Bayesian Sketch Learning for Program Synthesis

VIJAYARAGHAVAN MURALI, Rice University
LETAO QI, Rice University
SWARAT CHAUDHURI, Rice University
CHRIS JERMAINE, Rice University

We present a Bayesian statistical approach to the problem of automatic program synthesis. Our synthesizer starts by learning, offline and from an existing corpus, a probabilistic model of real-world programs. During synthesis, it is provided some ambiguous and incomplete evidence about the nature of the programming task that the user wants automated, for example sets of API calls or data types that are relevant for the task. Given this input, the synthesizer infers a posterior distribution over type-safe programs that assigns higher likelihood to programs that, according to the learned model, are more likely to match the evidence.

We realize this approach using two key ideas. First, our learning techniques operate not over code but syntactic abstractions, or sketches, of programs. During synthesis, we infer a posterior distribution over sketches, then concretize samples from this distribution into type-safe programs using combinatorial techniques. Second, our statistical model explicitly models the full intent behind a synthesis task as a latent variable. To infer sketches, we first estimate a posterior distribution on the intent, then use samples from this posterior to generate a distribution over possible sketches. We show that our model can be implemented effectively using the new neural architecture of Bayesian encoder-decoders, which can be trained with stochastic gradient descent and yields a simple inference procedure.

We implement our ideas in a system, called BAYOU, for the synthesis of API-heavy Java methods. We train BAYOU on a large corpus of Android apps, and find that the trained system can often synthesize complex methods given just a few API method names or data types as evidence. The experiments also justify the design choice of using a latent intent variable and the levels of abstraction at which sketches and evidence are defined.

1 INTRODUCTION

The automatic synthesis of programs [11, 18, 29, 47] is a long-standing goal in computer science. The last decade has seen a resurgence of interest in this problem [5, 14, 19, 44, 45], and tools solving versions of the problem have begun to see industrial adoption [20].

Existing approaches to synthesis commonly frame the problem as search over a space of programs [5]. The goal is to find a program that satisfies a user-provided specification. In spite of much recent progress, these methods continue to struggle with two basic challenges.

The first challenge is uncertainty in specification. For synthesis to be justified, the user of the synthesis system should be able to specify programming tasks easily. In most real-world settings, this means that the user writes an underspecification, rather than a full formal definition, of a task. The difficulty, however, is that a program may pass such an underspecification but violate the user’s true intent. For example, in the popular paradigm of synthesis from input-output examples [14, 19], a synthesizer could simply generate a monolithic if-statement that replays the outputs present in the example set on the corresponding inputs. Clearly, such a program does not match user intent, does not generalize beyond the given examples, and has no tolerance for error in user input.

The second major challenge is computational complexity. The space of programs relevant in a typical synthesis problem is vast, and only a few of these programs might fit a given specification. Finding such a program is a fundamentally hard problem.

The most popular way of addressing these challenges is to assume some syntactic constraints on the programs that are to be synthesized [5]. These constraints are imposed either by having an
incomplete syntactic model, known as a program sketch [43, 44], as part of the problem instance, or by restricting synthesis to a narrowly tailored domain-specific language (DSL) [9, 19, 35].

Such syntactic constraints rule out programs that obviously violate user intent, making it easier to generalize underspecifications. They also reduce the computational difficulty of synthesis by limiting the search space of programs. At the same time, these constraints represent low-level knowledge and domain-specific assumptions, and compromise the goal of building a general-purpose automatic programming system. Finding alternatives to such restrictions is therefore a natural research question.

In this paper, we present such an alternative, in the form of a Bayesian statistical approach to synthesis that uses knowledge discovered from pre-existing code to generalize underspecifications and efficiently search for programs. At the heart of our method is a statistical model that is trained on a corpus of real-world programs, and during training, learns a prior probability distribution that associates programs with specifications that match them. During synthesis, the user of the approach is expected to provide some shallow facts about the program they have in mind, for example the API calls that the program makes or the data types that it uses. This specification is viewed as not the ultimate truth, but rather evidence as to the nature of the synthesis task. Our synthesis problem is to infer a posterior distribution over programs that assigns: (1) positive likelihood only to programs that meet a set of language-level invariants (assumed to be encapsulated by a type system); and (2) higher likelihoods to programs that, given the data, are more likely to match the evidence.

Our solution to this problem relies on two key technical ideas. First, because the source code of programs contains many low-level details that do not generalize across programs, our statistical model does not directly learn over programs. Instead, it operates on syntactic abstractions, or sketches, of programs. At the time of synthesis, we use the model to infer a probabilistic process that can iteratively generate sketches. Synthesis now amounts to sampling from this process and fleshing the samples out into complete, type-safe programs using known combinatorial techniques.

The second essential idea concerns the design of our statistical model. This model uses a latent random variable $Z$ to explicitly model the intent that lies behind the user-provided evidence and completely defines the programming task at hand. Sketches and evidence are modeled as observed random variables $X$ and $Y$ that each depend on $Z$, but are conditionally independent given $Z$. Training the model amounts to learning a maximum likelihood estimate of the joint distribution $P(X, Y, Z)$. The main inference task is to estimate the posterior distribution $P(Y|X = X)$ over sketches given evidence $X$. This task is split into two phases, one using Bayes’ rule to estimate a posterior distribution $P(Z|X = X)$ over intent, and the other sampling sketches from the distribution $P(Y|Z = Z)$ given a $Z$ sampled from $P(Z|X = X)$.

We realize our model using a novel neural architecture called the Bayesian encoder-decoder (BED), a cousin of the recently popular model of variational autoencoders [26]. A BED consists of an encoder network and a decoder network, linked via the latent variable $Z$. The networks are trained together by optimizing, using stochastic gradient descent, an end-to-end differentiable loss function that is a variational approximation of the maximum likelihood objective in our model. Together, the networks enable a simple procedure for estimating the posterior distribution $P(Y|X = X)$ over sketches.

We have implemented our approach in a system, called BAYOU, for the synthesis of Java programs heavy on API usage. We evaluate BAYOU in the synthesis of API-manipulating Android methods, using a corpus of about 100,000 Android programs drawn from an online repository. Our experiments show that BAYOU can rapidly synthesize complex programs from far more ambiguous and under-specified goals than in existing approaches to synthesis. Some of these programs implement tasks that are not encountered during training. We also demonstrate the benefits of some of our key design choices, in particular the use of a latent variable for intent and the choice of the level of abstraction at which sketches and evidence are defined.
Now we summarize the main contributions of this paper:

- We propose a Bayesian statistical approach for program synthesis, in which the user presents some evidence as to the nature of a programming task, and the synthesizer infers a posterior distribution over type-safe programs given this evidence.
- We instantiate this approach in the form of a probabilistic model that correlates sketches with evidence and intent, generates sketches relevant to a given synthesis task, and concretizes the generated sketches using combinatorial techniques.
- We introduce the neural architecture of BEDs, which is trained using variational bounds and yields an efficient inference procedure, as a way of implementing this model.
- We implement our ideas in a system, called BAYOU, and demonstrate their benefits in the synthesis of API-usage-heavy Android methods.

The rest of the paper is organized as follows. Section 2 presents a few motivating examples for our approach. Section 3 formulates our learning and synthesis problems. Section 4 presents the methodology of synthesis with learning over sketches; Section 5 discusses BEDs. Section 6 describes the concretization of sketches into programs. Section 7 presents our experiments, and Section 8 discusses related work. Section 9 presents our conclusions.

2 MOTIVATING EXAMPLES

In this section, we illustrate some of the key capabilities of our approach, through a few concrete usage scenarios for the BAYOU system.

Suppose a user of BAYOU wants to synthesize a Java method foo that reads from a file. To achieve this, they write a “draft” program (Figure 1) that contains a placeholder for foo, and sets up an environment that defines the types of the free variables and return value of the synthesized code. They also write some evidence annotations that partially reveal what foo should look like. In the current implementation of BAYOU, such evidence can take three forms: (i) names of API methods that foo should invoke (e.g., readLine), (ii) API datatypes that foo should use (e.g., FileReader), and (iii) datatypes present in the environment of foo that are of particular importance (e.g., String).

To begin with, suppose the user only provides evidence of the first type: the single API method readLine. Given this evidence, BAYOU infers a posterior distribution over programs. This distribution assigns higher likelihood to programs that more frequently resemble programs in the training data with calls to readline. A ranked list of the “top k” most likely programs sampled from this distribution is then returned to the user.

Figure 2(a) shows a top-five program that BAYOU returns in a run on this specific input. As we see, this program indeed reads from a file. However, it also imports relevant classes, uses try-catch blocks, and closes the reader, even though these actions were not directly specified by the user.

Although the program in Figure 2-(a) satisfied the true user intent in this case, BAYOU also returns a few programs that do not meet this intent. For example, the program in Figure 2-(b), which uses an InputStreamReader object rather than a FileReader, also appears in the top five results in our trial run. Here, the synthesizer tried to construct an InputStream object from the given String argument to pass to the constructor of InputStreamReader. While it managed to find a chain of methods calls that does this, the resulting program reads from a stream rather than a file.

The user can rule out this program by instructing the synthesizer to use the FileReader type as opposed to InputStreamReader. This intent can be expressed with an additional piece of
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
public class Program {
    void foo(String file) {
        String s;
        BufferedReader br;
        FileReader fr;
        try {
            fr = new FileReader(file);
            br = new BufferedReader(fr);
            while ((s = br.readLine()) != null) {}
            br.close();
        } catch (FileNotFoundException _e) { }
        catch (IOException _e) { }
    }
}

(a)

import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.BufferedReader;
public class Program {
    void foo(String file) {
        String s;
        BufferedReader br;
        FileReader fr;
        try {
            fr = new FileReader((f = new File(file)));
            br = new BufferedReader(fr);
            while ((s = br.readLine()) != null) {}
            br.close();
        } catch (FileNotFoundException _e) { }
        catch (IOException _e) { }
    }
}

(b)

Fig. 2. Programs synthesized by BAYOU with the API method name readLine as evidence.

evidence: the datatype FileReader. Once this evidence is supplied, all programs in the top-five list only use FileReader. The variation in the results now arises from different ways of handling exceptions, and different ways of constructing a FileReader from the input String (some
programs use the String argument directly, while others create a File object out of the argument and used that instead). Figure 3 shows two other top-five programs returned on this input.

Interestingly, we obtain similar results by providing as evidence the datatype String present in the environment of foo. This illustrates an interesting fact that BAYOU learns from data, that programs that read from files often also make use of a String variable in their environment — indeed, this variable typically stores the filename. In all cases, however, BufferedReader always appears in the code, indicating another learned fact: that reading files and streams in Java is almost always done in a buffered manner.

```
try {
    call FileReader.new(String);
    call BufferedReader.new(Reader);
    while ([BufferedReader.readLine()] ) {};
    call BufferedReader.close()
} catch (FileNotFoundException) {
    call Throwable.printStackTrace()
} catch (IOException) {
    call Throwable.printStackTrace()
}
```

Fig. 4. Sketch of the program in Figure 3-(a)

The above results critically depend on synergy between a procedure for statistical learning over sketches and a combinatorial method for generating programs from sketches. In BAYOU, sketches are defined so as to only retain aspects of programs that are broadly shared across the training set, such as API calls, datatypes, and high-level control structure. For example, the sketch for the program in Figure 3-(a) is shown in Figure 4-(a). While this sketch contains control constructs such as loops and exception handling blocks, it elides all local names, and uses types to abstract arguments to method calls.

3 PROBLEM STATEMENT

In this section, we define our learning and synthesis problems. These problems are parameterized on the language of programs that is the target of synthesis. In the rest of this section, we first define the problems abstractly, then make them concrete using a core language, called AML, that captures the subset of Java programs that BAYOU handles.

3.1 Learning and synthesis

Let us assume a language of programs \( \text{Prog} \). The language has a static type system, which enforces the minimal level of correctness that programs returned to users must meet. At this point, we keep this type system abstract. Fix a universe of variable names and a universe of types, and let a type environment \( \text{Env} \) be a partial mapping from variable names to types. We simply assume a predicate safe(\( \text{Prog} \), \( \text{Env} \)) that evaluates to true if and only if \( \text{Prog} \) can be typed under \( \text{Env} \).

Let us assume a universe of evidence vectors that capture facets of program behavior and are used to express ambiguous user intent. During synthesis, the user provides an evidence vector \( X \), along with a type environment \( \text{Env} \) that captures the context where the synthesized code is inserted.

As a convention, we let names of random variables appear in italics and start with capital letters. When written in sans serif (for example \( X \)), the name of a variable denotes deterministic values of the variable. We write \( P(\text{X}, \text{Y}) \), and \( P(\text{Y}|\text{X}) \) respectively to denote the probabilities \( P(\text{X} = \text{X}, \text{Y} = \text{Y}) \) and \( P(\text{Y} = \text{Y}|\text{X} = \text{X}) \), respectively. The conditional distribution \( P(\text{X}|\text{Y} = \text{Y}) \) is abbreviated as \( P(\text{X}|\text{Y}) \). Notation such as \( P(\text{X}|\text{M}) \) is used to denote a family of distributions parameterized by a parameter set \( \text{M} \) (where \( \text{M} \) is understood, we will often drop it rather than giving it explicitly).

Our statistical model assumes three variables \( \text{Prog} \), \( \text{X} \), and \( \text{Env} \), respectively ranging over programs, evidence vectors, and type environments. Suppose we are given a training corpus \( \mathcal{D} \) consisting of triples \( (X_i, \text{Prog}_i, \text{Env}_i) \) of evidence vectors, programs, and environments. Given \( \text{Env}_i \), each \( (X_i, \text{Prog}_i) \) pair is assumed to be an independent sample taken from the joint distribution
\( P(X, \text{Prog}|M, \text{Env}_i) \). This distribution correlates programs with the environments and evidence vectors under which they would be written. Note that in this distribution, Env is a given, since synthesis will always happen within a particular (observable) environment.

In general, \( M \) is unknown and so we seek to learn an appropriate value for \( M \) so that the resulting distribution “fits” the observed data. We formulate this problem as one of maximum likelihood estimation [8], where the goal is find parameters of a generative statistical model so as to maximize the log probability of the model’s training set. Our learning problem amounts to finding an optimal value of \( M \):

**Problem 1** (Learning). Given a corpus \( \mathcal{D} = \{ (X_i, \text{Prog}_i, \text{Env}_i) \} \) and a model \( P(X, \text{Prog}|M, \text{Env}) \), find

\[
M^* = \arg \max_M \sum_i \log P(X_i, \text{Prog}_i|M, \text{Env}_i).
\]

The synthesis problem arises once we have solved the learning problem. The goal now is to estimate a posterior distribution over programs that are likely given an evidence vector \( X \), and are additionally type-safe under Env. Formally, we define the synthesis problem as follows:

**Problem 2** (Synthesis). Given an evidence vector \( X \), a type environment \( \text{Env} \), and a distribution \( P(X, \text{Prog} | \text{Env}) \) obtained by instantiating \( M \) in the model \( P(X, \text{Prog} | M, \text{Env}) \), estimate the posterior distribution \( P(\text{Prog}|X, \text{Env}, \text{safe}(\text{Prog}, \text{Env})) \).

### 3.2 The AML language

AML is a core language that is designed to capture the essence of API usage in Java-like languages. Now we present this language.

AML uses a finite set of API data types. A type is identified with a finite set of API method names (including constructors); the type for which this set is empty is said to be void. Each method name \( a \) is associated with a type signature \( (\tau_1, \ldots, \tau_k) \rightarrow \tau_0 \), where \( \tau_1, \ldots, \tau_k \) are the method’s input types and \( \tau_0 \) is its return type. A method for which \( \tau_0 \) is void is interpreted to not return a value. Finally, we assume predefined universes of constants and variable names.

The grammar for AML is as in Figure 5. Here, \( x, x_1, \ldots \) are variable names, \( c \) is a constant, and \( a \) is a method name. The syntax for programs \( \text{Prog} \) includes method calls, loops, branches, statement sequencing, and exception handling. We use variables to feed the output of one method into another, and the keyword \text{let} to store the return value of a call in a fresh variable. \( \text{Exp} \) stands for (object-valued) expressions, which include constants, variables, method calls, and let-expressions such as “\text{let} \ x = \text{Call} : \text{Exp}”, which stores the return value of a call in a fresh variable \( x \), then uses this binding to evaluate the expression \( \text{Exp} \). (Arithmetic and relational operators are assumed to be encompassed by API methods.) A variable is free if in the program if it is used but not declared; the set of types of these variables are known as the program’s context.

The operational semantics and type system for AML are standard, and consequently, we do not describe these in detail.

**Evidence vectors.** The current version of BAYOU allows three forms of evidence. These are:
(1) **Set of types:** The user can present a set of types $X_{Types}$ used by the target program. For example, in the problem of synthesizing the code in Figure 2-(a), the sets \{BufferedReader\}, \{FileReader\} and \{BufferedReader, FileReader\} can be used as evidence.

(2) **Set of method calls:** Evidence can include a set of method calls $X_{Calls}$ used in the target program. For example, we can use \{readline\}, \{close\} and \{readline, close\} as evidence for the synthesis of Figure 2-(a).

(3) **Relevant context:** Evidence can include a subset $X_{Cxt}$ of the target program’s context. For example, \{String\} can be used as evidence during the synthesis of Figure 2-(a).

Accordingly, an evidence vector in BAYOU is a triple $(X_{Cxt}, X_{Types}, X_{Calls})$.

### 4 SYNTHESIS WITH BAYESIAN SKETCH LEARNING

Now we present the statistical model that we use to solve our learning and synthesis problems. As mentioned earlier, our approach relies on two key ideas: learning over sketches and modeling user intent explicitly as a latent variable. We develop these ideas in sequence.

#### 4.1 Sketch learning

In principle, one could solve Problems 1 and 2 by directly learning over the source code of programs. However, source code contains many kinds of low-level information, for example program-specific names and low-level variations in the implementation of similar tasks, that make learning hard (more on this in Section 7). Accordingly, we reformulate our learning task to be over sketches. This means that during training, we associate evidence with sketches rather than programs. During synthesis, we infer a distribution over sketches and then concretize it into a distribution over programs.

For now, we define the notion of a sketch $Y$ abstractly, as a structure that captures some key facets of program syntax. Sketches do not contain variable names, which are unlikely to generalize across a corpus. However, they can carry information about broadly shared facets of programs, such as the types and API methods that the program uses. We also assume a random variable $Y$ ranging over sketches.

The process of abstracting programs into sketches is formalized using an abstraction function $\alpha : Prog \mapsto Y$. The inverse process, of generating a program given a sketch and a type environment, is captured by a conditional distribution $P(Prog | Y, Env)$, called the concretization distribution. Intuitively, the concretization distribution is an abstract representation of an idealized sketch-based, stochastic synthesis procedure. This procedure has the desirable property of only generating programs that can be typed under Env. If $Y$ cannot be concretized into type-safe programs under $Env$, the procedure returns a special error value with probability 1. For notational convenience, we define this error value to be a special program $\bot$.

Formally, we impose the following requirements on $\alpha$ and $P(Prog | Y, Env)$:

1. For all $Prog$, $Y$, $Env$, $P(Prog | Y, Env) \neq 0$ only if $Y = \alpha(Prog)$.  
2. For all $Prog$, $Y$, $Env$, if $Prog \neq \bot$ and $\neg safe(Prog, Env)$, then $P(Prog | Y, Env) = 0$  
3. If there exists a $Prog$ such that $safe(Prog, Env)$ and $\alpha(Prog) = Y$, then $P(\bot | Y, Env) = 0$.

We assume certain independence relationships among $Prog$, $Env$, $X$, and $Y$. These are summarized as a Bayesian network [8] in Figure 6. By our assumption of a concretization distribution, $Prog$ depends on $Y$ and $Env$. However, we assume $X$ and $Y$ are independent of $Env$. An intuitive justification of this assumption is that $Env$ defines variable names that for use in the low-level implementation of
a program, whereas \( X \) and \( Y \) capture high-level structural intuitions and do not use program-specific names. Together, these assumptions imply that for all \( \text{Prog}, \text{Env}, Y, X \),

\[
P(\text{Prog}, X, Y | \text{M}, \text{Env}) = P(X | \text{M}) P(Y | \text{M}, X) P(\text{Prog} | Y, \text{Env}). \tag{4}
\]

Now we reformulate our learning problem. We have, as before, a training set \( \mathcal{D} \) of triples \((X_i, \text{Prog}_i, \text{Env}_i)\). However, our model is now \( P(X, Y | \text{M}, \text{Env}) \), where \( Y \) ranges over sketches\(^1\). Our learning goal is still maximizing the likelihood of \( \mathcal{D} \). Let \( Y_i = \alpha(\text{Prog}_i) \). We want to compute

\[
M^* = \arg \max_M \sum_i \log P(X_i, \text{Prog}_i | \text{M}, \text{Env}_i)
= \arg \max_M \sum_i \log \left( \sum_Y P(\text{Prog}_i | Y, \text{Env}_i) P(Y, X_i | \text{M}) \right) \quad \text{[by Equation 4]}
= \arg \max_M \sum_i \log P(\text{Prog}_i | Y_i, \text{Env}_i) P(Y_i, X_i | \text{M}) \quad \text{[by Equation 1]}
= \arg \max_M \sum_i \log P(X_i, Y_i | \text{M}).
\]

Thus, our learning problem amounts to maximum likelihood estimation over a dataset of pairs of sketches and evidence vectors. We state the problem as follows:

**Problem 3** (Sketch learning). Given \( \mathcal{D} = \{(X_i, \text{Prog}_i, \text{Env}_i)\} \), \( \alpha \), and \( P(X, Y | \text{M}) \), and letting \( Y_i = \alpha(\text{Prog}_i) \), find \( M^* = \arg \max_M \sum_i \log P(X_i, Y_i | \text{M}) \).

Our synthesis problem now becomes:

**Problem 4** (Synthesis with sketch learning). Given \( X, \text{Env} \), and a learned distribution \( P(X, Y) \), estimate the posterior distribution

\[
P(\text{Prog} | X, \text{Env}, \text{safe}(\text{Prog}, \text{Env})) = \sum_Y P(\text{Prog} | Y, \text{Env}, \text{Prog} \neq \bot) P(Y | X).
\]

**Sketches in Bayou.** The abstraction process in BAYOU preserves information about a program’s high-level control structure and the method calls that the program makes, while erasing all local names. More precisely, each API call is abstracted into an abstract method call \( \tau.a(\tau_1, \ldots, \tau_k) \), where \( \tau \) is the type of the object on which the method is called, and \( \tau_i \) is the type of the method’s \( i \)-th argument. Each expression \( \text{Exp} \) is abstracted into an abstract expression: a list containing the abstraction of each method call that appears in \( \text{Exp} \).

We show the grammar for sketches in Figure 7, and define the abstraction function \( \alpha \) in Figure 8.

As for the concretization distribution \( P(\text{Prog} | Y, \text{Env}) \), we implement it in the form of a sampling procedure that guarantees the criteria in Equations 1-3. For more details, see Section 6.

### 4.2 User intent as a latent variable

\(^1\)The names \( X \) and \( Y \) for evidence vectors and sketches are motivated by the fact that in this model, the former are the inputs and the latter the outputs.
That is, the distribution for $X$ to capture the uncertainty in intent. This variable is latent, or unseen, as we do not have direct $X$ than socket programs. The distributions for all $X$ produced to match the intent) are generated conditionally, based on the intent. However, the evidence conditioned on the intent. Both the evidence (which describes the intent) and the sketch (which is produced to match the intent) are generated conditionally, based on the intent. However, the evidence and sketch are independent given a particular intent. We summarize the relationships between $X$, $Y$, $Z$, and $Env$ in the Bayesian network in Figure 9.

Fig. 8. The abstraction function $\alpha$.

Fig. 9. Bayes net for $Prog$, $X$, $Y$, $Z$, and $Env$

Now we present the statistical model that we use to solve Problems 3 and 4. As mentioned before, a central feature of this model is that it explicitly models the intent behind programming tasks. One can intuitively think of an intent $Z$ as a specification for a task, for example the goal of writing a GUI program, or a GUI program that does a specific kind of file I/O, or even a GUI program that achieves a very specific functionality. However, while specifications in the programming language literature are usually constraints, we view intents as parameters of distributions that assign likelihood to evidence vectors and sketches (and indirectly, programs) based on their relevance to a task. Also, intents are uncertain and need not have interpretable, logical representations. We use a random variable $Z$ to capture the uncertainty in intent. This variable is latent, or unseen, as we do not have direct access to user intent either at training or at synthesis time.

The addition of $Z$ causes another reformulation of our learning problem: the model we fit to the data is $P(X, Y, Z | M)$ as opposed to $P(X, Y | M)$.

**Problem 5** (Sketch learning with latent intent). Given $D = \{(X_i, Prog_i, Env_i)\}$, $\alpha$, and a model $P(X, Y, Z | M)$, and letting $Y_i = \alpha(Prog_i)$, find $M^* = \arg \max_M \sum_i \log P(X_i, Y_i | M)$, where $P(X, Y | M) = \int_{\Omega_Z} P(X, Y, Z | M) dZ$ and $\Omega_Z$ denotes the probability space for variable $Z$.

Now that we have added intent, we assume that the joint distribution $P(X, Y, Z)$ factorizes so that for all $X, Y, Z$,

$$P(X, Y, Z) = P(Z) P(X | Z) P(Y | Z).$$

That is, the distribution for $X$, $Y$, and $Z$ corresponds to a generative process where first, the intent is generated, and then both the observable evidence as to the intent as well as the sketch are generated conditioned on the intent. Both the evidence (which describes the intent) and the sketch (which is produced to match the intent) are generated conditionally, based on the intent. However, the evidence and sketch are independent given a particular intent. We summarize the relationships between $Z$ and the variables $X$, $Y$, $Prog$, and $Env$ in the Bayesian network in Figure 9.

During training, we use the corpus to learn the joint distribution $P(X, Y, Z)$. Of its factors, $P(Z)$ is a prior distribution that tells us how typical different types of intent are in the corpus. For example, $P(Z)$ may tell us that GUI programs are less common than file I/O programs, but are more common than socket programs. The distributions $P(X | Z)$ and $P(Y | Z)$ identify the evidence vectors and sketches.
that are typical under a various intents. For example, \( P(X|Z) \) may tell us that programs that read from a file (the intent) tend to be associated with the type `BufferedReader` (the evidence), and \( P(Y|Z) \) may tell us that program that read from a file tend to have one of a few distinct “shapes” (they have a loop that performs a file read in the termination check, for example). In the next section, we show how to use stochastic gradient descent to learn these distributions.

During synthesis, we have access to a specific evidence vector \( X \), and our goal is to sample programs from the distribution \( P(Prog|X, Env, safe(Prog, Env)) \). The process for this follows the structure of the network in Figure 9, and is described in Algorithm 1. Here, step (1) first uses \( X \) in conjunction with Bayes’ rule to obtain an updated or posterior distribution \( P(Z|X) \). Typically, this distribution has less variance than the prior on \( Z \). For example, if \( X \) contains the types `BufferedReader` and `FileReader`, it is likely that the program targeted has to do with file I/O, and the posterior \( P(Z|X) \) will reflect this. In steps (2) and (3) of this algorithm, we use a sample from this posterior to sample a sketch. Because of the way \( Z \) is generated, \( Y \) is also a sample from \( P(Y|X) \). Finally, we concretize this sketch into a type-safe program \( Prog \); given the way \( Y \) was generated, this program is a sample from \( P(Prog|X, Env) \). The final step protects against the case where concretization fails.

**Algorithm 1 Synthesis with Bayesian Sketch Learning**

**Input:** Evidence vector \( X \), type environment \( Env \)

**Output:** Sample program \( Prog \sim P(Prog|X, Env, safe(Prog, Env)) \)

1. Use Bayes’ rule to obtain a posterior distribution for the value of the latent intent \( P(Z|X) \).
2. Use this posterior to sample an intent \( Z \sim P(Z|X) \).
3. Use the trained model to sample a sketch \( Y \sim P(Y|Z) \).
4. Use the concretization distribution to sample a program \( Prog \sim P(Prog|Y, Env) \).
5. If \( Prog = \bot \), reject the sample and start afresh.

## 5 BAYESIAN ENCODER-DECODERS

Learning latent variable models is always challenging, and it is particularly challenging in our case due to the complex data we are modeling, including various forms of evidence, as well as a tree-structured program sketch.

To solve this problem, we propose the neural architecture of **Bayesian encoder-decoders** (BEDs), where the latent intent variable \( Z \) links an encoder modeling \( P(Z|X) \) and a decoder modeling \( P(Y|Z) \). The whole model is trained using a variational bound [26]. In this section, we present this architecture.

We begin our development of the Bayesian encoder-decoder by noting that for a particular \( (X, Y) \) pair, we have:

\[
\log P(X, Y | M) = D_{KL}[P(Z | M, X) \parallel P(Z | M, X, Y)] + \mathcal{L} \tag{6}
\]

On the left-hand side, we have the log-likelihood of the data point, given the model \( M \) (in the remainder of this section, for notational clarity, we do not explicitly list \( M \) in every displayed equation; its precise components are discussed at the end of the section). The first term on the right-hand side is the **Kullback-Leibler (KL)-divergence** measure between \( P(Z | M, X) \) and \( P(Z | M, X, Y) \). The second term \( \mathcal{L} \) is the variational lower bound [22] on \( \log P(X, Y|M) \) which, given that joint distribution \( P(X, Y, Z) \) factorizes as \( P(Z)P(X|Z)P(Y|Z) \), is of the following form:

\[
\mathcal{L} = \mathbb{E}_{Z \sim P(Z|X)}[\log P(X, Y | Z)] - D_{KL}[P(Z | X) \parallel P(Z)]
\]

\[
= \mathbb{E}_{Z \sim P(Z|X)}[\log P(X | Z) + \log P(Y | Z)] - D_{KL}[P(Z | X) \parallel P(Z)] \tag{7}
\]
Since the KL-divergence is always non-negative, it must be the case that \( \log P(X, Y) \geq \mathcal{L} \). Thus by maximizing, via stochastic gradient ascent, the expected value of \( \mathcal{L} \) over \((X, Y)\) pairs sampled from our training set with respect to the parameter \( \mathbf{M} \), we tend to maximize the log-likelihood \( \sum_i \log P(X_i, Y_i) \) over the training data.

Note that maximizing \( \mathcal{L} \) in this way leads to the optimal value for \( \mathbf{M} \) as long as the KL divergence term in Equation 6 is small at \( \mathbf{M}^* \). From Equation 6, this will be the case as long as the evidence \( X \) gives almost as much information as to the value of the intent \( Z \) as both the evidence and the sketch \( Y \) together do. If this is not the case, then the variational approximation may not reach \( \mathbf{M}^* \) — a trait BEDs share with most other variational methods.

5.1 BEDs in BAYOU

Now that we have described the BED framework, we now consider the exact form of the prior \( P(Z) \) as well as the conditional distributions \( P(X|Z) \), \( P(Z|X) \) and \( P(Y|Z) \) that are used in BAYOU.

Prior on Latent Intent. We begin with the prior \( P(Z) \). We define this to be the unit Normal, which has the zero vector as mean and the identity matrix as the covariance matrix:

\[
P(Z) = \text{Normal}(Z|0, I)
\]

This means that each intent \( Z \) is a real-valued vector of arbitrary dimensionality \( d \), where \( d \) is chosen so as to balance computational complexity and result quality. We choose a unit Normal prior for mathematical convenience — in particular, because there is a closed form for the KL-divergence between two Normal distributions. Further, since we never directly observe any values for \( Z \), the prior chosen need not match any real-world observations.

Distribution for Evidence. Next we consider \( P(X|Z) \). Here, mathematical convenience is not the only consideration. The distribution we choose needs to match reality since in practice, we do observe evidence, and the distribution we choose should match the evidence that we see in practice. A further complication is that BAYOU uses different kinds of evidence simultaneously during both training and inference. For example, the version of BAYOU described in this paper can make use of contextual information, type information, and desired API calls, so in fact, \( X = \langle X_{\text{Cxt}}, X_{\text{Types}}, X_{\text{Calls}} \rangle \). Yet another consideration is that we wish the framework to be flexible enough to easily accommodate other forms of evidence in the future.

To handle all of this, we simply redefine our notion of an evidence vector. First, let \( X_{\text{Cxt}} \) be composed of \( X_{\text{Cxt},1}, X_{\text{Cxt},2}, \ldots \); these are the individual pieces of contextual information. Likewise, let \( X_{\text{Types}} \) be composed of \( X_{\text{Types},1}, X_{\text{Types},2}, \ldots \); this is the set of types present in the code. Similarly, \( X_{\text{Calls}} \) has one entry for each API call given. Let \( f_{\text{Cxt}} \) be some arbitrary encoding function for each \( X_{\text{Cxt},i} \), which takes as input a piece of contextual information and maps it to a \( d \)-dimensional vector. We define \( f_{\text{Types}} \) and \( f_{\text{Calls}} \) similarly (more details on the encoding functions are presented in Section 5.2). Rather than the evidence vector \( X \) containing \( X_{\text{Cxt}}, X_{\text{Types}}, \) and \( X_{\text{Calls}} \), we instead define \( X \) so that it contains the encoded evidence:

\[
X = \langle f_{\text{Cxt}}(X_{\text{Cxt},1}), f_{\text{Cxt}}(X_{\text{Cxt},2}), \ldots, f_{\text{Types}}(X_{\text{Types},1}), f_{\text{Types}}(X_{\text{Types},2}), \ldots, f_{\text{Calls}}(X_{\text{Calls},1}), f_{\text{Calls}}(X_{\text{Calls},2}), \ldots \rangle
\]

\[
= \langle X_{\text{Cxt},1}, X_{\text{Cxt},2}, \ldots, X_{\text{Types},1}, X_{\text{Types},2}, \ldots, X_{\text{Calls},1}, X_{\text{Calls},2}, \ldots \rangle
\]

And we then use a Normal prior on each encoded evidence so that, assuming independence among the various pieces of evidence, we have
\[ P(X|Z) = \left( \prod_i \text{Normal}(\bar{X}_{\text{Cxt},i}|Z, \sigma_{\text{Cxt}}^2) \right) + \left( \prod_i \text{Normal}(\bar{X}_{\text{Types},i}|Z, \sigma_{\text{Types}}^2) \right) + \left( \prod_i \text{Normal}(\bar{X}_{\text{Calls},i}|Z, \sigma_{\text{Calls}}^2) \right). \]

Hence, it is likely that evidence generated using a particular intent \( Z \) has an encoded representation that is similar or close to \( Z \).

Note that each type of evidence is given a different variance; this allows different types of evidence to be less or more strongly associated with the intent used to generate the evidence. A relatively high variance means that the encoded evidence can easily be far away from the intent.

One of the benefits of this model is that due to Normal-Normal conjugacy, it is easy to obtain the posterior distribution \( P(Z|X) \), which is itself Normal, assuming that each individual piece of evidence is independently generated using the intent \( Z \). Let

\[
\bar{X} = \left( \sigma_{\text{Cxt}}^{-2} \sum_i \bar{X}_{\text{Cxt},i} \right) + \left( \sigma_{\text{Types}}^{-2} \sum_i \bar{X}_{\text{Types},i} \right) + \left( \sigma_{\text{Calls}}^{-2} \sum_i \bar{X}_{\text{Calls},i} \right)
\]

and let

\[ n = n_{\text{Cxt}} \sigma_{\text{Cxt}}^{-2} + n_{\text{Types}} \sigma_{\text{Types}}^{-2} + n_{\text{Calls}} \sigma_{\text{Calls}}^{-2} \]

where \( n_{\text{Cxt}} \) is the number of pieces of contextual evidence, and \( n_{\text{Types}}, n_{\text{Calls}} \) are defined similarly. Then, from Normal-Normal conjugacy, we have

\[ P(Z|X) = \text{Normal} \left( Z \left| \frac{\bar{X}}{1+n}, \frac{1}{1+n} \right. \right). \tag{8} \]

**Distribution for Sketches.** In practice, \( P(Y|Z) \) is realized as a neural decoder that takes as input \( Z \) and then uses this value to generate a vector of probabilities corresponding to the possible productions that could fire in our abstraction language. Once the first production \( Y_1 \) is sampled using this vector, the decoder uses \( Z \) and \( Y_1 \) to generate a vector of probabilities corresponding to the next set of possible firings. Once the second production \( Y_2 \) is sampled, the prior two firings are used in conjunction with \( Z \) to generate \( Y_3 \). This process is continued until termination. Thus,

\[ P(Y|Z) = P(Y_1|Z)P(Y_2|Z, Y_1)P(Y_3|Z, Y_1, Y_2)\ldots \]

where each individual term on the right-hand side of the above equation is computed using a tree-based neural decoder. For more details, see Section 5.3.

**KL-divergence between Intent Prior and Posterior.** The BED formulation requires that we compute the KL-divergence \( D_{KL}[P(Z|X) \parallel P(Z)] \). Note that both \( P(Z|X) \) and \( P(Z) \) are Normal distributions. Since there is a simple closed-form for the KL-divergence of two Normal distributions, \( D_{KL}[P(Z|X) \parallel P(Z)] \) in Equation 7 can easily be computed as:

\[
\frac{d}{2(1+n)} - \frac{d}{2} \left( 1 + \log \frac{1}{n} \right) + \frac{1}{2} \left( \frac{\bar{X}}{1+n} \right)^T \left( \frac{\bar{X}}{1+n} \right).
\]
Some Notes on Training. Given this, the set of parameters $\mathbf{M}$ to learn consists of the three standard deviations $\sigma_{\text{Cxt}}, \sigma_{\text{Types}}$ and $\sigma_{\text{Calls}}$ as well as all of the parameters of the neural decoder associated with $P(Y|Z)$. While not contained in the set of model parameters (since they are used to encode the evidence to produce $X$ and hence do not directly parameterize $P(X,Y)$), the parameters associated with the three encoding functions $f_{\text{Cxt}}, f_{\text{Types}}$ and $f_{\text{Calls}}$ are nevertheless learned simultaneously with $\mathbf{M}$. Our learning method resembles the gradient ascent algorithm used for variational autoencoders.

Why Bayesian? The key way in which the BED differs from supervised or semi-supervised variational autoencoders [26] is that it is designed to work with different categories of evidence, where each category contains an unordered set of individual pieces of evidence. These sets can have widely different sizes in each training and inference instance, and the different categories can be less or more indicative of the identity of the latent intent. The Bayesian update used to obtain the posterior $P(Z|X)$ naturally handles all of these issues. It does not matter whether $X$ is missing certain types of evidence, or whether $X$ has an abundance of others; Bayes’ rule still applies and provides a principled way to obtain $P(Z|X)$. In effect, Bayes’ rule (via Equation 8) prescribes something that resembles an average pooling over all of the evidence, weighted on the quality of the evidence.

5.2 Encoders in BAYOU

The task of the neural encoder is to implement the encoding function $f$ for evidence. We pick the encoding function for contextual evidence, $f_{\text{Cxt}}$, for demonstration. $f_{\text{Cxt}}$ accepts a piece of evidence $X_{\text{Cxt},i}$ as input and maps it into a vector in $d$-dimensional space.

To achieve this, first, we must have a canonical representation for every piece of contextual evidence. Typically, the one-hot vector representation is used for this purpose. Let $|\text{Cxt}|$ be the size of the vocabulary of contextual evidence. The one-hot vector of a given context $X_{\text{Cxt},i}$, denoted $X'_{\text{Cxt},i}$, is a vector of size $|\text{Cxt}|$ where all elements all zero, except for one element of value 1 whose index corresponds to the index of the evidence $X_{\text{Cxt},i}$ in the vocabulary. Now, all pieces of contextual evidence will have a fixed width representation and can be consumed easily by a neural architecture.

Let $h$ be the number of neural “hidden units” in the encoder for contextual evidence, and let $W_h \in \mathbb{R}^{|\text{Cxt}| \times h}$, $b_h \in \mathbb{R}^h$, $W_d \in \mathbb{R}^{h \times d}$, $b_d \in \mathbb{R}^d$ be real-valued matrices (those named $W$ are referred to as “weight” matrices and those named $b$ “bias” matrices of the neural network). The encoding function $f(X_{\text{Cxt},i})$ can be defined as follows:

$$f(X_{\text{Cxt},i}) = (W_h \cdot X'_{\text{Cxt},i} + b_h) \cdot W_d + b_d$$

This would map any given piece of contextual evidence into a $d$-dimensional real-valued vector. The values of entries in the matrices $W_h$, $b_h$, $W_d$ and $b_d$ will be learned as part of the training process. The encoding functions for the other forms of evidence, $f_{\text{Calls}}$ and $f_{\text{Types}}$, can be defined analogously, with each having their own set of the above matrices and hidden units $h$.

5.2.1 Embedding the evidence. It is common in deep learning models to not provide inputs directly to the neural network, but to pass them through an “embedding” layer that maps inputs into an abstract embedding space. In many cases, this has shown to make the model learn from the data better, and we observe this in our experiments as well (see Section 7).

In our implementation, we use a topic-model based embedding for our input evidence. Topic models are used in natural language processing to automatically extract topics from a large number of “documents” containing textual data as words. To use topic models in our setting, we have to consider a set of evidence as an input (document), instead of considering each individual piece of evidence as a separate input. Specifically, again picking the contextual evidence $X_{\text{Cxt}}$ for illustration, each document is the set $X_{\text{Cxt}}$ and words are symbols from the vocabulary of contextual evidence.
Latent Dirichlet Allocation (LDA) [12] is a well-known topic model that models the generative process of documents in a corpus where each document $X_{\text{Cxt},i}$ contains a bag of words. The input to LDA is the number of topics to be extracted $T$. A trained LDA model can be queried for the topic distribution of a document $X_{\text{Cxt},i}$, denoted as $X^T_{\text{Cxt},i}$, which is a vector of size $T$ that sums to 1, where the element at index $k$ represents the probability that the document is from topic $k$. This represents our embedding for the given document.

LDA models a document as a distribution over topics, and a topic as a distribution over words in the vocabulary. An LDA model is thus characterized by the variables: (i) $p_\alpha$ and $p_\eta$, which are hyper-parameters of a Dirichlet prior that chooses the topic distribution of each document and the word distribution of each topic, respectively (ii) $X^T_{\text{Cxt},i}$, the topic distribution of document $X_{\text{Cxt},i}$, and (iii) $p_\beta_k$, the word distribution of topic $k$.

The result of training an LDA model is a learned value for all the latent variables $p_\alpha$, $p_\eta$, $X^T_{\text{Cxt},i}$ for each document $X_{\text{Cxt},i}$ in the training data, and $p_\beta_i$.

During inference, we are given a document $X_{\text{Cxt}}$, and we would like to compute the embedding for the document, $X^T_{\text{Cxt}}$. Since LDA has already learned a joint distribution over topics and documents, i.e., $P(X^T_{\text{Cxt}}, X_{\text{Cxt}})$, we can condition this distribution with the newly observed $X_{\text{Cxt}}$ to get a posterior distribution over $X^T_{\text{Cxt}}$. Gibbs sampling [16] is a technique that is typically used for this purpose, as it approximates this distribution efficiently. Then, obtaining the embedding $P(X^T_{\text{Cxt}})$ is simply equivalent to sampling a value from this distribution.

If such an embedding is used for the evidence, the only change to the encoder is that instead of the $|\text{Cxt}|$-sized one-hot vector for each piece of evidence $X_{\text{Cxt},i}$, we provide the $T$-sized topic distribution for the entire set of contextual evidence $X_{\text{Cxt}}$ as input. Of course, the domain of $W_h$ would now be $\mathbb{R}^{T \times h}$, while the rest of the matrices are unaffected.

The embedding for the other two forms of evidence $X_{\text{Calls}}$ and $X_{\text{Types}}$ can be defined analogously, with their own versions of $T$ and the parameters of the LDA model.

5.3 The BAYOU decoder

The task of the neural decoder is to implement the sampler for $Y \sim P(Y|Z)$. This is implemented recursively via repeated samples of production rules $Y_i$ in the grammar of sketches, drawn as $Y_i \sim P(Y_i|Y_{i-1}, Z)$, where $Y_{i-1} = Y_1, \ldots, Y_{i-1}$. The generation of each $Y_i$ requires the generation of a new “path” from a series of previous “paths”, where each path corresponds to a series of production rules fired in the grammar.

Since a sketch is tree-structured, we use a top-down tree-structured Recurrent Neural Network (RNN) similar to [48], which we elaborate in this section. First, similar to the notion of a “dependency path” in [48], we define a production path as a sequence of pairs $\langle (v_1, e_1), (v_2, e_2), \ldots, (v_k, e_k) \rangle$ where $v_i$ is a node in the sketch (i.e., a
term in the grammar) and \( e_i \) is the type of edge that connects \( v_i \) with \( v_{i+1} \). Our representation has two types of edges: sibling and child. A sibling edge connects two nodes at the same level of the tree that are under the same parent node (i.e., two terms in the RHS of the same rule), whereas a child edge connects a node with another that is one level deeper in the tree (i.e., the LHS with a term in the RHS of a rule). We consider a sequence of API calls connected by sequential composition as siblings. The root of the entire tree is a special node named root, and so the first pair in all production paths is (root, child). The last edge in a production path is irrelevant (\( \cdot \)) as it does not connect the node to any subsequent nodes.

As an example, consider the sketch in Figure 3, whose representation as a tree for the decoder is shown in Figure 10. For brevity, we use \( s \) and \( c \) for sibling and child edges respectively, abbreviate some classnames with uppercase letters in their name, and omit the first pair (root, \( c \)) that occurs in all paths. There are four production paths in the tree of this sketch:

1. (try, \( c \)), (FR.new(String), \( s \)), (BR.new(FR), \( s \)), (while, \( c \)), (BR.readLine(), \( c \)), (skip, \( \cdot \))
2. (try, \( c \)), (FR.new(String), \( s \)), (BR.new(FR), \( s \)), (while, \( s \)), (BR.close(), \( \cdot \))
3. (try, \( s \)), (catch, \( c \)), (FNFException, \( c \)), (T.printStackTrace(), \( \cdot \))
4. (try, \( s \)), (catch, \( s \)), (catch, \( c \)), (IOException, \( c \)), (T.printStackTrace(), \( \cdot \))

Now, given a \( Z \) and a sequence of pairs \( Y_i = (v_i, e_i) \) along a production path, the next node in the path is assumed to be dependent solely on \( Z \) and \( Y_i \). Therefore, a single inference step of the decoder computes the probability \( P(v_{i+1}|Y_i, Z) \). To do this, the decoder uses two RNNs, one for each type of edge \( c \) and \( s \), that act on the production pairs in \( Y_i \). First, as in the encoder, all nodes \( v_i \) are converted into their one-hot vector encoding, denoted \( v'_i \).

Let \( h \) be the number of hidden units in the decoder, and \( |G| \) be the size of the decoder’s output vocabulary, i.e., the total number of terminals and non-terminals in the grammar of sketches. Let \( W^c_{\cdot} \in \mathbb{R}^{h \times h} \) and \( b^c \in \mathbb{R}^{d} \) be the decoder’s hidden state weight and bias matrices, \( W^e_{\cdot} \in \mathbb{R}^{|G| \times h} \) and \( b^e \in \mathbb{R}^{h} \) be the input weight and bias matrices, and \( W^e_y \in \mathbb{R}^{h \times |G|} \) and \( b^e_y \in \mathbb{R}^{|G|} \) be the output weight and bias matrices, where \( e \) is the type of edge: either \( c \) (child) or \( s \) (sibling). We also use “lifting” matrices \( W_l \in \mathbb{R}^{d \times h} \) and \( b_l \in \mathbb{R}^{h} \), to lift the \( d \)-dimensional vector \( Z \) onto the (typically) higher-dimensional hidden state space \( h \) of the decoder.

Let \( h_l \) and \( y_l \) be the hidden state and output of the network at time point \( i \). We compute these quantities as follows:

\[
\begin{align*}
    h_0 &= W_l \cdot Z + b_l \\
    h^{c}_i &= W^c_{h} \cdot h_{i-1} + b^c_{h} + W^c_{v} \cdot v'_i + b^c_{v} \\
    h^{s}_i &= W^s_{h} \cdot h_{i-1} + b^s_{h} + W^s_{v} \cdot v'_i + b^s_{v} \\
    h_i &= \begin{cases} 
        \tanh(h^{c}_i) & \text{if } e_i = \text{child} \\
        \tanh(h^{s}_i) & \text{if } e_i = \text{ sibling} 
    \end{cases} \\
    y_i &= \begin{cases} 
        \softmax(W^c_{y} \cdot h_i + b^c_{y}) & \text{if } e_i = \text{child} \\
        \softmax(W^s_{y} \cdot h_i + b^s_{y}) & \text{if } e_i = \text{ sibling} 
    \end{cases}
\end{align*}
\]

where

\[
\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad \text{and} \quad \softmax(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad \text{for } j = 1 \ldots K
\]

\( \tanh \) is a non-linear “activation” function that converts any given value to a value between -1 and 1, and \( \softmax \) converts a given \( K \)-sized vector of arbitrary values to another \( K \)-sized vector of values in the range [0, 1] that sum to 1—essentially a probability distribution.

The type of edge at time \( i \) decides which RNN to choose to update the (shared) hidden state \( h_i \) and the output \( y_i \). Training consists of learning values for the entries in all the \( W \) and \( b \) matrices. During training, \( v'_i \), \( e_i \) and the target output are known from the data point, and so we optimize a standard
cross-entropy loss function (over all $i$) between the output $y_i$ and the target output. During inference, $P(v_{i+1} | Y_i, Z)$ is simply the probability distribution $y_i$, the result of the softmax.

A sketch is obtained by starting with the root node pair $(v_1, e_1) = (\text{root}, \text{child})$, recursively applying Equation 9 to get the output distribution $y_i$, sampling a value for $v_{i+1}$ from $y_i$, and growing the tree by adding the sampled node to it. The edge $e_{i+1}$ is provided as $c$ or $s$ depending on the $v_{i+1}$ that was sampled. If only one type of edge is feasible (for instance, if the node is a terminal in the grammar, only a sibling edge is possible with the next node), then only that edge is provided. If both edges are feasible, then both possibilities are recursively explored, growing the tree in both directions.

Remarks. In our implementation, we generate trees in a depth-first fashion, by exploring a child edge before a sibling edge if both are possible. If a node has two children, a neural encoding of the nodes that were generated on the left is carried onto the right sub-tree so that the generation of this tree can leverage additional information about its previously generated sibling. We refer the reader to Section 2.4 of [48] for more details.

6 COMBINATORIAL CONCRETIZATION

The final step in our synthesis algorithm is to sample programs given an environment $Env$ and a sketch $Y$, using the concretization distribution $(Prog \mid Y, Env)$. BAYOU implements this step using a technique that builds on existing methods for stochastic [41] and enumerative synthesis [14].

In a nutshell, the procedure performs a random walk in a combinatorial space of partially concretized sketches (PCSs). A PCS is a term obtained by “concretizing” a subset of the abstract method calls and abstract expressions in a sketch, i.e., replacing them by AML method calls and AML expressions. For example, the term $x_1.a(x_2); \tau_1.b(\tau_2)$, which includes an abstract method call and a “concrete” method call, is a PCS.

The state of the procedure at the $i$-th point of the walk is a PCS $H_i$. The initial state is $Y$.

Each state $H$ has a set of neighbors $\text{Next}(H, Env)$. This set consists of all PCS-s $H'$ that are obtained by concretizing a single abstract method call or expression in $H$, using variable names in a way that is consistent with $Env$ as well the types of all API methods and declared variables in $H$.

The $(i+1)$-th state in a walk is a sample from a predefined, heuristically chosen probability distribution $P(H_{i+1} \mid H_i, Env)$. The only requirement on this distribution is that it assigns nonzero probability to a state if and only if it belongs to $\text{Next}(H_i, Env)$. In practice, our implementation of this distribution prioritizes programs that are simpler and declare few new variables. The random walk ends when it reaches a state $H^*$ that has no neighbors. If $H^*$ is fully concrete (that is, an AML program), then the walk is successful and $H^*$ is returned as a sample. If not, the current walk is rejected, and a fresh walk is started from the initial state. The procedure never explicitly returns the error value $\bot$. We interpret this error value as nontermination, i.e., never generating a sample program.

We note that our procedure satisfies criteria 1-3 from Section 4.1. Criterion 1 holds because our random walk starts with the sketch $Y$ and incrementally concretizes it. As for Criterion 2, suppose the walk successfully ends at a state $H^*$. Since every step in the derivation of $H^*$ used names in a way consistent with $Env$ and $H$, $H^*$ can be typed under $Env^*$. Finally, because the neighbors of a PCS $H_i$ cover all ways of concretizing a single expression or call in $H_i$, and because our random walk can transition to each of these neighbors with nonzero probability, the only way the random walk can only fail to terminate if $Y$ has no concretization that can be typed under $Env$. This ensures Criterion 3.

We omit an exhaustive definition of the set of neighbors of a state $H$. Instead, we only present two illustrative cases of this definition: when $H$ is an abstract call to a method that returns a value, and when $H$ is a while-loop with an abstract guard.
• Suppose \( H \) equals \texttt{``call } \tau_0.a(\tau_1, \ldots, \tau_k)\texttt{''}, where \( a \) has a return type \( \tau \). Let \( H' \) the statement \( \texttt{let } z = \text{Sexp}_0.a(\text{Sexp}_1, \ldots, \text{Sexp}_k) \), where \( z \) is a fresh variable, and each \( \text{Sexp}_i \) is either a variable \( x_i \) bound by \( \text{Env} \) to the type \( \tau_i \), or a constant of type \( \tau_i \). Then \( H' \) is a neighbor of \( H \).

• Suppose \( H \) equals \texttt{``while } ([\text{Call}^1, \ldots, \text{Call}^m]) \texttt{do } H_1 \texttt{''}, where \( \text{Call}^j = \tau_0.a(\tau_1^j, \ldots, \tau_k^j) \). Let \( H' \) be the PCS \texttt{``while } \exp \texttt{ do } H_1 \texttt{''}, where \( \exp \) equals

\[
\text{let } z_1 = \text{Sexp}_0^1.a^1(\text{Sexp}_1^1, \ldots, \text{Sexp}_k^1) : \text{let } z_2 = \text{Sexp}_0^2.a^2(\text{Sexp}_1^2, \ldots, \text{Sexp}_k^2) :
\hspace{1cm} \ldots \text{let } z_m = \text{Sexp}_0^m.a^m(\text{Sexp}_1^m, \ldots, \text{Sexp}_k^m) : z_m.
\]

Here, each \( z_i \) is a fresh variable, and each \( \text{Sexp}_i^j \) is either a constant of type \( \tau_i^j \), or a variable \( x_i^j \) that is: (a) a variable bound by \( \text{Env}' \) to the type \( \tau_i^j \), or (b) a variable of type \( \tau_i^j \) drawn from \( \{z_1, \ldots, z_{j-1}\} \). \( H' \) is a neighbor of \( H \).

Of course, in the actual implementation of \textsc{Bayou}, we synthesize Java rather than AML code. Because of the simplicity of AML, this step can be accomplished by a standard transformation.

7 EXPERIMENTS

In evaluating our method, we seek answers to the following research questions:

R1. How effectively does the training of the BED model learn specifications?
R2. How accurately does the BED model capture user intent and produce the desired results?
R3. How is the accuracy of the model affected by:
   (a) the Bayesian framework that explicitly models uncertainty?
   (b) the topic-model based evidence embedding?
   (c) the abstraction level of the output sketches?
R4. What do programs synthesized by \textsc{Bayou} look like, qualitatively?
R5. How accurate is \textsc{Bayou} in generalizing to unseen evidence and programs?

7.1 Experiment Setup

| \( X_{\text{Calls}} \) | Min | Max | Median | Mean |
|-----------------------|-----|-----|--------|------|
|                       | 1   | 9   | 2      | 2.6  |

\( X_{\text{Types}} \)  
| \( X_{\text{Cxt}} \)  

\( X \)

To answer the research questions posed earlier, we utilized data from an online repository of about 1500 Android apps [6]. Since the apps were compiled in the APK format, we used an off-the-shelf decompiler, JADX [42], to generate their source code. Analyzing about 100 million lines of code that were generated, we extracted 98619 programs on the entire Android API namespace and the Java IO library. From each program Prog, we extracted the set \( X_{\text{Calls}} \) of API calls that it uses, the set \( X_{\text{Types}} \) of types that it uses, and its context \( X_{\text{Cxt}} \), as well as a sketch \( Y \). Figure 11 gives some statistics on the sizes of the evidence sets in the data.

Of this data, 78619 programs were randomly selected to be part of the training data and 20000 part of the testing data, to ensure that there was no overlap between the two. This (almost) 80-20 split was made à priori before any experiments. If Prog was in the training set, we generated a set of training examples \((X, Y)\), where \( X \) is a triple consisting of randomly chosen subsets of \( X_{\text{Calls}}, X_{\text{Types}}, \) and \( X_{\text{Cxt}} \).

Implementation and training. \textsc{Bayou} uses TensorFlow [2] to learn and employ the BED model, and the Eclipse Java AST model to implement the abstraction from Java to the language of sketches and the combinatorial concretization. The implementation consists of about 13000 lines of Python and Java code, and is available as open-source at [1].

Our hyper-parameters for training the BED model are as follows. For the topic-model based embedding of the input evidence, we used 256, 64 and 32 topics for the API calls, types, and contexts,
7.2 R1. Evaluation of Specification Learning

We first present the results of training the topic-model based embeddings. Figure 12 shows the top-3 words in the vocabulary of some sample topics that were learned for embedding each of the three types of evidence. The topics indicate that the model has been able to group together commonly
related pieces of evidence. For instance, topic 1 for API calls is about the properties of Android web clients, whereas topic 2 is about managing runtime processes, and topic 3 is about showing dialog boxes. Similarly, for context evidence, topic 1 is about input readers, topic 2 is about notifications, and topic 3 is about event handlers for user actions.

We then train the BED model using the trained embeddings for processing individual pieces of evidence into topic vectors. As we use a gradient descent algorithm for optimization, the objective function for the BED model is the negative value of Equation 7, usually called the loss function. Figure 13(a) shows the convergence of this loss function and each term in it over 100 epochs of training. It can be seen that during the optimization, the KL-divergence term increases whereas the other two terms decrease correspondingly. This is expected, as the KL-divergence signifies the divergence between the prior \( P(Z) \), which is assumed to be a unit Normal, and the posterior \( P(Z|X) \), which is learned from data and would not typically be a simple unit Normal. This coincides with the decrease in the values of the other two terms, indicating that the model is fitting the distributions \( P(X|Z) \) and \( P(Y|Z) \) to the data with good accuracy.

Once the model is trained, we now visualize the learned 32-dimensional latent space. To do this, we provide the evidence vector \((X_{Calls}, X_{Types}, X_{Cxt})\) from each of the training programs to the model, and sample \( Z \) from \( P(Z|X) \). Since each such \( Z \) is a 32-dimensional vector, we use a standard dimensionality reduction technique, t-SNE [28], to project them onto a 2-dimensional space. For the purpose of visualization, we label each \( Z \) with one of the types from \( X_{Types} \), which is a very coarse approximation of the input evidence. Figure 13(b) shows this 2-dimensional space, where each label has been coded with a different color. To avoid clutter we only plot the top-10 most frequent labels (types). It is immediately apparent from the plot that the latent space is clustered according to the types. This is an indication that the BED model has learned to distribute the latent space smoothly among the different specifications in the training data, while also distinguishing them from each other.

### 7.3 R2. Evaluation of the BED model

We now evaluate the ability of the method to synthesize programs accurately. To this end, we can make use of the 20000 programs in the testing data, however, there are several dimensions to this evaluation.

Firstly, we would like to measure the effect of the different types of evidence provided as input to the model. To do this, we design four experimental trials where from each test program, we provide only \( X_{Calls} \), only \( X_{Types} \), only \( X_{Cxt} \) and a randomly picked subset of each as input to the model. For the latter, randomly picking a subset of the evidence resulted in, on average, about 3.3 pieces of evidence per program, with the minimum being 1, median 3 and maximum 15.

Secondly, we need a metric to compare the synthesized codes with the original codes in the testing data. A syntactic equivalence check between programs is clearly unsuitable here because variable names in the synthesized code will most certainly not match those in the original code, and programming constructs such as for loops may be present in the original code, whereas the synthesizer could have produced (equivalent) while loops. Even a semantic equivalence check is difficult here due to the API-heavy nature of the programs, which requires semantically analyzing the entire Android API library—a task that is out of scope of this paper.

Notice that at the level of abstraction of sketches that we have defined in this paper, syntactic differences, such as those mentioned above, do not occur. This makes the sketches themselves a suitable representation for defining metrics to compare programs. We have therefore come up with several quantitative metrics that compare the sketch of the original program and the sketches sampled from the BED model, given evidence from the original program.
Fig. 14. Metrics to evaluate the accuracy of different models over 20000 test programs.

The evaluation methodology of each test program is as follows: we provide an evidence vector $X$ from the program to the BED model, sample a $Z$ from $P(Z|X)$, and then sample 100 sketches from the posterior distribution $P(Y|Z)$. We rank sketches by the number of times they were sampled, and define the following four metrics on the top-5 ranked results, where the Jaccard distance between two sets $A$ and $B$ is a standard measure to compare the diversity of sets, and is defined as: $1 - \frac{|A \cap B|}{|A \cup B|}$.

M1. In our first metric, we store in $A$ and $B$ the set of sequences of API calls (with full signature) made by the original and sampled sketches, respectively, and compute the closest (minimum) distance. It is a measure of how close to the original sketch were we able to get in terms of sequences of API calls.

M2. In the second metric, we store in $A$ and $B$ the set of API calls made by the original and sampled sketches, respectively, and compute the closest distance. This measures our closest distance in terms of the set of API calls.

M3. The third metric computes the minimum absolute difference between the number of statements in the original and sampled sketches, as a ratio of that in the original. A statement is either an API call, loop, branch or try-catch statement.

M4. In the fourth metric, we compute the minimum absolute difference between the number of control structures in the original and sampled sketches, as a ratio of that in the original. A control structure is any statement that is not an API call.

(a) M1. Average (standard deviation) of minimum Jaccard distance on the set of sequences of API methods called in the test program vs the top-5 results.

(b) M2. Average (standard deviation) of minimum Jaccard distance on the set of API methods called in the test program vs the top-5 results.

(c) M3. Average (standard deviation) of minimum difference between the number of statements in the test program vs the top-5 results.

(d) M4. Average (standard deviation) of minimum difference between the number of control structures in the test program vs the top-5 results.
Figure 14 shows the results of this evaluation, where each entry computes the average (and standard deviation in parenthesis) of the above metrics over the 20000 test programs. (Please focus only on the row labeled “BED” in each table; the other rows refer to competing models introduced in Section 7.4.) It takes the BED model about 7 seconds, on average, to sample and rank 100 sketches.

In Figure 14(a), we can see that when providing only \( X_{\text{Calls}} \) as evidence, we observe the Jaccard distance to be, on average, 0.40. That number becomes 0.73 and 0.67 when providing only \( X_{\text{Types}} \) and \( X_{\text{Cxt}} \), respectively, as inputs. This makes sense as the names of API methods, being more specific, help convey user intent better than types and context. A random selection of evidence is more realistic in practice (i.e., the Rand. column), where the score is 0.49. This means that in practice, we can expect the BED model to capture user intent and generate a sketch that produces the desired sequences about half the time.

Figure 14(b) shows the Jaccard distance metric on the set of API calls. Again, we see the trend that \( X_{\text{Calls}} \) provides more accuracy than the other two kinds of evidence. The typical input of a random set of evidence results in a Jaccard distance of 0.23. Similar results are also observed in the other two metrics, shown in Figure 14(c) and (d), where there is only a 2-3% difference in the number of statements and control structures between the original and the sampled sketches.

The standard deviation is higher for metric M1, because it is possible for a single incorrectly predicted API call to completely change all sequences produced by the sketch and make the Jaccard distance 1. Hence, many values of this metric are either 0 or 1 with little variation in between, leading to a high variance. Other metrics, such as M2, are not affected this much by a single API call leading to a lower variance in their results.

| Amount of Evidence | Metric | M1     | M2     | M3     | M4     |
|--------------------|--------|--------|--------|--------|--------|
| Max-3              |        | 0.50   | 0.23   | 4%     | 2%     |
|                    |        | (0.49) | (0.29) | (11%)  | (10%)  |
| Max-2              |        | 0.59   | 0.31   | 4%     | 2%     |
|                    |        | (0.48) | (0.33) | (12%)  | (11%)  |
| Max-1              |        | 0.77   | 0.51   | 6%     | 2%     |
|                    |        | (0.42) | (0.37) | (14%)  | (12%)  |

Fig. 15. Metrics on BED model varying the maximum number of pieces of evidence

We run a final experiment to evaluate the ability of BAYOU to generalize from extremely underspecified inputs. In this experiment, for each of the 20000 test programs, we randomly pick a selection of evidence as before, however, with a bound on the maximum number of pieces of evidence that can be picked. We vary this bound for values 3, 2 and 1, increasingly reducing the amount of information provided to BAYOU. This models a very practical scenario where a typical user would want to provide only 3, 2 or 1 piece(s) of evidence. For each case, we compute the four metrics as before.

Figure 15 shows the results of this evaluation. With a maximum of 3 pieces of evidence, we see the accuracy is almost similar to the typical case we saw before. With a maximum of 2 pieces of evidence, the accuracy is still reasonably close, with some difference in the Jaccard metrics. The most extreme case is when BAYOU is given just 1 piece of evidence—the least we can expect short of reading the user’s mind. Even in this extreme case, the results look decent: it is able to accurately produce, on average, a quarter of the desired API sequences and half of the API calls, and get reasonably close to the structure of the desired sketch. This shows that BAYOU is able to generalize quite well from extremely underspecified user inputs.

7.4 R3. Comparison with other related models

To answer research questions R3(a), (b) and (c), we design three other models that each implement a different functionality compared to our BED.

**Non-Probabilistic Encoder (NPE).** We evaluate the claim that our underlying Bayesian framework that explicitly models the uncertainty in \( Z \) by the distribution \( P(Z|X) \) is critical to obtaining good
accuracy. For this purpose, we implement a model that does not allow for this uncertainty by having a non-probabilistic encoder for the evidence \( X \).

The encoder in this model produces a deterministic neural encoding of \( X \), and the decoder consumes this neural encoding as its start state and probabilistically fires production rules through the softmax sampling (see Section 5.3). The encoding of a set of evidence is the concatenation of the encoding of each piece of evidence. This encoder can be seen to implement the high-level ideas of the encoder used in [34], although low-level details may differ as their inputs are I/O examples. The embedding of input evidence and the decoder are unchanged in this model.

**Non-Embedded Evidence (NEE).** We evaluate the effect of the topic-model based input evidence embedding by implementing a model that directly consumes each piece of evidence in \( X \) instead of their topic vector embedding. The encoder and decoder are unchanged in this model.

**Token-Level Sketches (TLS).** We evaluate the effect of the abstraction level of the sketches in our method. A natural question one might ask is: can the BED model be used to synthesize programs directly, without going through the abstraction level of sketches? More formally, suppose that instead of the complex abstraction function \( \alpha \) that maps Java (AML) programs to the grammar of sketches, we have a much simpler \( \alpha' \) defined as \( \alpha'(\text{Prog}) = \text{tokens}(\alpha(\text{Prog})) \), where tokens is a function that returns the sequence of tokens in \( \text{Prog} \). The grammar of sketches here would be \( \Sigma^* \) where \( \Sigma \) is the set of all possible tokens. Can this model be used to directly synthesize programs?

We observe that with the above \( \alpha'(\text{Prog}) \) that provides no abstraction, one cannot hope to solve the program synthesis problem directly using a probabilistic model. The reason is that the model is not aware of the environment \( \text{Env} \) in which synthesis would take place, which could contain variables that are arbitrarily different from those in the data the model was trained on. Synthesizing entire programs directly using this model would never (unless by coincidence) make use of the variables in \( \text{Env} \). At the least, we need to abstract the names of the variables when using such a model.

To be conservative, for this experiment we use the abstraction function \( \alpha'(\text{Prog}) = \text{tokens}(\alpha(\text{Prog})) \). That is, instead of tokenizing the program \( \text{Prog} \) itself (which has no hope for synthesis), we tokenize the abstraction of \( \text{Prog} \) provided by \( \alpha \). The decoder is then trained to produce a sketch token-by-token, and the combinatorial concretizer attempts to synthesize a program using the sketch and the environment \( \text{Env} \). The embedding of input evidence and the encoder are unchanged in this model.

**Results.** We evaluate each of these three models using the same metrics defined in Section 7.3, using the same evaluation methodology described before. We denote the metrics for these models using the rows labeled NPE, NEE and TLS respectively in Figure 14. It is clear from the tables that our BED model fares better than all of these models.

Particularly, in Figure 14(a) and (b), the NPE model results in large average Jaccard distances of 0.93 and 0.78 respectively (the standard deviation for metric M1 is lower here as the average is already quite worse, meaning it consistently produces bad results). This indicates that the Bayesian aspect of our BED model really contributes to the accuracy of the method. The NEE model is comparatively better than NPE, where the distances are 0.57 and 0.29. This is the closest to our BED model’s numbers, indicating that the LDA embedding does contribute to the accuracy, albeit to a lesser extent. Finally, the TLS model is particularly bad, because 52% of the sequence of tokens generated by the model either could not be parsed or contain malformed API calls (for example, there is a mismatch in the number or type of arguments). The metrics shown in the tables in Figure 14 for this model only consider the rest 48% of the sketches, and even then the model performs poorly compared to the other three. This shows that our level of abstraction of sketches is really key for our method to produce meaningful outputs.
| Input to BAYOU | Synthesized program ranked among the top-5 |
|----------------|------------------------------------------|
| import java.io.File;  
public class Program {  
void foo(File file) {  
    int ch = '\n';  
    ??  
}  
}  |

$X_{\text{Types}} = \{\text{FileWriter}\}$  
$X_{\text{Calls}} = \{\text{write}\}$  
$X_{\text{Cxt}} = \{\text{int}\}$

| Fig. 16. Qualitative usage scenarios of BAYOU. |
We finally perform a statistical significance test to compare the other models with our BED model. For each metric M and alternative model MOD in Figure 14, we pose the null hypothesis “The average values of M when BED and MOD are used are the same”. We use the Student’s t-test [46] with a threshold $\alpha$ of 0.05 and compute the p-value for each hypothesis. We find that all p-values are below the threshold, except for two: metric M2 for the NEE model when providing $X_{Types}$ (0.21) and metric M3 for the NEE model when providing $X_{Cxt}$ (0.46). This means that the BED model’s results are better, in a statistically significant way, than those for all alternative models other than NEE.

We did not compare our method with existing non-data-driven synthesizers for a simple reason: our benchmarks are significantly beyond the scope of these tools. For example, JSKETCH [21] synthesizes Java code, but expects the input to include a detailed program sketch. While tools like $\lambda^2$ [14] do not require sketches, they assume restricted programming languages and cannot synthesize programs that use complex APIs.

7.5 R4. Qualitative Usage Scenarios of Synthesis

To give a sense of the quality of the end-to-end synthesis, we present and discuss a few usage scenarios for BAYOU. In each scenario, we start with a partially written skeleton program and provide some evidence towards what we (as the user) would like the synthesized program to achieve. We then pick a single program in the top-5 results returned by BAYOU and discuss it. Figure 16 shows three such example usage scenarios.

In the first scenario, we would like to write a single character stored in an int variable to a file using the Java IO library. We provide three pieces of evidence: the API call write, the datatype FileWriter and the context int (that stores the character). The program synthesized by BAYOU automatically defines a FileWriter and BufferedWriter, because, as we saw before, reading/writing in Java is typically done in a buffered manner. It then writes the character to the file and correctly closes the writer, even though this call was not explicitly specified in the input evidence. Finally, it also manages to catch the possible IOException that could be thrown by the call to write.

In the second scenario, we are given an Android Bluetooth adapter as argument (typically the phone’s adapter) and a hardware address to connect to as a String. We seek to synthesize code to obtain a stream to read data from the device at the given address, using a socket. We provide as evidence the datatype BluetoothSocket and the API call getInputStream that obtains the socket’s stream for reading. Given this, BAYOU is able to synthesize some relatively complex code. Since there is no BluetoothSocket in scope, it first creates a socket in two steps. It first calls getRemoteDevice, passing the address as argument, to obtain the device to create the socket to. Then, it creates the socket by invoking the createRfcommSocketToServiceRecord method. This method, however, requires a UUID for the transaction, for which BAYOU synthesizes code to automatically create a random UUID. It then obtains the input stream of the newly created socket. In addition, it also obtains the output stream, in case the user would like to write to the socket as well. Note that none of the steps in this process were explicitly specified by the user.

In the final scenario, we would like BAYOU to synthesize code to start preview mode in the phone’s camera using a fixed width and height for the dimensions of the preview. We provide as evidence simply the API call startPreview and the context int, representing the type of variables that store the two dimensions. With this input, BAYOU again synthesizes code that is quite complex. Primarily, without explicitly specifying that we are dealing with the camera API (note that we did not provide any datatype evidence), BAYOU is able to understand our intent.

The synthesized code first opens the back-facing camera by invoking Camera.open and obtains its parameters. Then, because of the given int context, it sets the preview width and height accordingly using the two variables in scope. In addition, it also sets the picture size (if a picture is taken) to be of the same dimensions. It then correctly sets the parameters of the camera to these modified
| Model | Metric | M1     | M2     | M3     | M4     |
|-------|--------|--------|--------|--------|--------|
| BED   |        | 0.67   | 0.31   | 5%     | 3%     |
|       |        | (0.46) | (0.32) | (12%)  | (13%)  |
| NPE   |        | 0.98   | 0.83   | 8%     | 4%     |
|       |        | (0.15) | (0.27) | (14%)  | (13%)  |
| NEE   |        | 0.75   | 0.38   | 6%     | 4%     |
|       |        | (0.42) | (0.32) | (12%)  | (14%)  |
| TLS   |        | 0.99   | 0.79   | 61%    | 17%    |
|       |        | (0.10) | (0.25) | (26%)  | (32%)  |

(a)

Fig. 17. (a) Metrics in the Rand. column of Figure 14 evaluated only on unseen evidence and sketches, (b), (c) Synthesizing a program that was unseen in the training data parameters, and then starts the preview. Again, note that BAYOU was able to synthesize all these intermediate steps starting with only two simple pieces of evidence.

7.6 R5. Generalization to unseen data

In this section we evaluate the ability of BAYOU to generalize to unseen data. The specific question that we ask is: how accurately does the model capture relations between user intent (evidence) and sketches that were never seen during training?

To evaluate this question, we gather a subset of the testing data whose data points, consisting of evidence-sketch pairs \((X, Y)\), never occurred in the training data. We then evaluate the same metrics in Figure 14 but only on this subset. We focus on the random subset of the evidence provided as input to the model (Rand. column). Figure 17(a) shows the results of this evaluation on the subset of 10681 unseen test data points. The metrics show that the BED model still fares better than the other models in generalizing to unseen data, and results in similar accuracy as when providing a maximum of 2 pieces of evidence, in Figure 15.

We illustrate the end-to-end implication of this through a concrete scenario. Suppose the user would like to read data from an Android bluetooth socket and handle exceptions that may arise by printing the stack trace. The user starts with the draft program in Figure 17(b) that has a BluetoothSocket object, and provides as evidence the API calls `readLine` and `printStackTrace`. Note that this programming task involves three very different APIs: Android Bluetooth, Java I/O and exceptions. We confirmed that none of the programs in the training data use these three APIs together, and so this is a completely unseen task for BAYOU.

Given this, BAYOU is able to synthesize the program in Figure 17(c) in one of the top-5 ranked programs. It is able to figure out that an InputStreamReader is to be used, automatically obtains the stream from the Bluetooth socket using calls that were not explicitly specified by the user,
and reads from it using a BufferedReader. Moreover, if an exception is thrown—specifically, an IOException—the code calls printStackTrace to handle it, as desired by the user. This is a concrete example that shows that BAYOU can synthesize programs never encountered during training.

8 RELATED WORK

Syntax-guided Synthesis. Many efforts in the programming language community [3, 14, 19, 33, 44] target synthesis using search and constraint solving. A common strategy in this area is syntax guided synthesis, where one puts syntactic limitations on the space of feasible programs [5]. This is done either by adding a human-provided sketch to a problem instance [44], or by restricting synthesis to a narrow DSL [9, 19, 35]. In contrast, our approach infers the syntactic constraints relevant for a synthesis task dynamically. Also, unlike prior approaches, we directly model uncertainty and ambiguity in a user-written specification. At the same time, because our method completes inferred sketches using tools from syntax-guided synthesis, it can leverage advances in this area. For example, BAYOU uses known ideas in syntax-guided stochastic [41] and enumerative [14] synthesis to concretize sketches into code.

Neural Program Induction. There is an emerging literature on neural program induction, the problem of learning end-to-end differentiable, neural models of programs from data [17, 23, 27, 30, 39]. The neural architectures used here are typically obtained by augmenting recurrent nets with capabilities inspired by discrete models like Turing machines [17] and stack machines [23]. The most basic difference between these methods and ours is that our programs are in a high-level, interpretable representation, as opposed to neural nets.

Neural Program Synthesis. The body of work most closely related to ours combines insights from deep learning and symbolic program synthesis. Terpret [15] and Neural Forth [40] use neural learning over a set of user-provided examples to complete a user-provided sketch. The biggest difference between these methods and ours is that they frame each synthesis task as a new learning problem. In contrast, our method learns a model of how to design programs once and for all, and uses this model in many synthesis tasks.

Among methods that learn reusable synthesis procedures from data, DeepCoder [7] uses neural techniques to speed up the synthesis of Flashfill [20] programs. Specifically, it uses a neural net to predict binary attributes of the solution to a synthesis problem, and uses these predictions to set goals for a combinatorial synthesizer. In contrast, our neural net generates detailed sketches that can closely guide combinatorial synthesis. Doing so is key to success in our more complex problem domain, but requires a more sophisticated neural model. In neuro-symbolic synthesis [34] and RobustFill [13], a neural architecture is used to encode a set of input-output examples and decode the resulting representation into a Flashfill program. One key difference between these methods and ours is that we learn over abstractions rather than programs. Second, we use a probabilistic model of latent intent, while these methods use nonprobabilistic encoders of the sort mentioned in Section 7. Finally, these methods are trained on synthetic data, while our work learns from a corpus of real-world programs.

Learning from Code Corpora. Another closely related literature focuses on learning hidden specifications [37] and syntactic patterns [4, 10, 31, 32, 37] from code in large software repositories. In general, these methods do not learn the sort of association between syntax and evidence that is key to our approach. The most relevant efforts [10, 36, 38] use statistical language models for code completion. However, unlike our work or other works on program synthesis, these methods are not driven by user-defined specifications.

Variational Auto-Encoders (VAEs). VAEs [25, 26] are a class of generative, probabilistic models that are closely related to our BED model. Similar to our model, VAEs also feature a latent variable...
Z that is assumed to generate the data to be modeled. Supervised (or semi-supervised) versions of VAEs exist that take as input a value X, such as our evidence, and encode that value to produce a distribution over Z from which a particular value Z is sampled, which is then decoded to produce a value for the dependent variable Y.

Unlike VAEs, however, BEDs are designed for the case when X is complex, consisting of many individual components, i.e., when X consists of \(X_1, X_2, \ldots\) X is a set of evidence vectors, where the number of items in each X may vary. Upon observing the evidence X, the BED uses Bayes’ rule and Normal-Normal conjugacy to obtain a posterior \(P(Z|X)\). We then sample a value Z from this distribution, and decode to obtain a set of parameters for the dependent variable Y.

9 CONCLUSION

We have given a data-driven method for program synthesis from ambiguous, incomplete specifications. Our key contribution is a Bayesian statistical model that correlates evidence about the nature of programming tasks, sketches of programs implementing the tasks, and the intent behind the tasks. While solving a synthesis problem, we use this model to infer sketches relevant to the problem, then concretize these sketches into code. We have realized this model using the new neural architecture of BEDs. We have implemented our ideas in BAYOU, a system for the synthesis of API-heavy code.

As for future work, a key feature of our approach is that it can support multiple forms of evidence simultaneously. While BAYOU currently supports three kinds of evidence, future work will explore others. For instance, such evidence can include input-output examples, traces, types, natural language descriptions, and even visual descriptions of program behavior.

A second direction involves richer interplay with constraint-based methods. Currently, type safety is the only correctness property we enforce with full certainty. One can imagine extensions where the output distribution over programs is conditioned on a user-defined correctness constraint.

Acknowledgements. This research was supported by DARPA MUSE award #FA8750-14-2-0270 and a Google Research Award. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] 2017. Github Bayou page. http://www.githubusercontent.com/capergroup/bayou. (2017). [Online; accessed 25-Jun-2017].
[2] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernando Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. (2015). http://download.tensorflow.org/paper/whitepaper2015.pdf
[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In CAV. Springer-Verlag, 934–950.
[4] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 472–483. DOI: http://dx.doi.org/10.1145/2635868.2635901
[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emin Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In FMCAD. 1–17.
[6] Androiddrawer. 2017. AndroidDrawer. http://www.androiddrawer.com. (2017). [Online; accessed 06-Jul-2017].
[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. arXiv preprint arXiv:1611.01989 (2016).
[8] David Barber. 2012. Bayesian reasoning and machine learning. Cambridge University Press.
[36] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. 761–774. DOI: http://dx.doi.org/10.1145/2837614.2837671
[37] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from Big Code. In POPL, Vol. 50, 111–124.
[38] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), 419–428.
[39] Scott Reed and Nando De Freitas. 2015. Neural programmer-interpreters. arXiv preprint arXiv:1511.06279 (2015).
[40] Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. 2016. Programming with a differentiable Forth interpreter. CoRR, abs/1605.06640 (2016).
[41] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic program optimization. Commun. ACM 59, 2 (2016), 114–122.
[42] skylot. 2017. JADX - Dex to Java decompiler. https://github.com/skylot/jadx. (2017). [Online; accessed 06-Jul-2017].
[43] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In Asian Symposium on Programming Languages and Systems. Springer, 4–13.
[44] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In ASPLOS, 404–415.
[45] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2012. Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer (STTT) (2012), 1–22.
[46] Student. 1908. The probable error of a mean. Biometrika (1908), 1–25.
[47] Phillip D Summers. 1977. A methodology for LISP program construction from examples. Journal of the ACM (JACM) 24, 1 (1977), 161–175.
[48] Xingxing Zhang, Liang Lu, and Mirella Lapata. 2016. Top-down Tree Long Short-Term Memory Networks. In NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016. 310–320.