Bootstrapping Library-Based Synthesis

Kangjing Huang and Xiaokang Qiu(✉)

Purdue University, West Lafayette, IN 47907, USA
{huangkangjing,xkqiu}@purdue.edu

Abstract. Constraint-based program synthesis techniques have been widely used in numerous settings. However, synthesizing programs that use libraries remains a major challenge. To handle complex or black-box libraries, the state of the art is to provide carefully crafted mocks or models to the synthesizer, requiring extra manual work. We address this challenge by proposing Toshokan, a new synthesis framework as an alternative approach in which library-using programs can be generated without any user-provided artifacts at the cost of moderate performance overhead. The framework extends the classic counterexample-guided synthesis framework with a bootstrapping, log-based library model. The model collects input-output samples from running failed candidate programs on witness inputs. We prove that the framework is sound when a sound, bounded verifier is available, and also complete if the underlying synthesizer and verifier promise to produce minimal outputs. We implement and incorporate the framework to JSketch, a Java sketching tool. Experiments show that Toshokan can successfully synthesize programs that use a variety of libraries, ranging from mathematical functions to data structures. Comparing to state-of-the-art synthesis algorithms which use mocks or models, Toshokan reduces up to 159 lines of code of required manual inputs, at the cost of less than 40 s of performance overheads.

Keywords: Program synthesis · Libraries · Java · Program sketching

1 Introduction

Recent years have seen drastic progress in the development of constraint-based synthesis technology, made possible by the advances in formal methods and automated constraint solvers. The constraint solving based techniques guarantee that the synthesized program satisfies formal specifications and make synthesis algorithms much more scalable, stepping across domain-specific programming

G. Singh and C. Urban (Eds.): SAS 2022, LNCS 13790, pp. 272–298, 2022.
https://doi.org/10.1007/978-3-031-22308-2_13

tasks and applicable to general-purpose software development using practical, real-world languages, such as C/C++ [39,41], Python [34], OCaml [9], Java [14, 20,25,27], or JavaScript [32,37].

Toward using constraint-based synthesis to aid practical software development, a major challenge is synthesizing programs that use libraries, which is common in most real-world software. Note that state-of-the-art programming tools such as those for component-based synthesis [12,14,24,33,43] and unit-test generation [3,29] only need to run candidate programs (including the library) for testing. However, for constraint-based synthesis, the synthesizer has to symbolically reason about and analyze the libraries and generate client code that appropriately exercises library calls. The simplest solution to support libraries would be inlining—concatenate the library source code onto the synthesis problem and handle library methods just like other methods. Unfortunately, in practice, libraries are designed for flexibility and extensibility, making their code large and complex, and hence difficult for the synthesizer to use. For example, the Android platform, which contains more than 12 million lines of code [10], is too big to reason about for any existing synthesizer. Even worse, some libraries may contain native code, which is entirely out of reach of this approach.

To address this issue, a straightforward approach is to manually create *mock libraries*—short pieces of code at the appropriate level of abstraction such that the essential library functionality is implemented in a simple and analyzable way. This approach is adopted by state-of-the-art sketch-based synthesis tools [20,35]. While mocks can be effective, they require extra manual work as balancing the code simplicity and the accuracy of functional equivalence to the authentic implementation. For example, JSketch [20] mocks the Java standard library `java.util.TreeSet` using an object array, whose size can be either bounded or dynamically resizable. The former option is simpler but may fail to mimic the `TreeSet`'s behavior when too many objects are added; the latter option is observably equivalent to the `TreeSet` container but introduces extra complexity, which makes synthesis performance slow. Researchers have developed techniques to automatically create mock libraries [4,5,16,19] for program analysis or symbolic execution. However, these techniques focus on special classes of libraries and the generated mocks do not aim to aid program synthesis.

Another approach is to use non-executable specifications as *library models*. These models usually capture the essential properties of the library which can be leveraged by the synthesizer. For example, a critical property for a cryptography library is that any decryption after an encryption with the same key is the identity. In [25], this property is described as an algebraic specification: `decrypt(encrypt(m, k), k)` $\Rightarrow$ `m`. Library models are also developed and used in other state-of-the-art synthesis tools, in various novel ways [15,21,23,38,41]. While this approach is promising in both terms of simplicity and performance, it still requires extra manual work in writing the library specifications. This is actually the well-known *specification mining* problem for formal verification, which has been studied for many years [1,2,11,22]. Moreover, these models are hard to reason about automatically and need special treatment when integrated

into a synthesis tool. This limits the possible applications their approach may stretch to. For example, The rewriting-based encoding in JLibSketch [25] only handles library models that can be represented as equational axioms.

In this paper, we propose a new synthesis framework called Toshokan ("library" in Japanese) to support constraint-based synthesis algorithms for handling libraries. This framework takes an alternative approach to the problem, extending counterexample-guided inductive synthesis (CEGIS)—a standard inductive synthesis framework [40]—with an automatically built library model from logged behavior of the library. Intuitively, the proposed framework approximates the behavior of the library using a dynamic set of input-output samples, and guesses the output of the library when the input is not covered by any sample. In each CEGIS iteration, when the verifier rejects a candidate program and provides a witness input, a logger runs the failed candidate with the witness input. The witness execution exercises the library on some critical input and the logged input-output pair is added to the library sampling. As the CEGIS loop runs, more and more logs are gained and the sampling eventually becomes precise enough and allows the synthesis problem to be solved or rejected. Comparing with existing library-based synthesis approaches, Toshokan has the following advantages:

1. it does not require any extra manual work (except for the optional query function annotation as discussed in Sect. 4) like writing mock implementation or library models;
2. it synthesizes *provably-correct* programs using real Java libraries, whose correctness is guaranteed by an off-the-shelf verifier (currently JBMC); and
3. it allows the synthesizer to treat the library as a *black box*, making the task solvable using state-of-the-art Java sketching tool through careful encodings.

We give an overview of the Toshokan framework and elaborate how it works through an example in Sect. 2. We then in Sect. 3 formally describe the library-based synthesis problem, the major components of the framework, and the main synthesis algorithm, and prove its soundness and relative completeness. Then in Sect. 4, we embody the framework in the setting of sketch-based synthesis, and present the techniques used in the angelic inductive synthesizer, the centerpiece of the Toshokan framework, including three different library encodings. In Sect. 5, we discuss the design of the library logger, focusing on how we handle references and aliasing, termination and exceptions.

We implemented the Toshokan framework in JSketch [20]—a sketch-based Java synthesizer—and compared the new system with standard JSketch that supports user-provided models or mocks.[1] The results demonstrate that, Toshokan successfully synthesized correct code for all 11 benchmarks and saves the user from the extra manual work of writing library-abstracting mocks or models. Meanwhile, for most benchmarks, our performance is moderately slower than but still comparable to existing algorithms. More detailed experimental results can be found in Sect. 6.

[1] The validated artifact is available via DOI 10.5281/zenodo.7009051.

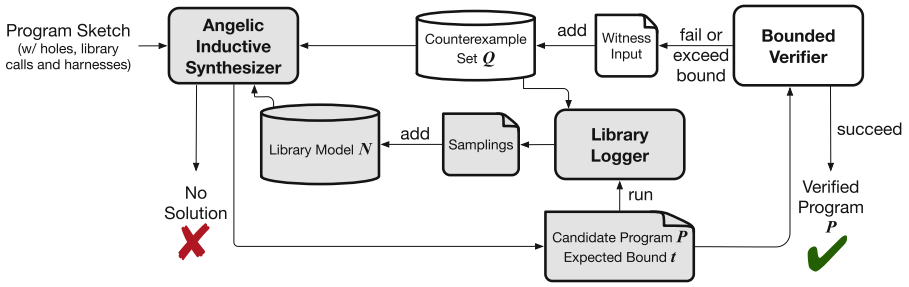

Fig. 1. Overview of the TOSHOKAN framework (distinct components in gray).

2 Overview

Figure 1 gives an overview of the TOSHOKAN framework, with the components distinct from standard CEGIS highlighted in gray. In addition to the standard components from normal CEGIS (verifier, synthesizer, counterexample set, etc.), the proposed framework features a *log-based library model* and a *library logger*. Whenever a candidate program P failed to be verified, the library logger takes P along with the authentic library (either source or binaries), runs the program on all existing counterexample inputs Q, and collects all observed library samplings, i.e., input-output pairs when the library is invoked. These samplings collectively form a library model N, which is an underspecification of the library, i.e., only partially covers the behavior of the library. The angelic inductive synthesizer (AIS) in TOSHOKAN takes N and determines the uncovered behavior of the library *angelically*. In other words, if the model does not cover a particular input, the AIS can determine the corresponding output arbitrarily. In each iteration, the AIS proposes a program P along with an expected bound t under which P should terminate on all inputs from Q; the bounded verifier checks whether P, along with the authentic library, satisfies the specification and terminates in t steps on all inputs. The whole synthesis process terminates when the verifier accepts the proposed program, or when the AIS concludes that there is no solution for the current counterexample set Q and library model N.

TOSHOKAN **by Example** We give a step-by-step illustration of TOSHOKAN's synthesis process through a simple JSKETCH example. Figure 2 shows the *gcd_n_numbers* benchmark (adapted from the Sketch source distribution [36]). The method MultiGCD.main purports to compute the greatest common divider (GCD) of five input integers, using Java standard library java.math.BigInteger.gcd[2] to compute the binary GCD. As a program sketch, the for-loop that calls gcd involves some unknown holes and choices (highlighted in the code) to be filled. Note that the authentic code for gcd is complicated and may even not be available as a black-box library. Hence the user has to provide

[2] The actual library operates BigInteger objects; for simplicity, we adapt the signature to handle int's.

```
class MultiGCD { /* synthesize algorithm for gcd of N numbers */
  harness void main(int[] nums) {
    int n = nums.length; assume n >= 2; ...
    int result = gcd(nums[0], nums[1]);
    for(int i = ??; i < {| n | n - 1 | n - 2 |}; i++)
      result = gcd({| result | nums[i] |}, {| result | nums[i] |});
    for(int i=0; i<N; i++) assert nums[i] % result== 0;
    for(int i=result+1; i <= nums[0]; i++) {
      bit divisible = 1;
      for(int j=0; j<N; j++) divisible = divisible && (nums[j] %i == 0);
      assert !divisible ;
    }
  }}
```

Fig. 2. JSketch example: *gcd_n_numbers*.

JSketch (or the underlying Sketch engine) a mock library or a library model (e.g., see [36]). Both require expertise and extra work from the user (16 and 20 LoC, respectively).

With Toshokan, the user does not need to write mocks or models anymore. Table 1 shows how Toshokan solves this synthesis problem in 4 iterations,[3] *without any user-provided artifacts*. In the initial iteration, the synthesizer proposes a random solution as the candidate program. Then the bounded verifier, which for this example is JBMC [8], checks whether the solution terminates in a fixed number of steps and satisfies all assertions on all inputs nums. The verifier reports a concrete input that violates assertions: num = {3,3,3,1,3}. Besides returning this witness input to the synthesizer, the most noteworthy thing is that Toshokan also runs the failed candidate on the witness input with the authentic gcd implementation, and logs the input-output samples of the all library calls. In this instance, the logger collects a sample gcd(3,3) = 3 and adds it to the library model N. This library model helps the synthesizer understand why the first candidate fails.

With the collected witness input and library sampling, the synthesizer proceeds to the second iteration and proposes a new candidate program. In this iteration, the verifier provides a new witness input num = {2,2,2,2,1}. This time, the logger runs the candidate program with both the current and the previous witness inputs, and collects two samples: gcd(1,3) = 1 and gcd(2,2) = 2. This process continues and collects new witness inputs and library samplings in each iteration, until the synthesizer finds the correct solution in the fourth iteration. The whole synthesis process finishes within 23 s (see full performance data in Sect. 6).

[3] Toshokan can actually solve this problem in 1 iteration (see Sect. 6); we use this 4-iteration run for illustration purpose.

Table 1. A TOSHOKAN run for the *gcd_n_numbers* problem in Fig. 2.

Iter#	Candidate Program (filling lines 5–6)	Witness Input	Collected Sampling
1	for(int i = 4; i ¡ n-1; i++) result = gcd(result, result);	nums = {3,3,3,1,3}	gcd(3,3)=3
2	for(int i = 2; i ¡ n-1; i++) result = gcd(num[i], result);	nums = {2,2,2,2,1}	gcd(1,3)=1, gcd(2,2)=2
3	for(int i = 3; i ¡ n; i++) result = gcd(num[i], num[i]);	nums = {3,3,2,3,3}	gcd(1,1)=1
4	for(int i = 1; i ¡ n; i++) result = gcd(num[i], result);	success	N/A

3 The TOSHOKAN Framework

In this section, we formally define the synthesis problem and introduce the main synthesis algorithm in TOSHOKAN. Note that the formalism we give in this section is purely semantical—it is agnostic to the syntax of the program and the implementation of the components it relies on. This allows us to present the key idea of TOSHOKAN framework in a succinct and general way. We will present in the next section more specifically, in the setting of sketch-based synthesis, how the synthesis problem is formulated and solved.

3.1 Libraries

Definition 1 (Library Signature). *A library signature is a pair* $\Sigma = (S, \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S})$*, where* S *is a set of sorts, and* $\{\Sigma_{w,s}\}_{(w,s)\in S^* \times S}$ *is an* $S^* \times S$*-indexed family of sets of symbols. We denote the set of all symbols by* $Funcs(\Sigma)$*.*

Definition 2 (Library). *For* $\Sigma = (S, \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S})$ *a library signature, a* Σ*-library* $L_\Sigma = \{L_f\}_{f\in Funcs(\Sigma)}$ *is a family of computable functions* $L_f : w \to s$ *for each symbol* $f \in \Sigma_{w,s}$*.*

Example 1. The library signature for the overview example is constituted by a single sort and a single function: $\Sigma = (\{\mathbb{Z}\}, \{\{\mathsf{gcd}\}_{\mathbb{Z}^2 \to \mathbb{Z}}\})$. In other words, the signature contains a single symbol gcd, which belongs to $\Sigma_{\mathbb{Z}^2 \to \mathbb{Z}}$. We denote the authentic Σ-library as $\mathsf{Real}_\Sigma = \{\mathsf{Real}_{\mathsf{gcd}}\}$, where $\mathsf{Real}_{\mathsf{gcd}} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ computes the binary greatest common divider.

Remark: Note that the library functions are defined to be computable and deterministic. This allows us to treat the library as a black box and make queries: providing concrete input values and asking what the output value is.

Handling Side Effect. Definition 2 considers pure library functions without side effects or multiple return values. This restriction does not affect the expressiveness of our framework as the definition is sufficient for encoding more complex libraries in real world. For example, we follow the idea of JLibSketch [25] to handle side effects. Given a Java class which maintains a complex internal state and contains methods that query and update the current internal state, every method in the class can be encoded to a pair of library function: both functions take the current state of the class as an extra argument; one gives the expected return value of the method and one gives the updated state of the class.

Example 2. Consider the class java.util.Stack in Java with an initializer init and two methods void push(int i) and int pop(). This class can be encoded to a library with four functions. The signature of the library is $\Sigma = \big(\{\mathbb{Z}, \mathsf{Stack}\}, \{\mathsf{pop}\}_{\mathsf{Stack}\to\mathbb{Z}}, \{\mathsf{pop!}\}_{\mathsf{Stack}\to\mathsf{Stack}}\big)$. A function $\mathsf{pop} : \mathsf{Stack} \to \mathbb{Z}$ captures the value returned from Stack.pop(). In addition, the side effects of the two methods can be represented as library functions $\mathsf{push!} : \mathsf{Stack} \times \mathbb{Z} \to \mathsf{Stack}$ and $\mathsf{pop!} : \mathsf{Stack} \to \mathsf{Stack}$.

Definition 3 (Sampling). *For Σ a library signature, a Σ-sampling is a family of sets $N_\Sigma = \{N_f\}_{f\in Funcs(\Sigma)}$ in which there is a finite set $N_f \subseteq_{fin} w \times s$ for each $f \in Funcs(\Sigma)$.*

Definition 4 (Consistency). *A Σ-sampling N is consistent with a Σ-library L, denoted as $N \prec_\Sigma L$, if for any $f \in Funcs(\Sigma)$ and any $(t, v) \in N_f$, $L_f(t) = v$.*

Example 3. In the overview example, as shown in Table 1, a Σ-sampling is maintained and expanded in each iteration of the synthesis process. After all four iterations, the sampling is $N = \{N_{\mathsf{gcd}}\}$ where $N_{\mathsf{gcd}} = \{(3,3,3),(1,3,1),(2,2,2),(1,1,1)\}$. By Definition 4, this sampling is consistent with the authentic library defined in Example 1, i.e., $N_{\mathsf{gcd}} \prec_\Sigma \mathsf{Real}_{\mathsf{gcd}}$.

3.2 The Library-Based Synthesis Problem

We next present the library-based synthesis problem, which is essentially tasked to find a correct program interacting with a known library. From the perspective of parametric programming, the space of candidate programs can be encode as a parameter and the underlying library can be given as another parameter. In other words, the synthesis problem can be represented as a parameterized program $\mathcal{P}[c, L](i)$ whose behavior is determined by the input i and two parameters: parameter c controls how to concretize $\mathcal{P}$ to a complete program; and parameter L is the concrete library that $\mathcal{P}$ calls. Once c and L are determined, $\mathcal{P}[c, L]$ becomes a complete program whose behavior is deterministic and verifiable.

The specification of the synthesis problem is also represented semantically by giving a validation condition. In other words, a program satisfies the specification if and only if all concrete runs of the program satisfy the validation condition.

Definition 5 (Validation Condition). *A validation condition for a parameterized program $\mathcal{P}$ is a family of formulae $\phi_\mathcal{P} = \{\phi^t_\mathcal{P}(c, L, i)\}_{t\in\mathbb{N}}$, in which each $\phi^t_\mathcal{P}(c, L, i)$ is satisfied if and only if running $\mathcal{P}[c, L]$ on input i terminates within t steps and satisfies the specification.*

Definition 6 (Library-Based Synthesis Problem). *A library-based synthesis problem is represented as a tuple $(\mathcal{P}, C, L_{\mathsf{Real}}, \phi_\mathcal{P})$ where $\mathcal{P}$ is a parameterized program, C is the space of parameters for $\mathcal{P}$, L_{Real} is the library used in $\mathcal{P}$, and $\phi_\mathcal{P}$ is a validation condition for $\mathcal{P}$. The synthesis problem is to find a value $\mathsf{ctr} \in C$ and a bound t such that for any input i, $\phi^t_\mathcal{P}(\mathsf{ctr}, L_{\mathsf{Real}}, i)$ is valid.*

Remark: Note that the synthesis problem only aims to produce programs verifiable in bounded steps (which can be implicitly enforced in $\mathcal{P}$ and/or $\phi_\mathcal{P}$). This is a common practice for modern synthesis tools [7,39,42].

3.3 Inductive Synthesis with Angelic Libraries

We solve the library-based synthesis problem set forth above using TOSHOKAN, an enhanced CEGIS framework as illustrated in Fig. 1. We now formally describe the angelic inductive synthesizer, the key component of the framework.

The Angelic Inductive Synthesizer (AIS), similar to a regular inductive synthesizer in the standard CEGIS loop, maintains a set of sample inputs, finds a candidate program that satisfies the specification at least for the sample set, and gives the candidate to a verification oracle for checking. The salient feature of the AIS is that it also maintains a library sampling N_Σ and ignores the exact behavior of the authentic library not covered by N_Σ. Inspired by angelic programming [6,13], the AIS *divines an angelic library* L_{Ang} that is consistent to N_Σ and guarantees the synthesized program satisfies the specification for L_{Real}. In other words, if executed with the authentic library, the synthesized program does not necessarily satisfy the specification, even if the input is restricted to a sample set. However, the CEGIS loop will collect more counterexamples and samplings in each iteration and guarantees the correctness of the final solution (see our synthesis algorithm later in this section).

Formally, given a validation condition $\phi_{\mathcal{P}}$, let the current input set and library sampling be Q and N, respectively, the synthesis task is to check the following second-order formula:

$$\Phi[\phi_{\mathcal{P}}, Q, N] \equiv \exists \mathsf{ctr}.\ \exists t.\ \exists L_{\mathsf{Ang}}.\ \left(N \prec_\Sigma L_{\mathsf{Ang}} \ \wedge \bigwedge_{\mathsf{inp} \in Q} \phi^t_{\mathcal{P}}(\mathsf{ctr}, L_{\mathsf{Ang}}, \mathsf{inp})\right)$$

where $\phi^t_{\mathcal{P}}$ is the validation condition for the synthesis problem. The following theorem states that any solution to the original library-based synthesis problem is also a solution to the inductive synthesis problem.

Theorem 1. *Given a library-based synthesis problem $\mathcal{L} = (\mathcal{P}, C, L_{\mathsf{Real}}, \phi_{\mathcal{P}})$, a set of inputs Q, and a library sampling N such that $N \prec_\Sigma L_{\mathsf{Real}}$, then if $\mathcal{L}$ has a solution ctr, it witnesses the validity of $\Phi[\phi_{\mathcal{P}}, Q, N]$.*

Proof. The solution ctr and the authentic library function L_{Real} witness the validity of $\Phi[\phi_{\mathcal{P}}, Q, N]$. First, as N is consistent with L_{Real}, $N \prec_\Sigma L_{\mathsf{Real}}$. Second, as ctr is a solution to $\mathcal{L}$, there is an integer t_0 such that running $\mathcal{P}[\mathsf{ctr}, L_{\mathsf{Real}}]$ on all inputs from Q terminates within t_0 steps, i.e., $\bigwedge_{\mathsf{inp} \in Q} \phi^{t_0}_{\mathcal{P}}[\mathsf{ctr}, L_{\mathsf{Real}}, \mathsf{inp}]$. Therefore $\Phi[\phi_{\mathcal{P}}, Q, N]$ is valid. □

An AIS just solves $\Phi[\phi_{\mathcal{P}}, Q, N]$ and returns the witnessing solution ctr and bound t. We define it below and discuss our approaches to developing it in Sect. 4.

Definition 7 (Angelic Inductive Synthesizer). *An angelic inductive synthesizer is a procedure that accepts queries of the form* **AIS**$(\phi_{\mathcal{P}}, Q, N)$ *where $\phi_{\mathcal{P}}$ is a validation condition, Q is a finite set of inputs, and N is a library sampling. If $\Phi[\phi_{\mathcal{P}}, Q, N]$ is valid, the procedure responds with a witnessing solution (ctr, t); otherwise it returns $(\mathsf{unsat}, 0)$.*

3.4 Verifier and Logger

While the angelic inductive synthesizer presented in Sect. 3.3 is complete (as illustrated in Theorem 1), it is not sufficient to solve the library-based synthesis problem: first, it only guarantees the correctness of the synthesized program on a finite set of inputs Q; second, the correctness of the synthesized program relies on an angelically chosen library L_{Ang}, which is not necessarily consistent with the authentic library L_{Real}. Therefore, our Toshokan framework requires two other components: a bounded verifier and a logger. We define them below.

The bounded verifier is slightly stronger than the standard one in a CEGIS framework: it promises to verify the correctness of the input program and its termination in bounded steps, or provide a counterexample. The logger runs a concrete program and collects the interaction with the underlying library.

Definition 8 (Bounded Verifier). *A bounded verifier is an oracle that accepts queries of the form* $\mathbf{BV}(\mathcal{P}, c, L, \phi_{\mathcal{P}}, t)$*, where* $(\mathcal{P}, c, L, \phi_{\mathcal{P}})$ *forms a library-based synthesis problem and* $t \in \mathbb{N}$ *is an execution bound, asking "Do all executions of* $\mathcal{P}[c, L]$ *terminate in* t *steps and satisfy the specification* $\phi_{\mathcal{P}}$*?" In other words, it checks the validation condition* $\forall i.\ \phi^t_{\mathcal{P}}(c, L, i)$*. If so, the oracle responds with a positive answer* $\top$*; otherwise it responds with a witness input* inp *such that* $\phi^t_{\mathcal{P}}(c, L, \mathsf{inp})$ *is invalid, i.e., the concrete execution of* $\mathcal{P}[c, L]$ *on* inp *does not terminate in* t *steps or violates the functional specification.*

Remark: While the **AIS** (cf. Definition 7) treats the library as an absolute black box, it becomes trickier for the bounded verifier—it can treat the library as a black box and do testing only, which can be very slow, or leverage the bytecode (or even source code if available) to make verification more symbolic and efficient.

Definition 9 (Logger). *A logger is an oracle that accepts queries of the form* $\mathbf{log}(\mathcal{P}, c, L, Q)$*, where* $\mathcal{P}$ *is a parameterized program,* c *is a control parameter, and* L *is a* Σ*-library,* Q *is a finite set of inputs, and runs program* $\mathcal{P}[c, L]$ *with every input from* Q*. The logger returns a library sampling* N *such that for any* $f \in \mathit{Funcs}(\Sigma)$*, a pair* $(t, v) \in N_f$ *if and only if one of the runs involves an invocation* $f(t)$ *to the library and returns value* v*.*

We will discuss more about the design and implementation of the logger in Sect. 5.

3.5 The Main Synthesis Algorithm

We are now ready to present the main synthesis algorithm for Toshokan, which is shown in Algorithm 1. The algorithm extends the classic CEGIS framework and leverages the three components we described above: **AIS** the angelic inductive synthesizer, **BV** the verifier and **log** the logger. The verifier and the logger repeatedly provide extra counterexamples and library samplings, respectively, to refine the inductive synthesis task.

input : A library-based synthesis problem $(\mathcal{P}, C, L_{\mathsf{Real}}, \phi_{\mathcal{P}})$
output: A solution ctr to the input problem, if any; otherwise $\perp$

```
1  def toshokan(P, C, L_Real, φ_P) :
2      Q, N, S ← ∅ // cex inputs, library sampling and checked solutions
3      ctr ← Init(C) // the control parameter, initially random from C
4      t ← Init() // bound of execution steps
5      repeat
6          w ← BV(P, ctr, L_Real, φ_P, t)
7          if w = ⊤ :
8              break
9          else:
10             S ← S ∪ {ctr},  Q ← Q ∪ {w} , N ← N ∪ log(P, ctr, L_Real, Q)
11             (ctr, t) ← AIS(φ_P, Q, N)
12     until ctr = unsat;
13     return ctr
```

Algorithm 1: Main synthesis algorithm for TOSHOKAN.

In addition to a set of witness inputs Q, it also maintains a library sampling N as an approximation/model of the authentic library L_{Real}. In other words, N is expanded along the synthesis/verification iterations but always consistent with L_{Real}. The algorithm starts from empty Q, empty N, a random solution ctr and an initial bound t. In each iteration, the **BV** checks whether the current ctr and t lead to a fully correct program $\mathcal{P}[\mathsf{ctr}, L_{\mathsf{Real}}]$ terminating in t steps (line 6). if so, the algorithm terminates and returns the solution ctr; otherwise, the failed solution is added to the set of checked solutions C, and the verification result, which is a new witness input w, is added to the set Q (line 10). Note that the **AIS** may not understand why the new witness input w violates the specification, because running $\mathcal{P}[\mathsf{ctr}, L_{\mathsf{Real}}]$ on w may involve calls to the library L_{Real} with arguments not covered by the current sampling N. To this end, the algorithm invokes **log** to run the program on all inputs in Q and record the behavior of the library (line 10). The newly generated sampling are added to N. Now with the updated S, Q and N, the algorithm asks the inductive synthesizer to generate a new solution and proceeds to the next iteration (line 11). If the synthesizer cannot find any more solution, the algorithm terminates and concludes that the synthesis problem is unsolvable.

Soundness and Completeness. We now discuss the soundness and completeness of the algorithm.

Theorem 2 (Soundness). *Given an input library-based synthesis problem $(\mathcal{P}, C, L_{\mathsf{Real}}, \phi_{\mathcal{P}})$, if Algorithm 1 terminates and returns a solution ctr, it is a solution to the synthesis problem. If the algorithm returns unsat, then the synthesis problem has no solution.*

Proof. If a solution ctr is returned by the algorithm, it must be produced by the **AIS** as a pair (ctr, t) and have passed the checking of the **BV**. Then by Definitions 6 and 8, ctr is indeed a solution to the input problem $(\mathcal{P}, C, L_{\mathsf{Real}}, \phi_{\mathcal{P}})$.

If the algorithm returns unsat, the last instance of $\mathbf{AIS}(\phi_{\mathcal{P}}, Q, N)$ has no solution. Then by Theorem 1, the input problem $(\mathcal{P}, C, L_{\mathsf{Real}}, \phi_{\mathcal{P}})$ does not have solution either. □

The completeness states that if the input synthesis problem is solvable, Algorithm 1 guarantees to produce a solution. We show that algorithm is *relatively complete*: if the underlying **BV** and **AIS** are both enumerative, then the whole algorithm is complete. Intuitively, **BV** and **AIS** are enumerative if they guarantee to provide the "minimal" witness input and candidate program, respectively. We next define the enumerative-ness and prove the relative completeness.

Definition 10. *A bounded verifier* **BV** *is* enumerative *if there exists a total ordering assigning a distinct natural number to each possible input of the parameterized program $\mathcal{W} : I \to \mathbb{N}$, such that for any invocation $\mathbf{BV}(\mathcal{P}, \mathsf{ctr}, L_{\mathsf{Real}}, \phi_{\mathcal{P}}, t)$, it returns a counterexample inp only if for any other input inp' such that $\mathcal{W}(\mathsf{inp}') < \mathcal{W}(\mathsf{inp})$, inp' is not a valid return value.*

Definition 11. *An angelic inductive synthesizer* **AIS** *is* enumerative *if there exists a total ordering assigning a distinct natural number to each control value $\mathcal{E} : C \to \mathbb{N}$, such that for any invocation $\mathbf{AIS}(\phi_{\mathcal{P}}, Q, N)$, it returns ctr only if for any other value ctr' such that $\mathcal{E}(\mathsf{ctr}') < \mathcal{E}(\mathsf{ctr})$, ctr' is not a solution to the AIS problem.*

Theorem 3 (Relative Completeness). *Let* **BV** *be an enumerative bounded verifier and let* **AIS** *be an enumerative angelic inductive synthesizer, then running Algorithm 1 with* **BV** *and* **AIS** *guarantees to produce a solution if the input library-based synthesis problem is solvable.*

Proof. If the synthesis problem is solvable, there exists a minimal solution ctr. Assume the algorithm does not produce a solution, then due to the soundness, the algorithm will not terminate and **AIS** will produce an infinite sequence of conjectured solution/bound pairs: $(\mathsf{ctr}_0, t_0), (\mathsf{ctr}_1, t_1), \ldots$ such that none of the ctr_i's is a solution. Now as ctr is the minimal solution and **AIS** is enumerative, we have $\mathcal{E}(\mathsf{ctr}_i) < \mathcal{E}(\mathsf{ctr})$ for all $i \geq 0$. Therefore there must exist a solution ctr_R appears in the sequence infinitely often. As ctr_R is not a solution, let inp_R be the minimal counterexample input and running inp_R on $\mathcal{P}[\mathsf{ctr}_R, L_{\mathsf{Real}}]$ terminates in t_R steps. As **BV** is enumerative, it is not hard to prove that there is a infinite subsequence $(\mathsf{ctr}_R, v_0), (\mathsf{ctr}_R, v_1), \ldots$ where $v_0, v_1, \ldots$ is strictly increasing. In other words, there must be a pair (ctr_R, v_m) proposed by the **AIS** and $v_m > t_R$. Therefore v_m is sufficiently large for **AIS** to know that ctr_R terminates and fails to satisfy the specification, then ctr_R will not be proposed again after (ctr_R, v_m). The contradiction concludes the proof. □

Remark: Note that enumerative verifiers and synthesizers are not uncommon in practice. For example, in Sketch, one can use the minimize keyword to enforce the synthesizer to fill holes with values as small as possible. Moreover, an enumerative verifier can be constructed from a regular verifier: once a witness input $i = \mathsf{inp}$

is found by the regular verifier, add an assumption $i < \mathsf{inp}$ to the program and rerun the regular verifier to find a smaller witness input; repeat the process until no more witness input can be found. The last found witness input in the process should be the minimal one.

4 Angelic Inductive Synthesis

So far we have overviewed the TOSHOKAN framework with the general library-based synthesis problem defined in a semantical way. In this section, we present our approaches to developing the angelic inductive synthesizer in depth, in the setting of program sketching. In other words, the representation of parameterized program and specification is concretized to a sketched program, and the synthesis task is to fill holes of a sketched program such that all assertions are satisfied. We first present a simple language for program sketching, then discuss three different ways to encode and solve the angelic inductive synthesis problem.

The TOSHOKAN Core Language. We instantiate the library-based synthesis problem (Definition 6) to JSKETCH, a Java sketching language [20]. Below we show how the **AIS** problem can be encoded for a TOSHOKAN core language. The language is similar to Sketch and allows us to reuse the JSKETCH-to-Sketch compilation [20] and the Sketch synthesis engine [39].

The syntax of the TOSHOKAN core language is presented in Fig. 3. Besides standard programming constructs covered by Sketch (e.g., assignments, conditionals and loops), this language also supports library function definitions and calls, which are shown as the highlighted portion of the syntax. The language describes a program sketch which begins with a list of library functions (the $\mathcal{L}$ part). A library function may take primitive or composite values as arguments and return an primitive or composite value. For each library function f used in the program, there exists a corresponding full, authentic implementation f_{Real}, which does not include any holes, assumptions or assertions, or calls to other library functions.[4] The second part of the sketch is the harness functions (or the $\mathcal{H}$ part). It may include constant holes of the form ??, choice of expressions of the form $\{| \cdot | \cdot |\}$, assumptions, assertions, and arbitrary calls to the library functions.

Intuitively, the synthesis task is to fill the unknown constants with values (assumed to be naturals) such that running the harness functions will not trigger any assertion failure before any assumption violation. Formally, let $\mathcal{P} = \mathcal{L}; \mathcal{H}$ be a program sketch in the TOSHOKAN core language, and let the number of holes in $\mathcal{H}$ be m, then this sketch characterizes a library-based synthesis problem as per Definition 6:

$$\left(\mathcal{H}, \mathbb{N}^m, \{f_{\mathsf{Real}}\}_{f_{\mathsf{Real}} \in \mathcal{L}}, \phi_{\mathcal{P}}\right)$$

[4] This limitation is not fundamental and can be generalized in the future. Without calls between library functions, the implementation of library logger becomes easier since logging instrumentation would only need to be done on client code.

$$
\begin{array}{rl}
& z, z_1, z_2 : \text{integer variable} \quad y, y_1, y_2 : \text{composite variable} \\
& k \in \mathbb{Z} : \text{int const} \quad f, g : \text{lib function} \quad h : \text{harness function} \\
\mathcal{P} & ::= \mathcal{L}; \mathcal{H} \\
\mathcal{L} & ::= \mathsf{library}\ f_{\mathsf{Real}}(\bar{z})\ S \mid \mathcal{L}; \mathcal{L} \\
\mathcal{H} & ::= \mathsf{harness}\ h(\bar{z})\ S' \mid \mathcal{H}; \mathcal{H} \\
S & ::= \mathsf{skip} \mid z = E \mid S; S \mid \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ B\ \mathsf{do}\ S \\
S' & ::= \mathsf{skip} \mid z = E' \mid \mathsf{assume}(B) \mid \mathsf{assert}(B) \mid z = f(\bar{V}) \\
& \quad \mid y = g(\bar{V}) \mid S'; S' \mid \mathsf{if}\ B\ \mathsf{then}\ S'\ \mathsf{else}\ S' \mid \mathsf{while}\ B\ \mathsf{do}\ S' \\
E & ::= k \mid z \mid E + E \mid E - E \\
E' & ::= E \mid ?? \mid \{\mid E' \mid E' \mid\} \\
B & ::= \mathsf{true} \mid z_1 < z_2 \mid y_1 = y_2 \mid B \wedge B \mid \neg B \\
V & ::= z \mid y
\end{array}
$$

Fig. 3. Syntax of the TOSHOKAN core language (the library-related part is highlighted).

where $\phi_{\mathcal{P}}$ generates bounded validation conditions from a concrete program and a concrete input, checking that the execution terminates (in a bounded number of steps) and satisfies all assertions. Formally, let ctr be the values filled to holes and let inp be the input to the harness, the validation condition can be formulated in the following form:

$$
\begin{aligned}
\phi^t_{\mathcal{P}}(\mathsf{ctr}, \mathsf{inp}) \ \equiv \ & \exists S_0 \ldots S_t . \exists Z_0 \ldots Z_t . \Big(\\
& \forall 0 \le j \le t . \mathit{Follow}(S_j, Z_{j+1}, S_{j+1}) \wedge \mathit{Exec}(Z_j, S_j, Z_{j+1}) \Big)
\end{aligned}
$$

The formula guesses a t-step run of the program, including the executed statement S_j and the valuation Z_j of the variables before the statement, for each step j. The predicate *Follow* checks that statement S_{j+1} follows statement S_j given the current valuation Z_{j+1}. The predicate *Exec* checks that running S_j with the current valuation Z_j will successfully yield the next valuation Z_{j+1}.

Direct Encoding for Libraries of Primitive Type. Now we have a library-based synthesis problem represented by the input sketch in the TOSHOKAN core language. Recall that a key step of our main synthesis algorithm is to solve the angelic inductive synthesis problem $\mathbf{AIS}(\phi_{\mathcal{P}}, Q, N)$ as described in Definition 7: given a library sampling N, guess an angelic library consistent with N and generate a candidate program that satisfies the specification $\phi_{\mathcal{P}}$ on the sample input set Q. Our approach is to represent this problem as another sketched program which does not contain library calls. As our core language is consistent with Sketch both in syntax and semantics, the problem can be directly solved by Sketch [39] or other synthesis engines.

We start from the simplest case: the library functions all take primitive arguments only and return primitive values—this is already sufficient for the overview example gcd_n_numbers. In this case, the angelic choices can be simply

represented as uninterpreted functions. Assuming $\mathcal{P}$ contains a library function int f(int u_1, ..., int u_m) among others, we encode the problem **AIS**$(\phi_{\mathcal{P}}, Q, N)$ to a program sketch as shown in Fig. 4. The function h is copied from the harness function in the original $\mathcal{P}$ which may involves unknown control holes to be synthesized, assumptions and assertions delimiting the behavior of the program, and calls to the library functions. The new harness function test simply takes the input for h and makes sure the input matches one of the sample inputs in set Q, then calls the real harness h. The library function f is implemented as follows: if the input $(u_1, \ldots, u_m)$ matches the one sample input $(s_1, \ldots, s_m)$ from the library sampling N, then return the corresponding output t; otherwise, return an angelic value from an uninterpreted function f_{Ang}. The uninterpreted function is arbitrary but guarantees the functionality, i.e., L_{Ang} always returns the same output with the same input.

```
int f_Ang(int u); // uninterpreted func for angelic choices, for each library func f
...
int f(int u_1, ..., int u_m) { //guessed angelic model, for each library func f
  if (u_1 = s_1 ∧ ··· ∧ u_m = s_m) return t; // for every sample input—output pair
       (s_1, ..., s_m, t) ∈ N_f
  ... return f_Ang(u); // if the input is not covered by N, make an angelic choice
} ...
void h(int i_1, ..., int i_n) { /* the original harness func */ }
harness test (int i_1, ..., int i_n) {
  assume ⋁_{inp∈Q} i == inp; //assume i is from the current witness input set
  h(i_1, ..., i_n); //run the original harness on i and check all assertions
}
```

Fig. 4. Direct encoding of **AIS**$(\phi_{\mathcal{P}}, Q, N)$.

Note that Fig. 4 assumes that function f takes integer parameter and returns integer values, but the encoding can be easily generalized to more primitive types supported by modern synthesizers. For example, Sketch has native support of synthesizing control parameters and uninterpreted functions of int and bit, as well as constant-sized arrays or nested arrays of primitive values.

Call-Tree-Based Encoding for Libraries of Composite Type. Now let us consider encoding libraries of composite type, i.e., the library function may take as argument or return values from user-defined, variable-size types, e.g., records, variable-size arrays, algebraic data types. While the direct encoding presented in Fig. 4 is straightforward and efficient and Sketch has native support for arrays, structs and algebraic data types, naturally extending this encoding to support composite types is not practically feasible for two reasons: first, for many real-world libraries (e.g., for encryption/decryption), the source code is not available and internal data is unknown; second, some libraries are implemented with

```
class Stack { ...
  public Stack() {...}
  public void push(int i) {...}
  public int pop() {...}
}
public class Main {
  static Stack main() {
    s = new Stack();
    s.push(1); s.push(2);
    int i = s.pop();
    assert i == 2; return s;
  } }
```

(a) Java program manipulating Stack.

```
int E_Stack = −1, E_push! = −2, E_pop! = −3;
int pop_Ang(int[] call_tree);
int pop(int[] call_tree) {
  ... return f_Ang(u); //direct encoding for
      primitive type
} ...
int[] main() {
  int[] s_tree = [E_Stack]; //s = new Stack()
  s_tree = [E_push!] + s_tree + [1]; //s.push(1)
  s_tree = [E_push!] + s_tree + [2]; //s.push(2)
  int i = pop(s_tree); //i = s.pop()
  assert i = 2; return s_tree ; }
```

(b) Encoded program.

Fig. 5. Example: call-tree-based encoding for Stack.

complex data structures (e.g., java.util.Stack is implemented as a dynamically resizable array), making the direct encoding inefficient.

To this end, we use a different, call-based encoding for libraries with composite type. The idea is to characterize the library's internal state using the call tree that creates the current value. We illustrate the call tree representation through the following example.

Consider a Java program that uses the Stack library (see Example 2), which is shown in Fig. 5a. The main function creates a Stack object s, computes an integer i through a sequence of method calls to s, and returns the updated s. While the exact representation of the returned object is hard/impossible to obtain, the object can be determined by an expression new Stack().push(1).push(2).pop(). This expression can be uniquely represented as a call tree as shown in Fig. 6. Furthermore, one can assign a unique number to every Stack-valued method. For example, in Fig. 6, init, push! and pop! are assigned -1, -2, -3, respectively. Then the call tree's Polish notation can be uniquely represented as an array (see the right hand side of Fig. 6).[5]

Based on this array representation of call trees, we encode library-using programs to array-manipulating programs. Intuitively, we maintain an array s_tree for each non-primitive value s used in the program, and every method call or object initialization m is simulated by a corresponding manipulation to the array: if a m updates s, then expand the call tree by extending s_tree accordingly; if m computes a primitive value from s, then follow the direct encoding and use an uninterpreted function m_{Ang} to make an angelic choice. Figure 5b shows the encoding of Fig. 5a: we use $[E_m]$ to represent the integer value encoding a method

[5] Here we assume all integer arguments are positive and use negative integers to represent methods. If negative integers are involved in the program, the array encoding has to have an extra bit to indicate a leaf node is a primitive value or a method call.

m, and the function main is generated by a line-by-line translation from the original Main method. Note that s.pop() returns a primitive value and hence is translated using the direct encoding as shown in Fig. 4. Figure 7 formally presents the call-tree-based encoding of method declarations calls, assuming there is a single composite type C and a single primitive type int.

Query-Based Encoding for Libraries with Query Functions. The call-tree-based encoding for composite type presented above has a potential scalability issue: the call tree grows unboundedly when more and more library calls are made. Therefore, the size of the corresponding array representation will quickly become larger than synthesis engines can handle, especially when library calls are involved in a loop.

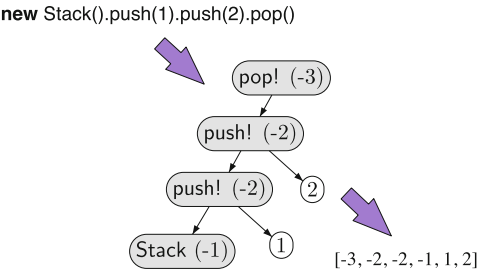

Fig. 6. Example: Polish notation of call tree.

We address this problem by using another query-based encoding when the library admits *query functions*, which are inspired by the state query methods proposed by Pei *et al.* [30]. Intuitively, query methods have no side effects and can be used to characterize the library class' internal mutable state. For example, consider a non-naive Java class SortedList defining a linked list data structure that would sort itself as new elements are added into it, as shown in Fig. 8a. The SortedList class contains two methods: insert and search. The search method is actually a query function—the internal sate and behavior of a SortedList object l is unique determined by l.search(i) for all possible input i. We

$$
\begin{aligned}
[\![C(\mathbf{int}\ \bar{z})\{\ldots\};]\!] &= \mathbf{int}\ E_C =< \text{fresh\_int} >; \\
[\![\mathbf{int}\ f(\mathbf{int}\ \bar{z})\{\ldots\};]\!] &= \mathbf{int}\ E_{f!} =< \text{fresh\_int} >;\ \ \mathbf{int}\ f_{\mathsf{Ang}}(\mathbf{int}[]\ \mathit{call\_tree}, \mathbf{int}\ \bar{z}); \\
&\quad\ \mathbf{int}\ f(\mathbf{int}[]\ \mathit{call\_tree}, \mathbf{int}\ \bar{z})\{< \text{direct encoding} >\} \\
[\![C\ g(\mathbf{int}\ \bar{z})\{\ldots\};]\!] &= \mathbf{int}\ E_{g!} =< \text{fresh\_int} >;\ \ \mathbf{int}\ E_g =< \text{fresh\_int} >; \\
[\![\mathbf{void}\ u(\mathbf{int}\ \bar{z})\{\ldots\};]\!] &= \mathbf{int}\ E_{u!} =< \text{fresh\_int} >; \\
[\![y := o.f(\bar{z});]\!] &= o_{tree} = [E_{f!}] + o_{tree} + \bar{z};\ \ y = f(o_{tree}, \bar{z}); \\
[\![q := o.g(\bar{z});]\!] &= o_{tree} = [E_{g!}] + o_{tree} + \bar{z};\ \ q_{tree} = [E_g] + o_{tree} + \bar{z}; \\
[\![o.u(\bar{z});]\!] &= \mathsf{o}_{tree} = [E_{u!}] + o_{tree} + \bar{z}; \\
[\![C\ o := \mathbf{new}\ C(\bar{z})]\!] &= \mathbf{int}[]\ \mathsf{o}_{tree} = [E_C] + \bar{z} \\
[\![\mathsf{skip}]\!] &= \mathsf{skip} \\
[\![S; S]\!] &= [\![S]\!]; [\![S]\!]
\end{aligned}
$$

Fig. 7. Call-tree-based encoding.

ignore the detection of query functions and assume the programmer manually marks query functions, using the `@query` keyword.

Given a library with query functions, we can solve the angelic inductive synthesis problem using a query-based encoding. We formally define query functions below:

Definition 12 (Query Function). *Let Σ be a library signature containing two sorts $\{P, C\}$ where P is primitive and C is composite. Then a Σ-library L_Σ admits a set of query functions $\mathcal{Q}$ if: 1) $\mathcal{Q} \subseteq \Sigma_{C \times P^* \to \{\mathsf{true},\ \mathsf{false}\}}$; and 2) For every non-query function $f \in \Sigma_{C \times \ldots \to C}$, and every $a, a' \in C$, if $f(a, \bar{b}) \neq f(a', \bar{b})$, a and a' are distinguishable by query functions, i.e., there exists a $g \in \mathcal{Q}$ and $\bar{e} \in P^*$ such that $g(a, \bar{e}) \neq g(a', \bar{e})$.*

Continuing on the SortedList example, Fig. 8b shows how this program is encoded. Note that the AIS is based on a finite set of inputs Q. Therefore, we can approximate the internal state of a SortedList object lst using a bit vector lst_query, which contains values search(inp) for every inp $\in Q$. When a new SortedList is created, the bit vector is initialized with all 0's as the search function always returns false. To update the bit vector, we expect the logger to invoke the query function search before and after each non-query function call, namely insert and insert!, and collect the inputs/outputs as a library sampling N. Based on N, all library functions are directly encoded in a way similar to Fig. 4.

In addition, for the search function itself, we encode it to an extra function search_Q. When search_Q is called for lst_query with input u, it essentially retrieves whether u matches any sample input covered by Q; if so, it simply returns the corresponding value in lst_query; otherwise it proceeds to the directly encoded search function.

5 The Logger

In this section, we discuss another major component of the Toshokan framework: the logger **log**, whose definitions have been given (in Definition 9). We discuss our design in the setting of program sketching to match the **AIS** design we described in Sect. 4. The design is mostly straightforward—simply run the candidate program with the current set of counterexample inputs Q, and for every library call it encounters, log the input and the corresponding output. Below we discuss some issues we identified and addressed in the design and implementation of the logger.

References and Aliasing. Real world libraries manipulate dynamically allocated structs and objects using references (pointers), which may be aliased or overlapped (e.g., two List references a and b such that a $\neq$ b but a.next == b.next). Therefore, a library function call will not only affect the references explicitly passed in as arguments, but also those aliased or overlapped with these arguments, which can be unboundedly many and cannot be tracked using library sampling.

```
public class SortedList {
  int value;
  SortedList next;
  SortedList ()
    { ... }
  @query public boolean
  /* query func annotation */
    search(int u)
    { ... }
  public boolean insert(int i)
    { ... }
  ...
}
public class Main {
  static void main(int i) {
    SortedList lst =
        new SortedList();
    assert !lst.search(i);
    lst.insert (i);
    assert lst.search(i);
  }
}
```

(a) Example: SortedList with query function annotated.

```
// direct encoding for all library functions
bit search_Ang(int u); bit insert_Ang(bit[n]
    q_result, int u);
bit[n] insert!_Ang(bit[n] q_result, int u);
bit search(bit[n] q_result, int u) { ... }
bit insert (bit[n] q_result, int u) { ... }
bit[n] insert!(bit[n] q_result, int u) { ... } ...
// query—based encoding for query functions
int[n] I_Stack = { 0, ..., 0 }
int[n] Q = { <current sample input set> }
bit search_Q(bit[n] q_result, int u) {
// if u is covered by Q return query result
  if (u == Q[0]) return q_result[0]; ...
  if (u == Q[n − 1]) return q_result[n − 1];
  return search(u);
//otherwise, use model from direct encoding
} ...
void main(int i) {
  bit[n] lst_query = I_Stack;
  assert !search_Q(lst_query, i);
  lst_query = insert!(lst_query, i);
  assert search_Q(lst_query, i);
}
```

(b) Query-based encoding for example SortedList.

Fig. 8. Example of SortedList and its corresponding query-based encoding.

To this end, we track library calls with arguments that are *aliased or disjoint only*. More concretely, we first extend the definition of library sampling (see Definition 3) and the logger in the following way. Assume Σ is a library signature containing a reference sort Ref and L is a Σ-library containing a function $f : \mathsf{Ref}^n \to \mathsf{Ref}$. Then an extended library sampling of f is a finite set of

$$N_f \subseteq_{\text{fin}} \mathcal{B}(n) \times \mathsf{Ref}^n \times \mathsf{Ref} \times 3^{\{0,1,\ldots,n\}}$$

where $\mathcal{B}(n)$ is the set of partitions of the set $\{1, \ldots, n\}$. The first element is a partition of the references into aliased equivalence classes; references from different equivalence classes must be disjoint. The second and the third elements are simply the input and output of the function. The last element indicates whether the output reference is aliased, overlapped, or disjoint with the n arguments.

Next, we also adapt the call-tree-based encoding for libraries. In addition to the encoding we presented in Fig. 7, the encoded program explicitly maintains the relationship between all references: aliased, disjoint, or overlapped. Whenever a reference is updated (via either an assignment or a library call), all aliased references will be updated in the same way and all disjoint references will be kept unchanged. For other overlapped references, as we don't track precise informa-

tion to updated them, they will be havoced, i.e., they will be updated arbitrarily, being disjoint or still overlapped with the updated reference.

Termination and Exceptions. Termination is a tricky issue for program analysis and verification, and also for our synthesis framework. Note that the input program to the logger is not necessarily terminating: it may invoke library calls infinitely often and the logger may not terminate either. In this case, the logger can set an execution limit T: if the execution reaches T steps, the logger just halts and returns the samples collected thus far. The limit T can be simply set as the integer t_0 found by the **AIS**—according to Definition 7 and the formula $\Phi[\phi_{\mathcal{P}}, Q, N]$ **AIS** solves, all sample runs of the synthesized program with the conjectured angelic library L_{Ang} guarantee to terminate within t_0 steps. In other words, if the real execution with the authentic library L_{Real} does not terminate within t_0 steps, the library behavior collected by the logger is already enough to distinguish L_{Real} and L_{Ang}.

Another similar issue is about the exceptions. While the Toshokan core language is simple, it may present exceptions such as division-by-zero and array index out-of-bound. More importantly, the library calls made by the candidate program also might be invalid and throw exceptions from running the authentic implementation of the library. In these cases, our logger simply returns the library samplings collected thus far and input to the last library call that causes the exception. These samplings will let the **AIS** know how the witness input leads to the exception so that the next candidate can avoid this scenario.

6 Evaluation

We implemented the Toshokan framework in JSketch [20]—a sketch-based Java synthesizer—and conducted experimental evaluation. It takes a JSketch file (intuitively a Java program with unknown constants, expressions, etc.) as input and produces a concrete Java program satisfying user-provided specification. Note that our goal is not outperforming vanilla JSketch using models and mocks [25]: the primary goal of Toshokan is to reduce the extra LoC that other methods require the user to write. Therefore, our evaluation attempts to answer the following research questions: **RQ1:** *Can* Toshokan *synthesize programs interacting with a wide variety of libraries?* **RQ2:** *Does* Toshokan *reduce the LoC that the synthesizer user needs to write with acceptable performance?* In this section, we first describe our implementation and benchmarks, then report the experimental results which answer the two research questions.

Implementation. The implementation was written in Rust and C++ with around 10k total LoC. We leverage the current frontend of JSketch to encode most Java-specific features to Sketch, and encode the angelic inductive synthesis problem as described in Sect. 4. Once a Sketch solution is obtained, we further leverage the decoder of JSketch to generate a concrete Java program candidate for verification and logging.

We employ JBMC [8] as the bounded verifier. Note that JBMC verifies compiled Java bytecode, i.e., it does not rely on the availability of the library's source code. JBMC also serves as the logger: if a candidate program failed verification, we build a Java program that explicitly runs the failed program with all witness inputs and finishes with assert false. JBMC will claim that the program is wrong and provide the trace of execution. We implemented a data extractor to collect input-output samples for the library calls involved in the trace. We remark that the JBMC-based logger is potentially unsound as JBMC uses its own library models.

Benchmarks. To evaluate Toshokan, we have adopted and converted a number of synthesis benchmarks from various sources, including Sketch modular synthesis benchmarks using function models [35], JDial benchmark using external libraries [17] and our own benchmarks using composite-type libraries. We also write some benchmarks ourselves by converting some well-known, widely used algorithms and data structures into sketch format with holes added, so that the benchmark set could be more diversified. We hope the wide range of the benchmarks adopted could help demonstrate the wide variety of libraries upon which our methods could be applied to.

Sketch modular synthesis [35] benchmarks are obtained from Sketch source repository as part of the project's experimental feature benchmarks [36]. These benchmarks come with two versions: one model version and one mock version for each synthesis task. The model version utilizes the function model features of Sketch to solve synthesis tasks with unknown library functions, as long as models of the functions are provided. The mock versions are effectively the same synthesis tasks, but with concrete implementations of the library functions. We excluded some benchmarks that are not legit (e.g., no synthesis task or no argument for the library function). This ends up in a total of 4 benchmarks adapted from this benchmark set. JDial [17] benchmarks are obtained from JDial-Debugger's Github repository [18]. Among the JDial direct manipulation benchmarks, we picked two out of three benchmarks that use external libraries (excluding *evalPoly_3* which seems to be identical to *evalPoly_2*). We also combined them into a larger one, *evalPoly_combined*.

Up to this point, all the benchmarks we adopted are using libraries of primitive types. To demonstrate the effectiveness of our methods handling libraries of composite types, we converted a number of well-known, widely used algorithms and data structures into sketch formats with added holes. This creates 4 new benchmarks, namely *stack_match*, *set_match*, *arraylist_match*, and *heap_sort*, all in encoding for libraries of composite types.

It ends up in a total of 11 benchmarks to be evaluated in experiments. Whereas the sizes of the client-code sketches are relatively small, the authentic libraries involved in these benchmarks are not small. E.g., the ArrayList from OpenJDK8 [28] contains 1.4kLOC. Moreover, the authentic versions of the libraries contain Java features like reflection and lambda expression that JSketch does not support yet; some of them also invoke native code. There-

Table 2. Description of benchmarks and experimental results.

Benchmark						Toshokan		JSketch (Model)		JSketch (Mock)	
Name	LoC	#C	Lib Data Type	Lib Func(s)	Enc	Time(s)	#I	Model LoC	Time(s)	Mock LoC	Time(s)
gcd_n_numbers	70	12	int	gcd	D	22.44	1	20(29%)	4.8	16(23%)	5.96
lcm_n_numbers	74	12	int	lcm	D	25.91	3	21(28%)	3.92	17(23%)	4.11
powerroot_sqrt	65	15	int	sqrt	D	15.40	1	14(22%)	3.81	19(29%)	3.84
primality_sqrt	56	12	int	sqrt	D	32.89	4	14(25%)	3.76	19(34%)	3.75
evalPoly_1	61	15	int	pow	D	22.77	3	30(49%)	3.91	11(18%)	4.32
evalPoly_2	59	15	int	pow	D	6.86	1	30(51%)	3.78	11(19%)	3.74
evalPoly_combined	97	30	int	pow	D	15.52	2	30(31%)	3.99	11(11%)	4.31
stack_match	24	8	Stack	push,pop	C	39.18	5	N/A	N/A	22(92%)	3.86
set_match	26	8	HashSet	add,contains	C,Q	8.76	1	29(112%)	4.02	32(123%)	4.01
arraylist_match	26	8	ArrayList	push_back,get	C,Q	41.75	5	29(112%)	4.1	22(85%)	3.8
heap_sort	72	20	Heap	insert,pop_min	C	32.56	4	N/A	N/A	159(221%)	4.44

Enc–Encoding(s) used in benchmark for Toshokan
D–Direct encoding of primitive types, C–Call-tree based encoding, Q–Query based encoding
#*C*–number of control bits in the sketch after preprocessing. #*I*–number of iterations Toshokan runs.

fore inlining these libraries is beyond the capacities of JSketch as currently implemented.

For each benchmark, we list the size of the program sketch, the number of control bits in the sketch, and the library's signature (see Table 2). Note that the "#*C*" column of the table describes the numbers of bits needed to represent a solution candidate for the synthesis task in the benchmark, i.e. a number of N control bits of the benchmark indicates that the search space of its solution is 2^N. Additionally, the LoC sizes of the model and mock code which JSketch uses for the respective benchmark are shown in the table, as well as their relative sizes to the benchmark per se.

Experimental Results. The experiments were conducted on a server with 2 Intel(R) Xeon(R) E5-2630 v4 10-core CPUs, with each core having 2 threads, at main frequency of 2.20 GHz, with 128 GB of memory. The experiments were run as 10 independent parallel tasks, and the whole process terminates once any of the 10 simultaneously running tasks returns with a correct synthesized solution.

Since the solving process of Sketch synthesis engine involves nondeterministic algorithms presenting nondeterministic intermediate results and performance, as well as having a large range of different configuration parameters that could be potentially optimized, the parallelism described above could be a great help in increasing overall performance for both our methods and our comparing methods, as long as parallel computing resources are available. Experiments were run on all 11 benchmarks with a timeout of 1 h. Performance of sketch with appropriate models and/or mocks on these same benchmarks are also collected whenever possible using the same parallel methods described above, as baseline of performance for the effectiveness evaluation.

Toshokan successfully solved all benchmarks within the timeout. Our experimental results, including the solving time and number of iterations taken by Toshokan to find the solution, are shown in Table 2. The results give an **answer to RQ1**: Toshokan was able to effectively handle synthesis tasks that interact with a wide range of different libraries and library functions, including advanced arithmetic operations, as well as complex composite data structures. This indicates a good variety of our methods' possible applications.

Now let us proceed to the second research question. We take the mock/-model LoC as a measure of the extra code needed for the synthesizer user to write which Toshokan managed to save, and the extra time Toshokan takes to solve the benchmark comparing to mock/model as measures of performance overhead. We believe the measures allow us to reasonably indicate the benefit/cost ratio of our approach.

Figure 9 compares the absolute amount of extra LoC against extra time by Toshokan. This figure indicates how Toshokan trades extra synthesis time for saving the programmer's effort (in terms of LoC). For example, by adopting Toshokan, a programmer who wants to write the *set_match* program could save 32 LoC writing at the cost of waiting for only 4 extra seconds. This figure would paint a picture from a potential user's perspective on the performance numbers.

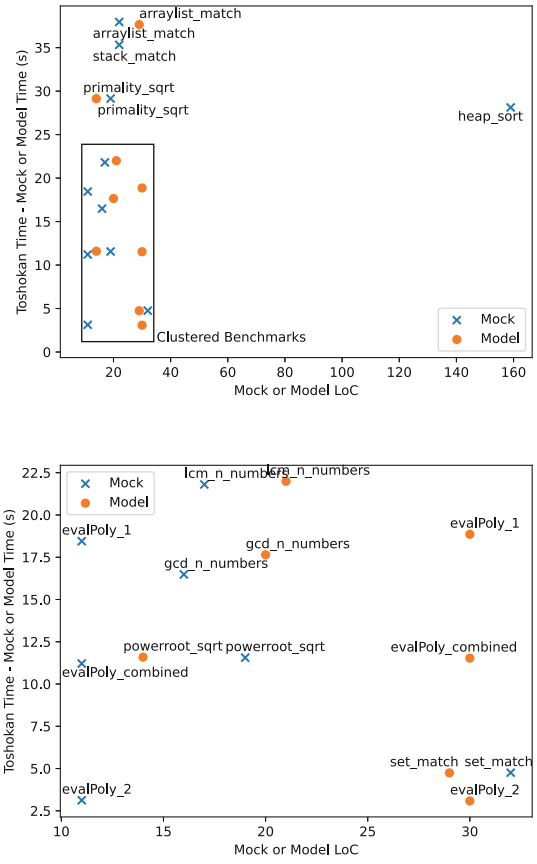

Fig. 9. Experimental results: saved LoC vs. extra time, absolute amount (above for all benchmarks and below for clustered benchmarks).

Observing Table 2 and the figures, we are encouraged toward an **answer to RQ2**: On one hand, Toshokan saves the user some work from writing various kinds of mocks and models. Depending on the actual complexities of the underlying library and synthesis task, the Mock/Model LoC ranges from 11 to 159, and could be as high as 2.21× of the original sketch LoC. On the other hand,

the performance slowdown is moderate. All benchmarks on Toshokan showed a slowdown of less than 40 s. Given the extra LoC of code writing Toshokan managed to eliminate, we consider this performance as acceptable in proving the effectiveness of our methods.

7 Related Work

Several library models have been proposed in different settings and handled by the synthesizer in different ways. Gascon *et al.* [15] model each component of the target language as a proof rule, which allows the synthesizer to symbolically execute programs consisting of these components. JLibSketch [25] model libraries as equational specifications and automate the reasoning about library models by term rewriting. Unlike above approaches in which the rules/equations are manually written by experts, the model in Toshokan is simply a partial function and automatically generated. We tried to include the JLibSketch on the reported benchmarks [25]. Unfortunately, we found the Java programs produced for all these benchmarks are flawed and cannot be handled by our implementation.[6]

The function models used in modular synthesis [35] are more flexible. The model can be either strong (deterministic) or weak (nondeterministic). In other words, there can be a unique or multiple valid outputs for the same input. The key contribution of their work is a CEGIS+ algorithm which handles both strong and weak models efficiently. However, the functional models and the canonicalization functions they rely on are still manually written, while our library models are generated automatically from the logger. This is made possible by the fact that there is a canonical, executable library in our setting. In contrast, the original functions in [35] are not necessarily executable—these functions per se can be templates with holes to be filled by the synthesizer. The JSketch (Model version) which we have compared with implements the CEGIS+ algorithm (see experimental results in Sect. 6). The idea of angelic synthesis is also used by Burst [26], which does not aim to synthesize and handle library models.

There is a rich literature in *component-based synthesis*, which aims to generate a program consisting of library calls to a provided API. This line of work was pioneered by Prospector [24] and followed by many synthesis tools including CodeHint [14], SyPet [12], EdSynth [43] and FrAngel [33]. These systems typically synthesize code making library calls by actually executing the candidate program on a set of test cases. Our approach also treats the library as a black box but not for testing. We synthesize provably-correct, library-using programs by incorporating inductive synthesis (using JSketch) and bounded verification (using JBMC).

Unit-test generation is another related research area. The task is to generate sequences of method calls to exercise the library to be tested. For example,

[6] As a limitation of current JSketch, when generators are involved, the raw output of JSketch is not compilable and some manual adaptation is needed. This is impossible for Toshokan because the CEGIS loop must compile JSketch output every iteration.

Pacheco *et al.* [29] generate method calls randomly and guide the generation with feedback from executing the generated sequences. FUDGE [3] extracts code snippets from corpus code and mutate them to generate fuzz drivers. Instead of testing libraries, our purpose is to synthesize client code of libraries that satisfies formal specifications.

The idea of delegating complex verification tasks to external oracles is also explored by SyMO [31], but the main distinction is about the synthesizer's side. SyMO (and all existing SyGuS solvers) treat libraries as a white-box, defined function. In other words, when library call f(x) is part of the grammar, SyGuS solvers need access to the implementation of f(x) as a defined function. In contrast, our angelic inductive synthesizer treats the library as a black box. Moreover, many libraries are not pure-functional (e.g. Stack.push), with side effects updating the internal composite data structure, which cannot be handled by SyMO.

External function handling in direct manipulation. The most related work to this paper is JDIAL [17], which performs direct manipulation, a special form of program repair. In each iteration of the synthesis procedure, JDIAL leverages Sketch to guess a *single input-output pair* of the library function which is not covered by the current program's execution, then runs the library function with the guessed input to check whether the guessed output is correct. While JDIAL and TOSHOKAN share the same idea of dynamically expanding the library model, they are different in several aspects. First, JDIAL supports interactive program repair, while TOSHOKAN, for the first time, integrates the dynamic library model into a fully-automatic sketch-based synthesis procedure. Second, JDIAL only handles simple mathematical functions such as Math.pow or Math.max, and it is not clear how their approach can be extended to support libraries manipulating objects, with internal states and references, etc., and how scalable their approach toward more sophisticated libraries. Third, JDIAL runs authentic libraries eagerly and not guided by counterexamples, e.g., its Math.max example takes more than 90 iterations. By contrast, TOSHOKAN runs the whole rejected program and only logs those library calls witness the failure, making the generated library model smaller and more helpful for the synthesizer.

8 Conclusion

We proposed TOSHOKAN, a new program synthesis framework in which programs that use external libraries could be synthesized without any mock or model from the user. TOSHOKAN extends the classic counterexample-guided inductive synthesis framework with a bootstrapping, log-based library model. We found that, comparing to existing synthesis techniques that are able to handle external libraries through user-provided models or mocks, our methods save the user from the extra manual work, at the cost of moderate performance overhead.

Acknowledgments. This research was supported in part by the National Science Foundation under Grant Nos. CCF-1919197 and CCF-2046071.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2002. ACM (2002). https://doi.org/10.1145/503272.503275
2. Astorga, A., Madhusudan, P., Saha, S., Wang, S., Xie, T.: Learning stateful preconditions modulo a test generator. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019. ACM (2019). https://doi.org/10.1145/3314221.3314641
3. Babić, D., et al.: Fudge: fuzz driver generation at scale. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019. ACM (2019). https://doi.org/10.1145/3338906.3340456
4. Bastani, O., Anand, S., Aiken, A.: Specification inference using context-free language reachability. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2015. ACM (2015). https://doi.org/10.1145/2676726.2676977
5. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Active learning of points-to specifications. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2018. ACM (2018). https://doi.org/10.1145/3192366.3192383
6. Bodik, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL 2010. ACM (2010). https://doi.org/10.1145/1706299.1706339
7. Bornholt, J., Torlak, E.: Finding code that explodes under symbolic evaluation. In: Proc. of the ACM on Programming Languages. OOPSLA 2018, vol. 2. ACM, October 2018. https://doi.org/10.1145/3276519
8. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: a bounded model checking tool for verifying Java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10
9. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: deductive synthesis of abstract data types in a proof assistant. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2015. ACM (2015). https://doi.org/10.1145/2676726.2677006
10. Doughty-White, P., Quick, M.: Codebases: millions of lines of code (2015). https://informationisbeautiful.net/visualizations/million-lines-of-code/
11. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015, special issue on Experimental Software and Toolkits
12. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017. ACM (2017). https://doi.org/10.1145/3009837.3009851
13. Floyd, R.W.: Nondeterministic algorithms. J. ACM (JACM) **14**(4), 636–644 (1967). https://doi.org/10.1145/321420.321422
14. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: CodeHint: dynamic and interactive synthesis of code snippets. In: Proceedings of the 36th International Conference on Software Engineering. ICSE 2014. ACM (2014). https://doi.org/10.1145/2568225.2568250

15. Gascón, A., Tiwari, A., Carmer, B., Mathur, U.: Look for the proof to find the program: decorated-component-based program synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 86–103. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_5
16. Heule, S., Sridharan, M., Chandra, S.: Mimic: computing models for opaque code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015 (2015). https://doi.org/10.1145/2786805.2786875
17. Hu, Q., Samanta, R., Singh, R., D'Antoni, L.: Direct manipulation for imperative programs. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 347–367. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_17
18. JDial Debugger (2021). https://github.com/JDial-Debugger/backend/tree/master/SkechObject/benchmarks
19. Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J.S., Solar-Lezama, A.: Synthesizing framework models for symbolic execution. In: ICSE 2016. ACM (2016). https://doi.org/10.1145/2884781.2884856
20. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Jsketch: sketching for Java. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015. ACM (2015). https://doi.org/10.1145/2786805.2803189
21. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conf. on Software Engineering. ICSE 2010, vol. 1. ACM (2010). https://doi.org/10.1145/1806799.1806833
22. Li, W., Seshia, S.A.: Sparse coding for specification mining and error localization. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 64–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_9
23. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. Int. J. Softw. Tools Technol. Transf. 603–618 (2012). https://doi.org/10.1007/s10009-012-0236-z
24. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2005. ACM (2005). https://doi.org/10.1145/1065010.1065018
25. Mariano, B., et al.: Program synthesis with algebraic library specifications. In: Proceedings of the ACM on Programming Languages. OOPSLA 2019, vol. 3. ACM (Oct 2019). https://doi.org/10.1145/3360558
26. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. In: Proceedings of the ACM on Programming Languages. POPL 2022, vol. 6. ACM, January 2022. https://doi.org/10.1145/3498682
27. Murali, V., Qi, L., Chaudhuri, S., Jermaine, C.: Neural sketch learning for conditional program generation. In: International Conference on Learning Representations (2018). https://openreview.net/forum?id=HkfXMz-Ab
28. OpenJDK (2014). https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java
29. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE 2007), May 2007. https://doi.org/10.1109/ICSE.2007.37
30. Pei, Y., Furia, C.A., Nordio, M., Wei, Y., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. IEEE Trans. Softw. Eng. **40**(5), 427–449 (2014). https://doi.org/10.1109/TSE.2014.2312918

31. Polgreen, E., Reynolds, A., Seshia, S.A.: Satisfiability and synthesis modulo oracles. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 263–284. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_13
32. Raychev, V., Bielik, P., Vechev, M., Krause, A.: Learning programs from noisy data. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2016. ACM (2016). https://doi.org/10.1145/2837614.2837671
33. Shi, K., Steinhardt, J., Liang, P.: Frangel: component-based synthesis with control structures. In: Proceedings of the ACM on Programming Languages. POPL 2019, vol. 3. ACM, January 2019. https://doi.org/10.1145/3290386
34. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2013. ACM (2013). https://doi.org/10.1145/2491956.2462195
35. Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 395–414. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_22
36. Sketch (2021). https://github.com/asolarlez/sketch-frontend/blob/master/src/experiments/sk/models/
37. Skrupsky, N., Monshizadeh, M., Bisht, P., Hinrichs, T., Venkatakrishnan, V.N., Zuck, L.: Waves: automatic synthesis of client-side validation code for web applications. In: 2012 International Conference on Cyber Security, December 2012. https://doi.org/10.1109/CyberSecurity.2012.13
38. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_2
39. Solar-Lezama, A.: The sketch programmers manual (2020). https://people.csail.mit.edu/asolar/manual.pdf, version 1.7.6
40. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM (2006). https://doi.org/10.1145/1168857.1168907
41. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2011. ACM (2011). https://doi.org/10.1145/1993498.1993557
42. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2014. ACM (2014). https://doi.org/10.1145/2594291.2594340
43. Yang, Z., Hua, J., Wang, K., Khurshid, S.: EdSynth: synthesizing API sequences with conditionals and loops. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. ICST 2018, April 2018. https://doi.org/10.1109/ICST.2018.00025