receivers. In one benchmark (*Comparator*), we modified stmts to include multiple assignments per hole, to reduce the number of recursive calls to stmts.

The second group of columns in Table 2 shows the number of statements (*#S*) and guards (*#G*) necessary for successful synthesis as well as the approximate search space size. More precisely, (*#G*) is the number of calls to guard, while (*#S*) is at least the number of calls to stmts, since each call to the latter might generate multiple statements. The search space size is approximate because inlining of generators in Sketch is determined heuristically at runtime.

Sketch Performance Considerations. In running the benchmarks, we found that some Sketch command-line flags had significant performance effects. In particular, the loop unrolling bound (*U*) and function call inlining bound (*I*) affect all nine benchmarks. Additionally, the range of integers (*R*), the control bit bound (*C*), and use of adaptive concretization [Jeon et al. 2015a] (*A*) affected performance of some of the benchmarks. Currently, there is no automated mechanism for determining these bounds, so we started with each set to one, adaptive concretization disabled, and no control-bit or integer bounds. We then incremented the bounds until synthesis succeeded. For each benchmark, the resulting bounds were the same for both mocks and algebraic specifications.

Table 2. Description of synthesis problem and experimental results.

Bench-mark	Synthesis Problem			Args					Mock		Alg. Spec	
	#S	*#G*	*Search space*	*U*	*I*	*A*	*R*	*C*	Median (s)		Median (s)	
SuffixArray	14	2	3.8×10^{59}	8	3	N	-	-	TO	–	**6602**	1151
HashMap1	7	1	5.4×10^{85}	5	1	N	43	-	186	57	**6**	0.5
HashMap2	18	4	2.2×10^{40}	2	1	N	8	2	397	358	397	348
PasswordManager	10	0	2.6×10^{33}	16	2	N	-	-	TO	–	**176**	17
CipherFactory	16	0	1.2×10^{21}	9	3	N	-	-	33	0.03	**4**	0.03
Kafka	18	0	3.1×10^{113}	35	2	N	-	-	TO	–	**472**	0.8
EasyCSV	12	1	1.2×10^{27}	5	3	N	-	-	4682	7	**1576**	2.1
RomList	7	1	1.9×10^{10}	26	2	Y	-	-	150	473	111	151
Comparator	7	0	2.8×10^{14}	10	2	Y	-	-	97	87	**42**	50

U–loop unrolling bound, I–function inlining bound, A–adaptive concretization R–int range
C–control bits, $\#S$–number of statements, $\#G$–number of guards, TO–timeout (>14,400s)

Runtime Performance. The last group of columns in Table 2 shows the runtime of synthesis, in seconds, for mocks and for algebraic specifications. All tests were performed using a 10-core Intel Xeon v4 CPU running at 2.2 GHz with 128 GB RAM. We ran each synthesis problem 31 times with mocks and 31 times with algebraic specifications; the table gives the median time and, in a small font, the semi-interquartile range (SIQR). We set a timeout of 4 hours (14400 seconds) for all problems. We manually verified that each synthesized solution was correct.

We observed that three benchmarks timed out for mocks but succeeded with algebraic specifications, sometimes in only a few minutes (*PasswordManager* and *Kafka*). Note that it was always the case that either all runs of a particular experimental condition finished, or all runs timed out.

To compare performance of the remaining six benchmarks, we used the non-parametric Mann-Whitney U-Test [Mann and Whitney 1947] to compare the distribution of results for mocks versus algebraic specifications. We found that, with greater than 95% confidence, synthesis with algebraic specifications was statistically significantly faster than with mocks for four of the remaining six programs (indicated in bold), and was the same for the other two programs.

While it is difficult to understand the causes of performance differences in the experiments, we did identify several trends. In the cryptographic examples, implementation choices in mocks can have a notable performance effect, e.g., replacing a shift cipher mock with the identity function sped up synthesis of *CipherFactory* by almost 20%. We also noticed that bounding the range of integers can significantly improve performance. Without such a bound, *Hashmap1* and *HashMap2* do not solve within the timeout for both mocks and algebraic specifications. Additionally, we found that while adaptive concretization was needed for *RomList* and *Comparator*, it introduces significant performance variance. Notice that the SIQRs for those programs are larger than or near the median.

Overall, in our experiments, synthesis with algebraic specifications is faster than with mocks for seven of nine benchmarks, and performs the same for the other two benchmarks. Moreover, the median performance increase ranges from 2× to more than 81×, and on three examples algebraic specifications finish where mocks time out. Thus, our results suggest that algebraic specifications can provide significant synthesis performance benefits over mocks for a range of programs.

5.3 Limitations of Algebraic Specifications

While our results are promising, algebraic specifications are not always better. Some libraries are not amenable to algebraic specification, e.g., it seems hard to specify the behavior of square root algebraically. However, since JLibSketch also allows mocks (they are just other methods in the sketch), it can still support such libraries. Additionally, reasoning about termination of rewriting (Section 3) might be harder than reasoning about looping in mocks. As discussed above, though, reasoning about orderedness and non-unifiability was straightforward for the libraries

we used. Further, unlike pre- and post-conditions, algebraic specifications cannot easily describe a method's semantics in isolation [Henkel and Diwan 2003; Henkel et al. 2007]. However, algebraic specifications have an advantage when combinations of methods are simpler to reason about, e.g., the combination of encryption and decryption mentioned earlier. Finally, algebraic specifications could potentially be implemented using mocks. However, this approach would be more cumbersome and would likely have worse performance because it would not take advantage of Sketch's ADT features.

Lastly, in our experience, both approaches are similar when it comes to debugging synthesis failures. We found it is often challenging to determine if such failures are due to the concrete parts of the sketch, the holes and generators, or the library models, either algebraic specifications or mocks. Indeed, improving debugging of synthesis is an interesting direction for future work.

Ultimately, we expect that neither algebraic specifications nor mocks is strictly better than the other, and that having both available in the same system, as they are in JLibSketch, will provide the most benefit.

5.4 Threats to Validity

There are several threats to the validity of these experiments. First, we evaluated one particular implementation each of both mocks and specifications, but there could be other variations that would affect performance. We leave further experimentation to future work, but we believe our implementation decisions for both mocks and specs are reasonable (see supplemental materials for all specifications and mocks). Second, the synthesis problems we chose might not be representative of real problems faced in the wild. We tried to address this concern by developing problems from a variety of programs from GitHub, written by a variety of authors and across three domains. Finally, while we believe our technique is general, we have only evaluated it within JSketch. We leave it as future work to extend other synthesizers with algebraic specifications.

6 RELATED WORK

Program Synthesis with Rewriting. Recently, Smith and Albarghouthi [2019] proposed using algebraic rewriting as part of program synthesis. In their approach, candidate solutions, which are generated separately and without using algebraic specifications, are rewritten to a normal form using the algebraic rules. If a candidate rewrites to a candidate that was previously seen, then there is no need to reverify it, which improves performance. In contrast, in JLibSketch, algebraic specifications are used to entirely replace modeling of libraries, and are used both during candidate generation and during verification.

Program Synthesis for Java. Several researchers have explored program synthesis in the context of Java. Prospect [Mandelin et al. 2005] finds call sequences matching a given type signature. Sypet [Feng et al. 2017] is similar, but also includes I/O examples as a harness. FrAngel [Shi et al. 2019] builds on SyPet, using angelic execution to find programs that satisfy test cases and then instantiating the angelic control-structure conditions for those programs. Unlike JLibSketch, none of these systems support template-based synthesis, nor do they support algebraic library specifications.

Algebraic specifications in SMT Solvers. An alternative to implementing algebraic specifications via encoding would be to add support for them directly in the solver. We leave developing such an approach in JLibSketch to future work. However, a few existing SMT solvers allow users to provide universally quantified axioms. We investigated whether these approaches might work for algebraic specifications.

Z3 supports universally quantified axioms using E-matching [de Moura and Bjørner 2007]. We wrote a small Z3 problem with an axiom similar to the third rule in Figure 2a. Unfortunately, Z3 fails to find a solution, and it is unclear to us whether there is a straightforward way to make the example work. Hence we believe Z3 cannot yet incorporate algebraic specifications in a practical manner.

We tried an equivalent problem with CVC4 [Barrett et al. 2011], which also fails to solve it. However, re-encoding that problem to use recursive functions [Reynolds et al. 2016] does yield a solution. The main issue with this encoding is that it does not work for *partial* algebraic specifications, because it solves a slightly different problem. In JLibSketch, in contrast to the recursive function encoding, library methods cannot have any properties beyond those in the specifications. For example, given a partial specification $\mathsf{decrypt}(\mathsf{encrypt}(m, k), k) \Rightarrow m$, the recursive function encoding could also assume decrypt and encrypt are the identity, which would not be allowed in JLibSketch. We leave it as future work to determine if there is another encoding in CVC4 that would be effective.

There are other approaches to handling quantifiers in SMT solvers. For example, Amin et al. [2014] propose two encoding systems for quantified SMT formulae that skirt the balance between efficiency and completeness, while Suter et al. [2011] present a semi-decision procedure to avoid the use of quantifiers in the solver altogether. Vazou et al. [2018] introduce refinement reflection so that a function's implementation is reflected into output refinement type specification. We leave exploring these approaches in our domain to future work.

Program Synthesis using Models. Other works propose alternative library models for synthesis. Singh et al. [2014] propose using manually created models, which are similar to mocks and have many of the same strengths and weaknesses. Lustig and Vardi [2013] synthesize LTL systems using libraries components specified using transducers, i.e., finite state machines with outputs. Gulwani et al. [2011] synthesize programs using a library described with logical relations among inputs and outputs. However, they only apply this approach to loop-free bitvector programs.

Algebraic Specification Inference. There has been some work in inferring algebraic specifications from code. Henkel and Diwan [2003]; Henkel et al. [2007] dynamically enumerate and test specification candidates for Java methods. In subsequent work [Henkel et al. 2008], they apply these specifications for program debugging by compiling the specifications to executable code. Several researchers suggest techniques to improve the performance of inferring axioms for API usage. Ghezzi et al. [2007] reduce the search space of dynamically inferring algebraic specification by creating a graph that captures ordering of function calls observed when running the program on sample inputs. Nguyen et al. [2009] generalizes this idea to infer API usage patterns. These approaches could potentially provide algebraic specifications to JLibSketch.

7 CONCLUSION

We introduced JLibSketch, a Java program synthesis tool in which library methods can be described with algebraic specifications. JLibSketch compiles its problems into Sketch problems in which library methods return ADTs. Algebraic specifications are compiled to functions that rewrite those ADTs. We formalized compilation and proved it sound and complete if the algebraic specifications are ordered and non-unifiable. We evaluated JLibSketch on nine synthesis problems from three domains. We found that, compared to mocks, algebraic specifications are on average half the size and, on seven of the nine problems, enable synthesis to be 2× to 81× faster. Thus, we believe JLibSketch takes an important step forward in making synthesis of programs with libraries more practical.

A SEMANTICS OF SKETCH$^{+\text{lib}}_{\text{core}}$

Figure 10 presents the big-step operational semantics for complete Sketch$^{+\text{lib}}_{\text{core}}$ programs (i.e., without unknowns) with respect to an oracle *Orcl* for library calls.

B TERMINATION AND CONFLUENCE

Definition B.1 (Ordered Signature). An ordered signature is $(\Sigma, \succ)$ with Σ a signature (Σ^P, Σ^F, a) and $\succ$ an ordering over $\Sigma^P \cup \Sigma^F$.

Definition B.2 (Lexicographic Path Ordering). Given an ordered signature $(\Sigma, \succ)$, the lexicographic path ordering $\succ_{lpo}$ is an ordering over Σ-terms defined recursively as follows:

- $\alpha \succ_{lpo} x$ if x is a variable in α and $x \neq \alpha$.
- $S \succ_{lpo} \alpha$ (or $T \succ_{lpo} \alpha$) if there exists an element β in S (or in T) such that $\beta \succ_{lpo} \alpha$.
- $\alpha \succ_{lpo} S$ (or $\alpha \succ_{lpo} T$) if for any element β in S (or in T), $\alpha \succ_{lpo} \beta$.
- $T \succ_{lpo} T'$ if there is a subsequence T'' of T such that T' can be obtained from T'' by replacing every element of T'' with a string of strictly smaller (w.r.t. lpo) elements.
- $S \succ_{lpo} S'$ if there is a subset S'' of S such that S' can be obtained from S'' by replacing every element of S'' with a multiset of strictly smaller (w.r.t. lpo) elements.
- $e(\alpha_1, \ldots, \alpha_n) \succ_{lpo} e'(\alpha'_1, \ldots, \alpha'_n)$ if
  - $e' \succ e$ and for some i, $\alpha_i \succ_{lpo} e'(\alpha'_1, \ldots, \alpha'_n)$;
  - $e \succ e'$ and $e(\alpha_1, \ldots, \alpha_n) \succ_{lpo} \alpha'_j$ for all j; or
  - $e = e'$ and $e(\alpha_1, \ldots, \alpha_n) \succ_{lpo} \alpha'_j$ for all j, and $\alpha_1.\cdots.\alpha_n \succ_{lpo} \alpha'_1.\cdots.\alpha'_n$ in the lexicographical order.

Lemma B.3. *Let $(\Sigma, \succ)$ be an ordered signature. We call a rewriting system $(\Sigma, \mathcal{E})$ ordered if we have $\alpha \succ_{lpo} \beta$ for every equational rule $\alpha = \beta$ in $\mathcal{E}$. An ordered rewriting system is terminating.*

Proof. As Σ is finite and $\succ_{lpo}$ is a reduction ordering as it is a LPO [Baader and Nipkow 1998]. $(\Sigma, \mathcal{E})$. By the definitioin of reduction ordering, $(\Sigma, \mathcal{E})$ is terminating [Dershowitz and Jouannaud 1990]. □

Definition B.4. A rewriting system $(\Sigma, \mathcal{E})$ is unifiable if there exist two equational rules $\alpha(\vec{x}) = \beta$ and $\alpha'(\vec{y}) = \beta'$ such that:

(1) either there is a substitution Θ that replaces every variable in $\vec{x}$ or $\vec{y}$ with a formula, and $\Theta(\alpha(\vec{x})) = \Theta(\alpha'(\vec{y}))$,
(2) or there is a symbol f occurs in the root position of $\alpha(\vec{x})$ and a non-root position of $\alpha'(\vec{y})$.

Proof of Theorem 3.6. Let $(\Sigma, \mathcal{E})$ be an ordered and non-unifiable rewriting system. By Lemma B.3, $(\Sigma, \mathcal{E})$ is terminating. Then by the Newman's Lemma [Huet 1980; Newman 1942], to prove the confluence of $(\Sigma, \mathcal{E})$, it suffices to show it is locally confluent, i.e., given two one-step rewritings $\varphi \rightsquigarrow_{\mathcal{E}} \psi$ and $\varphi \rightsquigarrow_{\mathcal{E}} \psi'$, there exists θ such that $\psi \rightsquigarrow^*_{\mathcal{E}} \theta$ and $\psi' \rightsquigarrow^*_{\mathcal{E}} \theta$. Notice that if the two one-step rewritings to ψ and ψ' are at two different positions φ, by Definition B.4, the two positions are not overlapped, i.e., either side-by-side or one position nested in a variable-position of another position. If the two rewritings are at the same position, only one equational rule can be applied and $\psi = \psi'$. Hence the rewriting system does not have any critical pair [Knuth and Bendix 1970] and therefore is confluent.

C COMPLETENESS

Before proving Theorem 3.8, we start with defining two terminologies:

Definition C.1 (Σ-sentence). Given a signature Σ, a Σ-sentence is a Σ-formula in which all occurrences of variables are bounded occurrences.

$$\textsc{Harness}\ \frac{Env' = [\bar{x} \mapsto \bar{u}; \bar{z} \mapsto \bar{n}]}{\langle \mathsf{harness}\ (\bar{x}; \bar{z})\ , S\rangle \xrightarrow{Orcl} \langle Env', S\rangle} \qquad \textsc{Val}\ \frac{}{\langle Env, n\rangle \to n}$$

$$\textsc{Var1}\ \frac{}{\langle Env, x\rangle \to Env(x)} \qquad \textsc{Var2}\ \frac{}{\langle Env, z\rangle \to Env(z)} \qquad \textsc{NegT}\ \frac{\langle Env, B\rangle \to \mathsf{true}}{\langle Env, \neg B\rangle \to \mathsf{false}} \qquad \textsc{NegF}\ \frac{\langle Env, B\rangle \to \mathsf{false}}{\langle Env, \neg B\rangle \to \mathsf{true}}$$

$$\textsc{Conj}\ \frac{\langle Env, B_1\rangle \to b_1 \quad \langle Env, E_2\rangle \to b_2}{\langle Env, B_1 \wedge B_2\rangle \to b_1 \wedge b_2} \qquad \textsc{Disj}\ \frac{\langle Env, B_1\rangle \to b_1 \quad \langle Env, E_2\rangle \to b_2}{\langle Env, B_1 \vee B_2\rangle \to b_1 \vee b_2}$$

$$\textsc{LessThan}\ \frac{\langle Env, E_1\rangle \to n_1 \quad \langle Env, E_2\rangle \to n_2}{\langle Env, E_1 < E_2\rangle \to n_1 < n_2}$$

$$\textsc{Plus}\ \frac{\langle Env, E_1\rangle \to n_1 \qquad \langle Env, E_2\rangle \to n_2}{\langle Env, E_1 + E_2\rangle \to n_1 + n_2} \qquad \textsc{Minus}\ \frac{\langle Env, E_1\rangle \to n_1 \qquad \langle Env, E_2\rangle \to n_2}{\langle Env, E_1 - E_2\rangle \to n_1 - n_2}$$

$$\textsc{Call1}\ \frac{\langle Env, \bar{V}\rangle \to \bar{v} \quad Env' = Env[z \mapsto [\![f(\bar{v})_{Orcl}]\!]]}{\langle Env, z := f(\bar{V})\rangle \xrightarrow{Orcl} \langle Env', \mathsf{skip}\rangle} \qquad \textsc{Call2}\ \frac{\langle Env, \bar{V}\rangle \to \bar{v} \quad Env' = Env[x \mapsto [\![f(\bar{v})_{Orcl}]\!]]}{\langle Env, x := f(\bar{V})\rangle \xrightarrow{Orcl} \langle Env', \mathsf{skip}\rangle}$$

$$\textsc{Assn1}\ \frac{\langle Env, E\rangle \to v}{\langle Env, z := E\rangle \xrightarrow{Orcl} \langle Env[z \mapsto v], \mathsf{skip}\rangle} \qquad \textsc{Assn2}\ \frac{}{\langle Env, d := R\rangle \xrightarrow{Orcl} \langle Env[d \mapsto R], \mathsf{skip}\rangle}$$

$$\textsc{Match}\ \frac{Vars(R)\{x_1, \ldots, x_n\}, \langle Env, \bar{d}\rangle \to R', R[r_1, \ldots, r_n] \equiv R'}{\langle Env, \bar{d} := \mathsf{match}(d, R)\rangle \xrightarrow{Orcl} \langle Env[d_1 \mapsto r_1, \ldots, d_n \mapsto r_n], \mathsf{skip}\rangle}$$

$$\textsc{Assume}\ \frac{\langle Env, B\rangle \to \mathsf{true}}{\langle Env, \mathsf{assume}(B)\rangle \xrightarrow{Orcl} \langle Env, \mathsf{skip}\rangle} \qquad \textsc{AssertT}\ \frac{\langle Env, B\rangle \to \mathsf{true}}{\langle Env, \mathsf{assert}(B)\rangle \xrightarrow{Orcl} \langle Env, \mathsf{skip}\rangle} \qquad \textsc{AssertF}\ \frac{\langle Env, B\rangle \to \mathsf{false}}{\langle Env, \mathsf{assert}(B)\rangle \xrightarrow{Orcl} \mathit{ERROR}}$$

$$\textsc{SeqL}\ \frac{\langle Env, S_1\rangle \xrightarrow{Orcl} \langle Env', S_1'\rangle}{\langle Env, S_1; S_2\rangle \xrightarrow{Orcl} \langle Env', S_1'; S_2\rangle} \qquad \textsc{SeqR}\ \frac{}{\langle Env, \mathsf{skip}; S\rangle \xrightarrow{Orcl} \langle Env, S\rangle} \qquad \textsc{SeqE}\ \frac{\langle Env, S_1\rangle \xrightarrow{Orcl} \mathit{ERROR}}{\langle Env, S_1; S_2\rangle \xrightarrow{Orcl} \mathit{ERROR}}$$

Fig. 10. Operational semantics of $\mathsf{Sketch}^{+\mathsf{lib}}_{\mathsf{core}}$.

Definition C.2 (Σ-theory). Given a signature Σ, a Σ-theory $\mathcal{T}$ is a set of Σ-sentences. We say a Σ-structure $\mathcal{A}$ is a $\mathcal{T}$-model if $\mathcal{A} \models \sigma$ for any $\sigma \in \mathcal{T}$.

S	$\text{wp}(S, \varphi) =$
skip	φ
$z := E$	$\varphi[z/E]$
$z := f(\bar{V})$	$\varphi[z/f^{ret}(x, \bar{V})]$
$x := f(\bar{V})$	$\varphi[x_1/f^{ret}(x_2, \bar{V})]$
assume(B)	$\neg B \vee \varphi$
assert(B)	$B \wedge \varphi$
$S_1; S_2$	$\text{wp}(S_2, \text{wp}(S_1, \varphi))$

Fig. 11. Weakest precondition.

Proof of Theorem 3.8. From right to left: For any model of $\mathcal{E}$, the definition of canonical form guarantees that every step in the rewriting from φ to $Can_{\mathcal{E}}(\varphi)$ is valid, i.e., $\mathcal{E} \models \varphi \leftrightarrow Can_{\mathcal{E}}(\varphi)$. Then if $\mathcal{T} \models Can_{\mathcal{E}}(\varphi)$, $\mathcal{T} \cup \mathcal{E} \models Can_{\mathcal{E}}(\varphi) \wedge (\varphi \leftrightarrow Can_{\mathcal{E}}(\varphi))$. By modus ponens, $\mathcal{T} \cup \mathcal{E} \models \varphi$.

From left to right: We prove by contradiction. Suppose M is a model of $\mathcal{T}$ violating $Can_{\mathcal{E}}(\varphi)$, we construct a model M' of $\mathcal{T} \cup \mathcal{E}$ violating φ.

Note that $\mathcal{E}$ is a finite set of quantified formulas. Then to find a model of $\mathcal{T} \cup \mathcal{E}$ violating φ, by the recent completeness result by [Löding et al. 2017], it suffices to find a model of $\mathcal{T} \cup \mathcal{E}_{\text{ins}}$ violating φ, where $\mathcal{E}_{\text{ins}}$ is the set of exhaustive quantifier instantiation of $\mathcal{E}$ with terms from φ. Now given a model M of $\mathcal{T}$, $\mathcal{E}_{\text{ins}}$ divides elements of M to equivalence classes. Each foreground element a in M belongs to the equivalence class $[m] = \{b \mid \text{there exists terms } t_a, t_b \in Terms(\varphi) \text{ such that } [\![t_a]\!]_M = a, [\![t_b]\!]_M = b, \text{ and } Can_{\mathcal{E}}(t_a)) = Can_{\mathcal{E}}(t_b))\}$. Then for each equivalence class, there is a unique canonical term t such that $[\![t]\!]_M$ belongs to the class. We use such $[\![t]\!]_M$ as the representative for each class $[m]$, denoted as $\text{Rep}(m)$. Then we can construct M' from M: the elements are $\{\text{Rep}(m) \mid m \in M\}$, and for every n-ary function f, $[\![f]\!]_{M'}(\text{Rep}(m_1), \ldots \text{Rep}(m_n)) = [\![f]\!]_M(m_1, \ldots, m_n)$.

Note that M' is a model of $\mathcal{T} \cup \mathcal{E}_{\text{ins}}$: on the one hand, M is a model of the background theory $\mathcal{T}$ and M' and M agree on the background sort, M' is a model of $\mathcal{T}$ as well; on the other hand, the construction of M' guarantees that M' satisfies all equations possibly used in rewriting φ, namely $\mathcal{E}_{\text{ins}}$. Moreover, if M violates $Can_{\mathcal{E}}(\varphi))$, so does M', as M' preserves all elements used in interpreting $Can_{\mathcal{E}}(\varphi))$. In conclusion, M' is a model of $\mathcal{T} \cup \mathcal{E}$ violating φ, which concludes the proof.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their helpful comments. This research was supported in part by the National Science Foundation under Grant Nos. CCF-1139021 and CCF-1837023.

REFERENCES

Nada Amin, K. Rustan M. Leino, and Tiark Rompf. 2014. Computing with an SMT Solver. In *Tests and Proofs*, Martina Seidl and Nikolai Tillmann (Eds.). Springer International Publishing, Cham, 20–35.

Franz Baader and Tobias Nipkow. 1998. *Term rewriting and all that.* Cambridge University Press, University Press, Cambridge, UK.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, Berlin, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and litmus tests. *ACM SIGPLAN Notices* 52, 6 (2017), 467–481.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, Vol. 51. ACM, New York, NY, USA, 775–788.

Luca Cardelli, Milan Češka, Martin Fränzle, Marta Kwiatkowska, Luca Laurenti, Nicola Paoletti, and Max Whitby. 2017. Syntax-guided optimal synthesis for chemical reaction networks. In *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 375–395.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Using program synthesis for social recommendations. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, Xue-wen Chen, Guy Lebanon, Haixun Wang, and Mohammed J. Zaki (Eds.). ACM, Hawaii, USA, 1732–1736. https://doi.org/10.1145/2396761.2398507

Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 383–401.

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. Elsevier, Cambridge, MA, USA, 243–320.

Francisco Durán and José Meseguer. 2010. A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In *Rewriting Logic and Its Applications*, Peter Csaba Ölveczky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–85.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* 52, 1 (2017), 599–612.

Carlo Ghezzi, Andrea Mocci, and Mattia Monga. 2007. Efficient recovery of algebraic specifications for stateful components. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, ACM, New York, NY, USA, 98–105.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/1993498.1993506

Johannes Henkel and Amer Diwan. 2003. Discovering algebraic specifications from Java classes. In *European Conference on Object-Oriented Programming*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 431–456.

Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2007. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering* 33, 8 (2007), 526–543.

Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2008. Developing and debugging algebraic specifications for Java classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 3 (2008), 14.

Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, New York, NY, USA, 162–171.

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4, Article 1 (Oct. 1980), 25 pages. https://doi.org/10.1145/322217.322230

Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. 2017. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–263.

P. Z. Ingerman. 1961. Thunks: A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations. *Commun. ACM* 4, 1, Article 1 (Jan. 1961), 4 pages. https://doi.org/10.1145/366062.366084

Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. IEEE, Austin, TX, USA, 156–167. https://doi.org/10.1145/2884781.2884856

Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015a. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Vol. 9207. Springer International Publishing, Cham, 377–394.

Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015b. JSketch: Sketching for Java. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE), Tool Demo Track*. ACM, Bergamo, Italy, Article 1, 4 pages.

D. E. Knuth and P. B. Bendix. 1970. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebras*, J. Leech (Ed.). Pergamon Press, Oxford, 263–297.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2016. *The Java Virtual Machine Specification, Java SE 8 Edition*. Pearson Education, Redwood City, CA , U.S.A.

Christof Löding, P. Madhusudan, and Lucas Peña. 2017. Foundations for natural proofs and quantifier instantiation. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec 2017), 1–30. https://doi.org/10.1145/3158098

Yoad Lustig and Moshe Y. Vardi. 2013. Synthesis from component libraries. *International Journal on Software Tools for Technology Transfer* 15, 5 (01 Oct 2013), 603–618. https://doi.org/10.1007/s10009-012-0236-z

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *ACM Sigplan Notices*, Vol. 40. ACM, New York, 48–61.

H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. http://www.jstor.org/stable/2236101

M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. http://www.jstor.org/stable/1968867

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/1595696.1595767

Andreas Raabe and Rastislav Bodik. 2009. Synthesizing hardware from sketches. In *2009 46th ACM/IEEE Design Automation Conference*. IEEE, San Francisco, California, USA, 623–624.

Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. 2016. Model finding for recursive functions in SMT. In *International Joint Conference on Automated Reasoning*. Springer, Berlin, 133–151.

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 73.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *Acm Sigplan Notices* 48, 6 (2013), 15–26.

Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular Synthesis of Sketches Using Models. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 395–414.

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, Berlin, Heidelberg, 24–47.

Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Armando Solar-Lezama. 2016. *The Sketch Programmers Manual*. MIT. Version 1.7.5.

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, Vol. 42. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1273442.1250754

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, New York, NY, USA, 136–148.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS '06*. ACM Press, San Jose, CA, USA, Article 1, 12 pages.

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 298–315.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *PLDI'14*. ACM, Edinburgh, UK, 530–541.

Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. 2015. Generation of Library Models for Verification of Android Applications. *SIGSOFT Softw. Eng. Notes* 40, 1, Article 1 (Feb. 2015), 5 pages.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *PACMPL* 2, POPL (2018), 53:1–53:31.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 62.

David Wheeler. 2009. SLOCcount. http://www.dwheeler.com/sloccount/