Deep NLP-Based Co-evolvement for Synthesizing Code Analysis from Natural Language

Zifan Nan
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
znan@ncsu.edu

Hui Guan
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, Massachusetts, USA
huiguan@cs.umass.edu

Xipeng Shen
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, California, USA
liao6@llnl.gov

Abstract

This paper presents *Deepsy*, a Natural Language-based synthesizer to assist source code analysis. It takes English descriptions of to-be-found code patterns as its inputs, and automatically produces ASTMatcher expressions that are directly usable by LLVM/Clang to materialize intended code analysis. The code analysis domain features profuse complexities in data types and operations, which make it elusive for prior rule-based synthesizers to tackle. On the other hand, machine learning-based solutions are neither applicable due to the scarcity of well labeled examples. This paper presents how *Deepsy* addresses the challenges by leveraging deep Natural Language Processing (NLP) and creating a new technique named *dependency tree-based co-evolvement*. *Deepsy* features an effective design that seamlessly integrates *Natural Language dependency analysis* into code analysis and meanwhile synergizes it with *type-based narrowing* and *domain-specific guidance*. *Deepsy* achieves over 70.0% expression-level accuracy and 85.1% individual API-level accuracy, significantly outperforming previous solutions.

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

Keywords: Program synthesis, ASTMatcher, natural language programming, compiler

ACM Reference Format:
Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. 2021. Deep NLP-Based Co-evolvement for Synthesizing Code Analysis from Natural Language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446852

CC '21, March 2–3, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8325-7/21/03…$15.00
https://doi.org/10.1145/3446804.3446852

1 Introduction

This paper presents the first systematic exploration on synthesizing code from Natural Language (NL) to assist program analysis.

Motivation. Source code analysis is fundamental for software quality controls, underpinning various program optimizations, software debugging, security, and so on. Developing programs for code analysis has been a daunting task for general programmers, for the needs to dive deep into a usually complex compiler infrastructure to understand the bolts and nuts before making any meaningful extensions.

ASTMatcher [3] is a popular tool created to ease the process. As a module in Clang/LLVM [22], it provides a set of APIs for programmers to use to construct Abstract Syntax Tree (AST) matching expressions. Such expressions can then be integrated into a Clang program to find some part of the AST or the source code of a given program of interest. Table 1 gives three examples of nested ASTMatcher API calls for finding some code patterns in programs.

However, ASTMatcher has not received much adoption beyond expert developers. The primary reason is that, like other program analysis tools, ASTMatcher offers a large set of APIs that a programmer needs to learn before using them to express the code patterns of their interest. In ASTMatcher, there are over 500 APIs, and a large portion of them appear quite useful in many code analysis tasks: In a code analysis tool Clang-tidy, for instance, 310 ASTMatcher APIs are used in 214 source code files, and 110 APIs among them appear more than 10 times. The APIs have many subtle differences and various usage conditions. To use them, programmers need to go through a long learning process, and even after that, often have to look up the references frequently

Table 1. Examples of ASTMatcher expressions.

#	English description	Matcher expression
1	Find for statements whose init portion declares a single variable which is initialized to the integer literal 0.	forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(hasInitializer(integerLiteral(equals(0))))))));
2	Return a binary operator, [b]. [b]'s name is "=". [b]'s right hand side is integer 0.	binaryOperator(hasOperatorName("="), hasRHS(integerLiteral(equals(0))));
3	Search for all the functions that use a particular global variable named "PI".	functionDecl(hasDescendant(declRefExpr(to(varDecl(hasGlobalStorage(), hasName("PI"))))));

when writing code analysis programs. This problem of ASTMatcher typifies the major hurdle standing between general programmers and source code analysis.

The complexities of source code analysis make it elusive for prior synthesis solutions. Prior work has tackled domains consisting of only a few kinds of operations or simple data type systems. For instance, text editing [7] contains around 40 operations on strings or integers. The popular domain, regular expression synthesis [21, 28, 31, 46], contains only a handful of operations on characters. No DSLs in the recent studies [14, 24] have more than 50 operations and all work on data of simple types. In contrast, ASTMatcher contains over 500 APIs dealing with over 200 types of data. These data types form a full-fledged class hierarchy with many layers of type inheritance.

Most recent program synthesis work [7, 10, 16, 25, 26, 31, 36, 45] builds on statistical machine learning methods. These methods have used thousands or even millions of examples [10] for learning. Given the profuse complexity of the ASTMatcher domain, applying those methods would need even more examples. But there are not yet many real-world ASTMatcher examples—due to the exact barrier that this work tries to remove. And moreover, in this context, it can be difficult for users to come up with examples. Depending on what training is used, an example needed by prior methods could be an ASTMatcher specification or a whole program with codelets labeled for each code pattern of interest.

On the other hand, previous rule-based methods [23] heavily rely on data types for selecting constructs to use and for deciding their relative positions in the synthesized expression. Although it works on the previous domain with simple data types, its effectiveness quickly reduces when facing the complex polymorphism permitted by the full-fledged class hierarchy of ASTMatcher (as our experiments confirm, Section 5.2).

Novelty. This work introduces Deepsy[1], an NL-based code synthesizer for source code analysis. Deepsy takes English descriptions of to-be-found code patterns (e.g., column two in Table 1) as its inputs, and automatically produces ASTMatcher expressions (e.g., column three in Table 1) that can be used by LLVM/Clang to materialize the intended code analysis.

Deepsy addresses the aforementioned challenges through two distinctive features. First, it emphasizes the use of deep NLP, particularly dependency-based NL parsing, to guide the synthesis. Second, it uses an innovative technique, named *dependency-based co-evolvement*, to address the entwined challenges in API matching and ordering in the synthesis process. *API finding* and *API ordering* are the two main tasks in the synthesis. The former is about finding the suitable APIs for each part of the input English query, and the latter is about putting these APIs in an appropriate order to compose the final ASTMatcher expression. They depend on each other. Ordering obviously needs to know what APIs to order; inversely, the appropriate APIs to choose depends on their orders in the final expression—for instance, the returning type of a later API (e.g., `declStmt()` in Example 1 Table 1) needs to match with the type expected by its caller API (e.g., `hasLoopInit()`). Although both tasks existed in prior code synthesis problems, they were mitigated by either a large number of examples or the simplicity of the target domain. Unfortunately, neither of the conditions hold for ASTMatcher. Deepsy addresses the special challenges through a five-stage continuous evolvement scheme enabled by *dependency-based co-evolvement*. The approach makes *API finding* and *API ordering* able to leverage each other's evolving results throughout their own refinements, effectively resolving their tangled concerns.

We evaluate Deepsy through several experiments, in which, Deepsy generates the matcher expressions for a set of English queries. Deepsy achieves 70.0% accuracy at the entire expression level, significantly outperforming the 52% accuracy by previous rule-based methods. At the individual API level, Deepsy gives 85.1% prediction accuracy. The results demonstrate the effectiveness of the novel design of Deepsy in code synthesis for complex domains with scarce labeled examples.

This work has a specific focus, ASTMatcher in Clang/LLVM. The focus makes Deepsy relevant to a broad range of users for the popularity of Clang/LLVM. Deepsy can help lower the barrier to code analysis for software tool developers, programming educators or coding learners who need to find certain types of code segments from code repositories. For many of such analysis tasks, only front-end source code pattern analysis is sufficient, in which, writing ASTMatcher expressions is the most difficult part while the rest is largely boilerplate code easily derivable from some templates an IDE may provide. Meanwhile, the design and techniques of Deepsy may provide some general insights to other code synthesis problems in domains with scarce labeled examples.

This work makes the following main contributions:

- It presents one of the first NL-based ASTMatcher synthesizers for source code analysis and demonstrates

[1] Deep NLP-Guided Natural Language-Based Synthesizer for Code Analysis

Table 2. Examples of AST matchers.

Matcher	Category	Input Type	Return Type	Description
`forStmt`	Node	<ForStmt>	<Stmt>	Matches for statements.
`varDecl`	Node	<VarDecl>	<Decl>	Matches variable declarations.
`hasLoopInit`	Traversal	<Stmt>	<ForStmt>	Matches the initialization statement of a `for` loop
`hasName`	Narrowing	std::string	<NamedDecl>	Matches NamedDecl nodes that have the specified name.

the power of deep NLP in tackling complex domains with scarce examples;

- It develops *dependency tree-based co-evolvement*, a novel design of NL-based synthesis process that leverages dependency relations while addressing the tangled concerns of API finding and API ordering;
- It evaluates the effectiveness of the techniques through a set of experiments and comparisons, demonstrating the effectiveness of NLP-based synthesis for program analysis.

2 Background

This section introduces the background on ASTMatcher and dependency-based NLP.

ASTMatcher APIs. ASTMatcher is a tool in Clang/LLVM that can be used to construct AST matching expressions that correspond to code patterns of interest. There are totally 505 ASTMatcher APIs, called **matchers**. These matchers are grouped into three categories: *node matchers*, *narrowing matchers*, and *traversal matchers*. The number of matchers in each category is 171, 159, and 175 respectively. Table 2 lists four matchers as examples.

Node matchers match a specific type of AST node. In Table 2, the `forStmt` matcher matches the *ForStmt* nodes inside an AST, and the `varDecl` matcher matches the variable declaration nodes inside an AST. *Narrowing matchers* match attributes on AST nodes, thus narrowing down the set of nodes of the current node type to match on. In Table 2, the `hasName` matcher matches the nodes that have a specified name. *Traversal matchers* allow traversal between AST nodes. For example, using the `hasLoopInit` matcher inside the `forStmt` matcher moves the matching focus from a *ForStmt* node to the initialization statement of that for loop.

The arguments and output of a matcher are usually associated with the type of the matched AST node (e.g. columns *Input Type* and *Return Type* in Table 2). The `varDecl` matcher, for instance, has input parameter type as *Matcher<VarDecl>* and return type as *Matcher<Decl>*. Deepsy leverages the input and return type to refine the mapping from NL to matchers.

An ASTMatcher expression is a sequence of matchers. It always starts with a node matcher. ASTMatcher has an *alternative restriction*: The callees (i.e., arguments) of a node matcher can be only the other two classes of matchers and vise versa. A matcher expression can be integrated into a Clang program to find AST nodes and code segments that match the target code patterns in a given program.

ASTMatcher features a complex data type system that contains a full-fledged class hierarchy. There are totally 209 major data types. These types form the class hierarchy of the AST in Clang. Most classes are the descendants of root classes such as `Decl`, `Stmt`, `Type`. For example, the `Decl` class has 21 sub-classes, and one of the sub-classes `NamededDecl` has 14 sub-classes, including `ValueDecl`, which has 7 sub-classes. A consequence of the type hierarchy is the entailed polymorphism: All descendants of a class can be type compatible at places where that class is expected. Polymorphism complicates the use of data types for expression synthesis. Previous rule-based synthesis methods strongly rely on data types for choosing the right components to use in an expression. The common presence of polymorphism in the ASTMatcher domain makes them hard to apply.

Dependency Tree. Dependency trees are the results of applying dependency parsing on sentences. Dependency parsing is an automatic approach that analyzes binary asymmetrical relations (called dependency relations) between words within a sentence [19]. A dependency relation is composed of a subordinate word (called the *dependent*), a word on which it depends (called the *governor*), and a dependency type between the two words. Figure 1 shows the dependency structure of an example sentence generated by the Standford CoreNLP dependency parser [30]. The dependency relations are represented as arrows pointing from a governor to a dependent. Each arrow is labeled with a dependency type. A dependency tree uses a tree to represent the dependency structure of a sentence. The root of the tree is the word in the sentence in a "root" dependency relation. Each node in the tree is a word in the sentence. A node n_1 is the parent of another node n_2 if n_1 is the governor of n_2.

3 Challenges and Solution Overview

For each word or phrase in a given English query, it may match with multiple matchers in their names or documentations. For instance, in Example 1 in Table 1, the keyword "declare" matches with 113 matcher APIs' (partial) names or documentations. Most of them are wrong choices for their mismatching with the context in the query. The challenge is how to determine the best choice.

An even harder challenge is in order. The final ASTMatcher expression is usually a series of matchers composed together in a certain order, as the examples in Table 1 have shown. The correct order, however, often differs from the appearing order of the corresponding words or phrases in the input query. The challenge is how to infer the right order of the matchers.

The two challenges, matchers and order, are entwined together. On one hand, the order of matchers clearly depends on what matchers are there. On the other hand, the appropriate matcher for a phrase depends on its position relative to other matchers. In Example 1 in Table 1, if the matcher API corresponding to word "init" follows `forStmt` in the final ASTMatcher expression, then it must be a matcher API that has a return type compatible with *Matcher<ForStmt>*. But if its correct position follows `varDecl`, it would need to be a

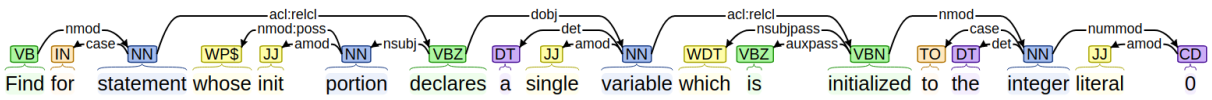


Figure 1. Dependency structure.

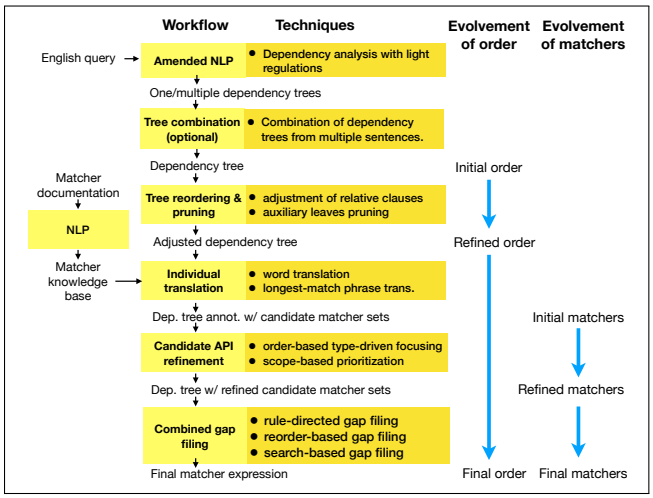


Figure 2. Overall workflow of Deepsy, its main techniques, and the enabled evolvement of the order and matchers of ASTMatcher expressions.

matcher of return type compatible with *Matcher<VarDecl>*. The inter-dependence between finding matchers and finding their right order makes the problem especially difficult to tackle. A simple divide and conquer strategy cannot work.

Our new solution, *dependency tree-based co-evolvement*, is designed to overcome these challenges by centering around the dependency tree of the input query, and co-evolving the matcher finding and the order finding together. The strategy allows it to fully leverage the dependency relations of the query and dependency relationship.

More specifically, it creates a set of candidate matchers for each phrase and word in the query, which are then refined and prioritized through *order-based type-driven focusing* and *scope-based prioritization*; the former capitalizes on argument and return types of matchers, and the latter employs the scopes of API documentation as heuristics. It employs the techniques, *dependency tree restructuring* and *combined gap filing*, to address the `order mismatching` and `tangled concerns`. The techniques gradually refine the order of matchers and also fill gaps between matchers with extra matchers to compose all together into one ASTMatcher expression. Next, we describe each stage of Deepsy based on the workflow in Figure 2.

4 Internals of Deepsy

This section describes all major stages in Deepsy by following a top-down order as shown in Figure 2.

4.1 Matcher Knowledge Base Construction

This stage builds a knowledge base. It is the only stage that happens ahead of time. The matcher knowledge base contains three parts: a *matcher table*, a *matcher graph*, and the *type hierarchy* of ASTMatcher. The *matcher table* contains the knowledge of all the matchers listed in the official AST Matcher Reference [3], as shown in Table 2. The knowledge base is constructed by parsing the original AST Matcher Reference automatically with a python script. The name of a matcher usually is a combination of two or three words or word abbreviations that imply the functionality of the matcher. The description is a sentence that briefly describes what this matcher matches. Deepsy leverages the name and description of a matcher to search for matcher candidates for a phrase in a query and then use the type properties to refine the mapping. The *matcher graph* is a directed graph. Each data type in ASTMatcher is represented as a node in the graph. A directed edge $x \rightarrow y$ is created and labeled with a matcher m if the matcher m has an input type y and return type x. The *type hierarchy* is the default class hierarchy defined in ASTMatcher.

4.2 Amended NLP

This stage is the first stage in the workflow of Deepsy. It takes an English query as its input, applies dependency-based NLP analysis (CoreNLP [30]) to the query to produce the dependency tree.

As the prior section has mentioned, a problem specific to the program analysis domain is terminology confusion. We use light regulations to address the problem. Specifically, we require all of the special words to be put inside double-quotes. So, `find for loops` should be written as `find "for loops"`. With the special words explicitly marked, Deepsy can easily replace them with some understudies before applying NLP to the input query. We use element_id as the understudy for a particular special word (we assign each special word with an id number); the NLP engine would label element_id as NN (noun).

With terminology confusions resolved by the amendment, the NLP engine analyzes the input query and produces the dependency result. Deepsy then represents the result in a dependency tree, as illustrated in Figure 3 (a). The root of the tree is the word that depends on no other words in the sentence (e.g., `Find` in Figure 3 (a)). If word X depends on word Y in relation Z, then in the tree, X is a child of Y and the edge (Y→X) carries Z on it. For instance, the top two nodes in Figure 3 (a) come from the dependency shown in the NLP result (Figure 1) that `for loops` (which has been replaced with `element_1`) is the object of `find` (represented with `dobj` relation).

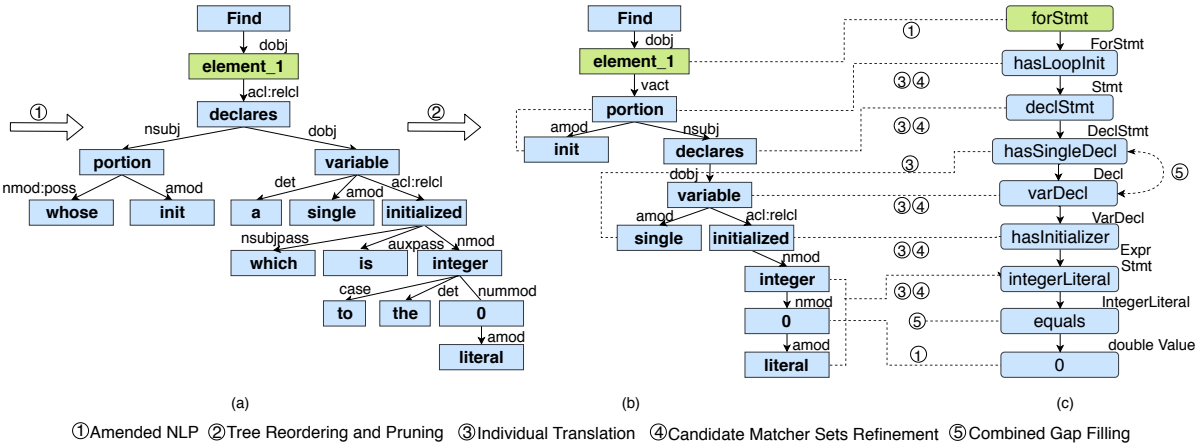


Figure 3. The synthesis process.

When creating the tree, Deepsy allocates a *candidate matcher list* for each node. This list will hold the potential matchers corresponding to the word in that node. At the end of this stage, all lists are empty except for those of special terms (element_id nodes). Because each of the special terms corresponds to one basic matcher (e.g., `forStmt` for `for loop`), Deepsy simply puts that matcher into the list when creating those tree nodes.

The dependency tree offers the underlying vehicle for all of the remaining stages in Deepsy to work on. It gives conveniences for Deepsy to leverage the words relations for the dependencies the tree explicitly carries, as we will see later.

4.3 Dependency Tree Combination

Many matcher expressions are easier to be described using multiple sentences. Deepsy allows users to use multi-sentence description as an input query. In this case, Deepsy parses each sentence independently using Amended NLP and then joins the dependency trees into one single dependency tree via the relations among tags.

One complexity in parsing multi-sentence description is to resolve coreference. Coreference occurs when two or more expressions refer to the same entity in a text. Finding the coreference relations is the prerequisite to merging dependency trees into a single one. The state-of-the-art coreference resolution results, however, are still not quite accurate yet (59.56% average F1 on multiple standard benchmarks [40]).

We propose *tagging*, a simple yet practical solution to circumvent the difficulty, in which, tags are used to mark the expressions that occur more than once. A tag is added at the first occurrence of the expression and then later used to refer to the expression whenever it is needed. The tag is called *label tag* in the former case and called *reference tag* in the latter one. An example tagged multi-sentence description is shown as follows:

```
S1: Find a for statement, [s].
S2: [s]'s initial portion declares a single
    variable, [v].
S3: [v] is initialized to the integer 0."
```

The tags include [s] and [v], which refer to "for statement" and "a single variable" respectively. When "for statement" appears for the first time in S1, a label tag [s] is appended to the expression. In S2, [F] is a reference tag used to represent "for statement".

Dependency tree combination generates a single dependency tree from a tagged multi-sentence description. Figure 4 illustrates the combination process. It follows two major steps: (1) The multi-sentence description is split into single sentences. Each sentence is transformed into a dependency tree using the amended NLP. We refer to the dependency tree of a sentence that contains a label tag as a *parent tree* and that contains the reference tag as a *child tree*. (2) Two dependency trees are connected by adding a new dependency relation between the root node of a child tree and the node corresponding to the label tag in a parent tree (called *label tag node*). Two possible dependency relations could be added depending on the relative positions of the node corresponding to the reference tag in the child tree (called *reference tag node*): If the reference tag node is the direct child of the root node in a child tree, it indicates that the child tree is a detailed explanation of the label tag. The child tree is appended to the label tag node with the dependency relation "relcl:new", implying a relative clause relation with "which" statement. Otherwise, the dependency relation is "relcl:poss:new", implying a relative clause relation with "whose" statement.

In figure 4, the dependency tree of S1 contains a label tag node "elemement_1". The dependency tree of S2 are attached to the label tag node with the dependency relation "relcl:poss:new". The dependency tree of S3 are attached to the label tag node "variable" with the dependency relation "relcl:new".

After a single dependency tree is created, all the reference tag nodes are deleted from the tree.

4.4 Tree Reordering and Pruning

Although the order of nodes in the dependency tree aligns with the suitable orders of matchers to a certain degree, there

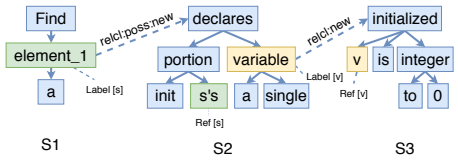


Figure 4. An illustration of dependency tree combination.

are some important discrepancies. One of them comes from the use of relative clauses in the input query. This stage of Deepsy fixes such discrepancies and additionally prunes away some trivial leaves nodes from the tree.

A *relative clause*, also called *adjective clause*, modifies a noun to specify some of its properties. As a clause, it has its own sentence elements, such as subject, verb, and object. For Example 1 in Table 1, there are two relative clauses.

Clause-1: It is led with the pronoun "*whose*" ("*whose init portion declares ...*");

Clause-2: It is led with the pronoun "*which*" ("*which is initialized to ...*").

When treating relative clauses, the dependency analyzer in NLP engines always considers that there is an acl:relcl (stands for relative clause modifier) dependency between the verb of the clause and the noun that the clause modifies. As a result, the verb is a child node of the noun, and the subject and object of the clauses become the child nodes of the verb, illustrated by the subtrees led by `element_1`→`declares` and by `variable` → `initialized` in Figure 3 (a).

In clause-1, `whose` indicates the possessive relation of a noun, and the clause means that "the init portion of element_1 (for statement) declares a variable". In this case, from ASTMatcher's point of view, `init portion` should have a closer relation with element_1 than `declares` has—in another word, the matcher of *init portion* should immediately follow the matcher of `element_1`, which is a discrepancy from the order of nodes in the dependency tree.

Note that such discrepancy does not happen to all relative clauses. In Clause-2, `which` refers to the `variable` and serves as the subject in the clause, and the clause means "a variable is initialized to the integer literal 0". In this case, the subtree gives an order of nodes that is consistent with the desired order of the corresponding matchers of those nodes.

So when dealing with this kind of discrepancy, Deepsy discerns the two kinds of cases. For relative clauses led by `whose`, Deepsy restructures the corresponding subtree through the following steps: (1) attaches the noun modified by `whose` to the parent node of the relative clause as its immediate child; (2) labels the edge between them with an `acl:relcl` dependency; (3) puts the original parent node of the noun as a child of that noun, and label that edge with a new dependency we introduce `vact`. For other relative clauses, Deepsy does no restructuring.

In addition, in this stage, Deepsy prunes trivial leaves (leaves with trivial words) from the tree. These leaves include several categories: (1) articles (`a, an, the`) that have "det" relation with some nouns; (2) relative pronouns (e.g., `whose, which`); (3) the copula verbs that have "cop" (stands for copula) relation with the complement of the copula verb (e.g., "is", "are" related with adjective); (4) dependent words in the "auxpass" (stands for passive auxiliary) dependency relation (e.g., "is" related with a verb) (5) words carrying relation `case` (stands for case marking) (e.g., `is` and `to` in Figure 3 (a)). These words are not useful for the synthesis process as they do not map to any matcher.

Figure 3 (b) shows the dependency tree of (a) after the restructuring and pruning.

4.5 Individual Translation

Based on the tree from the previous stage, this stage tries to find the potentially suitable AST matchers for each individual word or phrase in the query; this stage does not yet consider the relations among them. It fills the matcher lists of the nodes in the dependency tree through two passes of tree traversal.

Word Translation. In the first pass, for each node in the dependency tree (except those that have been replaced with `element_id`), Deepsy searches the matcher knowledge base for matchers whose names or descriptions contain a match with that word. Here, being a match means that the stem of that word is contained in the names or descriptions of the matcher. We use the WordNet [33] synonym list to gather the synonym words, which are also used to find a match inside the knowledge base.

After the first pass, all the matched matchers are put into the matcher list of every tree node as their candidate matchers. However, there are situations where, one matcher materializes a phrase rather than a single word. Example 2 in Table 1 shows such a case. In this example, "binary" has an 'amod' (adjective modifier) dependency on "operator", and together they form a phrase which matches matcher `binaryOperator`. In the second pass, Deepsy tries to identify such phrases and update the matcher lists accordingly. It does it through a longest-match scheme.

Longest-match scheme. The longest-match scheme is inspired by the longest-match principle many Scanners take when tokenizing strings [6]. For a node (say x) that has one or more modifiers, Deepsy creates a list `mod_list` and adds all its modifiers into the list. It then goes through the matcher list of node x. For each of the candidate matchers, it checks to see how many times the matcher also appears in the matcher lists of the nodes in its `mod_list`; the result is taken as the score of that matcher. The matchers with the highest score (ties can happen) are the longest matches for the phrase. Deepsy then keeps these matchers in the matcher list of node x, removes the other matchers from that list. It records with each of the matchers the list of modifiers that the matcher covers; if later, that matcher is selected for node x in the final ASTMatcher expression, no matchers will be taken from the matcher lists of the modifiers in x's covered list, as they are already covered in the expression.

4.6 Candidate Matcher Sets Refinement

In this stage, Deepsy tries to refine the matcher sets of the nodes in the dependency tree by leveraging two sources of information: types of the ASTMatcher APIs, and scopes of the matchers. We do that by creating two techniques, *order-based type-driven focusing* and *scope-based prioritization*.

Order-based type-driven focusing. Although data types alone are insufficient for this synthesis domain, they are still useful. As Section 2 mentions, every ASTMatcher has a return type and possibly one or more arguments of some types. If a matcher X serves as an argument of another matcher Y, the return type of X must be compatible with the type of the argument of Y. Reflected on the dependency tree, this principle implies that if the order of matchers follows the order of nodes in the tree, the return type of the matcher of a child node must be compatible with the type of the corresponding argument of the parent's matcher. *Order-based type-driven focusing* is to use such type compatibility as constraints to identify the promising matchers in a node's matcher list. It does that by temporarily assuming that the current order of nodes in the dependency tree is the correct order of matchers in the final ASTMatcher expression. Although typically most of the order is correct, it often still has some disparity from the final order. Therefore, *order-based type-driven focusing* does not eliminate the other matchers, only highlights the promising ones through special marks, such that later stages can come back to those "unpromising" matchers if necessary (detailed later).

Order-based type-driven focusing uses the partially finalized order to help refine choices of matchers. Although it helps, it may still leave many matcher lists with more than one promising matchers. A major reason is the full-fledged class hierarchy of AST in Clang, as mentioned in Section 2.

Scope-based prioritization. Deepsy takes another pass of the dependency tree to further discern the promise of the matchers by leveraging scopes. Here, the *scope* of a matcher refers to the number of non-trivial words (i.e., trivial words are defined in Section 4.4) that its description contains. The larger the scope is, the more specific the matcher is. For example, the description of the matcher `varDecl` is "Matches variable declarations" (three non-trivial words) while the description for the more specific one `parmVarDecl` is "Matches parameter variable declarations" (four non-trivial words). If a node's matcher whose description contains many words outside the node, it is a sign that the matcher could be too specific. For instance, for node "variable" in Figure 3 (b), one candidate matcher is `declaratorDecl` whose description is "Matches declarator declarations (field, variable, function and non-type template parameter declarations)." The matcher contains too many non-trivial words other than the word "variable" and thus is too specific for the node.

So in this pass, for each node, Deepsy computes the *extra scope* of every promising matcher in that node's matcher list; here, the *extra scope* is defined as the number of non-trivial words in that matcher's description that do not match with the words contained in that node. Deepsy sorts the promising matchers in the size of the *extra scope*; the ones with the smallest extra scope is regarded as the most promising matchers.

4.7 Combined Gap Filling

After all the prior stages, each node in the dependency tree usually has only one promising matcher that has the highest priority. But there are cases where some nodes do not have any promising matcher in their matcher lists. There are three cases.

(1) The node is part of a phrase of its parent node and its parent node's matcher (determined through the *longest-match scheme*) already covers this node. Such case has no problem.

(2) Some verbs (e.g., `use`) represent general relations between subjects and objects and do not match with any matcher's name or documentation directly. Such cases need special attention because these verbs may suggest the need for some matchers to connect the matchers of its parent and siblings or children.

Consider example 3 in Table 1, "find a function that uses a particular global variable". The word `use` here does not have any single corresponding matcher, but it links "function" and "variable", which are respectively translated to "`functionDecl()`" and "`varDecl()`". Both are *node matchers*; the word "use" should be translated into some narrowing or traversal matchers to meet the *alternative requirement* of ASTMatcher (Section 2).

(3) None of the matchers in a child node match with the type expected by the parent node. This case suggests two possibilities: the order of the parent and children nodes needs some adjustment or some connecting matchers need to be inserted between them.

To address the gaps caused by cases two and three, Deepsy employs a combination of three gap filling approaches.

Rule-directed gap filling. For the second case, based on observations, we create a set of rules for a set of commonly seen special verbs in that category. These rules help Deepsy come up with the matchers corresponding to these special verbs in a query. For the "use" example, the created rule is that if in dependency tree the matcher of the parent node of "use" expects an argument of type `decl`, and the matcher of the child node of "use" has `decl` as its return type, insert `hasDecendant(declRefExpr(to()))` in between. Or, more concisely,

```
decl–use–decl
⇒ decl–hasDecendant(declRefExpr(to()))–decl.
```

Reorder-based gap filling. For the remaining gaps, Deepsy explores reordering the parents and children nodes if the child node is an adjective modifier of the parent. This exploration is based on our observations that in the design of ASTMatcher, although most of time the matcher of an adjective modifier comes after a node it modifies, it may also appear before, depending on the matchers and contexts. In

the exploration, Deepsy uses type consistency (similar to *order-base type-driven focusing*) to check all the matchers in their matcher list, regardless of whether they are marked as "promising". The category of a matcher, *node, narrowing, traversal*, is taken as part of the type info.

Search-based gap filling. Finally, if there are still gaps between a parent matcher and a child matcher in the dependency graph, Deepsy resorts to type-driven search to fill them. Specifically, it does a breadth-first search on the *matcher graph*. The matcher graph is a directed graph in the matcher knowledge base. Each ASTMatcher data type is a node; an edge $x \rightarrow y$ carries label z if ASTMatcher z has a return type x and an argument of type y.

Let p and c be the parent and child matchers with gaps in between. The breadth-first search starts from the expected argument type of p, and stops if it reaches the return type of c or has reached the upper limit of the number of hops (set to 3 in our experiments). The hop limit prevents the search from falling into dead loops as the graph contains cycles.

The matcher `equals` in Figure 3 (c) is found through such a process. The return type of node `0` is `double value` (ASTMatcher treats all numerical values as `double value` or `unsigned value`), while the input type of the matcher `integerLiteral()` is `IntegerLiteral`. The search finds matcher `equals()` which takes `double value` as input and returns `IntegerLiteral`, which fills the gap.

The three gap filling methods focus on different cases; the combined approach takes advantage of them all, and is successful in addressing most gaps in our experiments (reported next).

5 Evaluation

We conduct a set of experiments to examine the efficacy of the Deepsy framework. Our experiments are designed to answer the following major questions: (1) What is the accuracy of Deepsy on synthesizing matcher expressions based on natural language inputs, and how does that compare with prior solutions? (2) How much benefit can we get from each decision made in the synthesis process? (3) How much time does the synthesis take?

We first describe the experiment settings in Section 5.1, then report our experiment results and comparisons in Sections 5.2 and 5.4 to answer the questions. In Section 5.3, we further provide a detailed error analysis in several representative cases.

5.1 Methodology

Dataset. For tests, we collect 90 real-world ASTMatcher expressions, with 82 from the usage of ASTMatcher in Clang-tidy [4], and 8 from ASTMatcher tutorial [5] and blogs [18]. The expressions from the tutorial and blogs have English descriptions, but those from Clang-tidy do not. We hired five graduate students to write English descriptions for the cases after receiving a 20-min tutorial of ASTMatcher. We end up with two single-sentence descriptions from two humans for each of 50 matcher expressions, one multi-sentence description for each of the remaining 40 matcher expressions. Totally, there are 140 English descriptions (listed in supplementary material).

Evaluation Metrics. We use four metrics to evaluate the performance: (1) *Overall accuracy*. This is the ratio between the number of *correctly* synthesized ASTMatcher expressions and the number of total test cases. A synthesized ASTMatcher expression is *correct* if it is identical to the ground truth matcher expression in terms of both the set of matchers and their relative order. (2) Accuracy on *easy* cases and *hard* cases. These metrics show the effects of the number of matchers in an expression on the accuracy of Deepsy. A matcher expression is *easy* if it contains less than 5 matchers; Otherwise, it is hard. (3) Individual matcher accuracy (*Inner accuracy*). It is the ratio between the number of correctly found matchers in a synthesized matcher expression and the total number matchers in the ground truth matcher expression. (4) Execution time. It is the time to synthesize a complete matcher expression from NLP parsing results.

Methods for Comparison. For overall accuracy of single sentence test cases, we have compared Deepsy with a prior rule-based method and two variants of Deepsy:

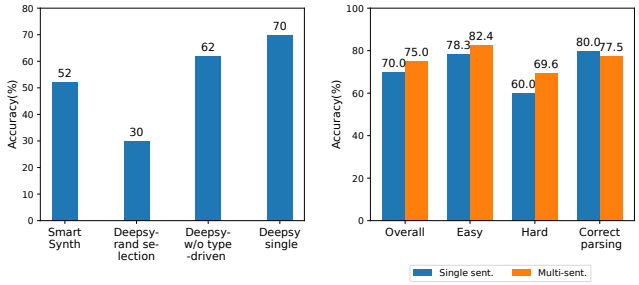
(1) *SmartSynth* [23]. We pick SmartSynth for several reasons. First, it is a rule-based synthesizer, requiring no labeled examples for training, which is a property necessary for ASTMatcher for the lack of labeled samples in the domain. Second, it showed promising results in synthesizing smartphone codelets (reported 90% accuracy), dealing with a domain with 106 operations/APIs. There are some synthesizers proposed later. They combine the rule-based method with machine learning, rather than to enhance its capability in tackling complex domains with scarce labeled data. In fact, the domains evaluated in later studies—such as text editing [7], SQL expression [24], spreadsheet programming [14] —are even smaller than the one SmartSynth tackles. SmartSynth is the most promising prior work that we have found for code synthesis in complex domains with scarce labeled data.

(2) *Deepsy- w/o type-driven*. This is Deepsy but without the order-based type-driven focusing step;

(3) *Deepsy-rand selection*. This is Deepsy with the scope-based prioritization replaced with a random-select prioritization.

The two variants of Deepsy help us examine the benefits brought by the two focusing and prioritization techniques. We in addition report the quality of the result by Deepsy in a step by step manner to show the benefits from each individual technique it uses.

For comparison, we implemented the method in SmartSynth and applied it on all single sentence test cases. Our implementation first tagged each of ASTMatcher APIs with its name and the keywords in its description. For a given input sentence, the original SmartSynth uses an iterative



(a) Overall Accuracy of Deepsy and SmartSynth on single sentence test cases.

(b) Accuracy comparison on single/multi sentence test cases

Figure 5. Accuracy comparison.

search to find a good way to separate it into chunks. To prevent wrong chunking results from hurting the later steps, we always use correct chunking results. In the mapping step, for each chunk, SmartSynth selects the APIs whose labels are related with the chunk. It then enters the dataflow relations discovery step, in which, it determines the order the selected APIs mainly based on data types and the distance between chunks in the input sentence. It uses some rules to help detect dataflow relations.

All our experiments were performed on a machine that equipped with an 4-core 2.6GHz Intel® Core(TM) i7-6700 (8GB RAM).

5.2 Overall Results

The average execution time for the synthesis process in Deepsy takes only 0.19s. Our discussion hence concentrates on the synthesis accuracy.

The overall accuracy of Deepsy is shown in Figure 5(a). Deepsy correctly generates 70 matcher expressions out of the 100 single sentence cases, achieving an accuracy of 70.0%; it correctly produces 29 matcher expressions out of the 40 multi-sentence cases, achieving an accuracy of 72.5%.

Deepsy- w/o type-driven hits 62 matcher expressions, which is 62.0% accuracy. Deepsy-rand selection achieves only an accuracy of 30.0%. Inside Deepsy, the type-driven step is used to identify the promising matchers whose type is compatible with its parent and children, while the scope-based prioritization is a decision making step that determines the promising matcher.

For the inner accuracy, Deepsy, Deepsy-w/o type-driven, and Deepsy-rand selection achieve an accuracy of 85.1%, 80.7%, 62.1% respectively. Similar to the overall accuracy, a correct matcher can be possibly selected without the type-driven step but is not likely to be identified with a random choice. Deepsy achieves the highest inner accuracy, implying the necessity of including both the steps.

The overall expression-level accuracy from SmartSynth is 52%, significantly lower than the 68% accuracy from Deepsy. The large errors come from its lack of considerations of the dependency relations and treatment to the tangled concerns between mapping and ordering. SmartSynth heavily relies on data types and distances rather than dependency relations among chunks, and deals with the mapping and ordering in a loosely coupled manner.

Consider Example 3 in Table 1. Suppose that the mapping step selects the correct matchers, and correctly produces the first part of the expression, `functionDecl(hasDescendant(declRefExpre(to())))`. The matcher `to()` expects type `Decl` as its argument's type. The rest of the matcher APIs, `hasGlobalStorage()` (denoted as A), `varDecl()` (as B), `hasName()` (as C), all return `Decl` or a descendant class of `Decl`. So (A, B, C), (A, B(C)), (B(A), C), are all possible orders that give correct data types. The issue is addressed in Deepsy for its consideration of the semantic connections among the different parts of the NL query. The co-evolvement helps deliver the effects in both mapping and ordering in a synergistic way.

Figure 5(b) shows the accuracy of Deepsy on multi-sentence descriptions, and on easy and hard expressions. Deepsy achieves 75.0% accuracy on 40 multi-sentence test cases. For easy expressions, Deepsy achieves an accuracy of 78.3% for single sentences and 82.4% for multi-sentences. For hard expressions, the single sentence cases and multi-sentence cases have an accuracy of 60.0% and 65.2%, respectively.

If the dependency parsing could be improved to avoid NL parsing errors (here we manually correct the dependency tree), the accuracy of the single sentence cases jumps to 80% and multi-sentence cases to 77.5%. Correcting parsing errors benefits more for the single-sentence descriptions than the multi-sentence ones. It is because dependency parsing is prone to make more mistakes for a complex single sentence.

5.3 Error Analysis

In this section, we analyze the cases where Deepsy fails to synthesize correct matching expressions. Three reasons are responsible for the errors: wrong decisions in search-based gap filling and scope-based prioritization, and limitations in dependency parsing. We elaborate on each reason using one test case as an example. The example test cases are listed in Table 3.

Wrong decisions in search-based gap filling. The first test case in Table 3 falls into this case. Two neighbored nodes in the dependency tree of the English description are mapped to the matcher `cxxMemberCallExpr` and the matcher `cxxMethodDecl` respectively. Due to the *alternative restriction*, search-based gap filling searchs within the *matcher graph* for an appropriate matcher or a sequence of matchers to fill in the gap. The searched results contain both the matcher `callee` and the matcher `thisPointerType`. Since the return type of `thisPointerType` exactly matches the input type of `cxxMemberCallExpr` while `callee` returns an inheritance type, `thisPointerType` has higher priority and is selected as the gap filling matcher.

Wrong decisions in scope-based prioritization. Scope-based prioritization prefers a matcher with the smallest extra scope. For the second test case in Table 3, `hasConditionVariableStatement` instead of `hasCondition` is chosen in this step

Table 3. Errors in synthesizing matcher expressions.

No.		English description	Matcher expression	Error expression	Reason
1		Find all c++ call expression of the c++ method named string_1.	cxxMemberCallExpr(**callee**(cxxMethodDecl(hasName(string_1)	cxxMemberCallExpr(**thisPointerType**(cxxMethodDecl(hasName(string_1)	search-based gap filling
2		Find if statement whose condition is smaller than 10.	ifStmt(**hasCondition**(binaryOperator(hasOperatorName("<"), hasRHS(integerLiteral(equals(10))))))	ifStmt(**hasConditionVariableStatement**(binaryOperator(hasOperatorName("<"), hasRHS(integerLiteral(equals(10))))))	scope-based prioritization
3	a	Find call expression which calls the function named string_1.	callExpr(callee(**functionDecl**(hasName(string_1))))	callExpr(callee(**typedefNameDecl**(**functionType**(string_1))))	dependency parsing
	b	Find call expression which calls the function whose name is string_1		(The synthesized expression is correct)	–

Table 4. Benefits of steps: the numbers of correct matcher APIs after each step.

ID	Ground truth	Word trans.	Longest-match phrase trans.	Order-based type-driven focusing	Scope-based prioritization	Rule-directed gap filling	Reorder-based gap filling	Search-based gap filling
1	8	1	1	2	5	8	8	8
2	9	2	2	3	6	6	8	9
3	7	1	2	2	4	7	7	7
4	3	0	0	0	2	2	3	3
5	5	1	1	2	4	4	5	5
6	9	2	2	3	3	9	9	9
7	5	1	1	1	2	4	5	5
8	5	1	1	2	4	5	5	5
9	3	0	0	0	2	2	2	3
10	5	0	0	0	4	4	4	5

1. Search for all the functions that use a particular global variable named "PI".
2. Find for loops whose init portions declare a single variable which is initialized to the integer literal 0.
3. Search for all call expressions that use a function named "pi".
4. Return field declarations whose types are a typedef.
5. Return all cxx call expressions which call the cxx method named "cxxfunc".
6. Get for statements whose conditions are less than 10.
7. Return call expressions which call the function whose name is "func".
8. Find all call expressions of a function whose name is "func".
9. Search for all binary operators whose operator names are "=".
10. Find call expressions whose argument is a C-style cast expression with destination type being a real floating point.

because the former one has a longer description and thus a smaller extra scope. More knowledge of matchers may help this situation.

Limitations in dependency parsing. For the third example in Table 3, we list two sentences that have the same matcher expression. Although the two sentences have the same meaning, only the query in 3.b yields the correct matcher expression. The dependency parsing result of the query in 3.a treats "named" wrong and puts it as the parent node of "function", resulting in a wrong matcher order and thus an incorrect matching expression. Dealing with the NLP errors is deferred to our future work.

5.4 Benefits from Each Individual Step

We report the benefits of each step in Deepsy by showing the *match score* after each step for ten of the test cases. The *match score* is the number of matchers which have been decided correctly. The result is shown in Table 4. The second column "Ground truth" shows the results in the ground truth. The following columns are the steps of Deepsy, and we report the scores after each step. The NL queries are listed below the table.

According to Table 4, as Deepsy processes a dependency tree step by step, the match score increases gradually. After the word translation step, the match score is usually one or two because the nodes with special terms in the dependency tree are successfully matched to some basic matchers (e.g., `forStmt` for `for loop`). For each of the remaining nodes, a list of candidate matchers (i.e. *matcher list*) are created and will be refined in the following steps. In most of the cases, the match score increases significantly after the scope-based prioritization step. It is because this step identifies the most promising matcher from the matcher list of each node and some of the decisions are correct. In some cases, rule-directed gap filling also contributes to a large increase in the match scores because special verbs (e.g. "use") are mapped to a sequence of matchers in the step.

In most of the cases, the top-down order of the nodes in a dependency tree already give a correct order for the corresponding matchers. The discrepancies appear in several cases. For instance, test case 2 in Table 4 has a wrong order initially due to the use of relative clauses. The tree reordering by Deepsy fixes all of the ordering issues in the test cases successfully.

6 Discussions

Although the synthesis results are not always correct, they can still be useful. For instance, an IDE integrated with Deepsy can suggest several top choices of expressions for a programmer to pick. Even if minor fixes may be needed occasionally, it is still much easier than manually coming up with the whole expression from scratch based on the 500 APIs and a full-fledged type hierarchy.

Our experiments show that using simple tags to help with coreference resolution offers a good tradeoff between the synthesis accuracy and usability. The inconveniences it adds to users is insignificant according to the feedback from the subjects who created the test cases. As automatic coreference resolution techniques in NLP improve, the tagging can be eventually replaced with automatic analysis.

7 Related Work

Program Synthesis is the task of automatically synthesizing a program that satisfies the user intent expressed in the form of some specification [15]. A specification can be first order logic expressions [13], a set of input/output examples [11, 12], natural language [7, 14, 23, 39, 44], partial programs [9, 41] or any other form that is easier to write than the expected program. As Deepsy is a natural language-based synthesizer, we concentrate on prior work on program synthesis from natural language (NL).

Recent efforts have been spent on machine learning-based approaches [1, 2, 7, 8, 10, 16, 17, 20, 25–27, 35–37, 39, 42, 45]. Rule-based approaches have been developed to synthesize programs for domain-specific tasks such as smartphone automation [23], SQL queries [24, 44], and SpreadSheet data analysis [14]. The most complex domain among them is the smartphone automation [23]. The complexity is still much more limited compared to ASTMatcher. The previous section has provided a quantitative comparison with the previous rule-based method.

Some works try to identify some statistical patterns of API usage. Examples include code search tools [29, 32], API usage pattern mining [38, 43], API sequence generation [10]. These studies rely on statistical machine learning techniques. They hence require a large set of examples, making them hard to apply to program analysis.

8 Conclusion

This paper presents Deepsy, one of the first NL-based synthesizers for ASTMatcher to help general programmers conduct program analysis. Deepsy features *dependency tree-based co-evolvement*, a novel design that leverages dependency relations from *NL dependency analysis* [19] and synergizes NLP techniques and domain knowledge. It is worth mentioning that the demonstrated potential of NL dependency analysis for program synthesis has led the authors to the development of a more general approach, named *Human-Learning Inspired Program Synthesis*, described in another paper [34], which combines the NL depedency analysis and grammar-guided synthesis together, producing a synthesizer with an even greater applicability and accuracy.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Initial work was supported by LLNL-LDRD 18-ERD-006 (LLNL-CONF-794949). Revisions were funded through subsequent support from the U.S. DOE Advanced Scientific Computing Research.

References

[1] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[2] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 602–612.

[3] The Clang-Team. [n.d.]. ASTMatcher Reference. clang.llvm.org/docs/LibASTMatchersReference.html.

[4] The Clang-Team. [n.d.]. Clang. clang.llvm.org..

[5] The Clang-Team. [n.d.]. Tutorial for building tools using LibTooling and LibASTMatchers. clang.llvm.org/docs/LibASTMatchersTutorial.html.

[6] Keith Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

[7] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 345–356.

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 990–998.

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

[10] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[12] Sumit Gulwani. 2012. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*. IEEE, 8–14.

[13] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

[14] Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 803–814.

[15] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[16] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Acm Sigplan Notices*, Vol. 50. ACM, 416–432.

[17] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.

[18] Tim Kelley. 2017. FINDING GLOBAL VARIABLES WITH CLANG AST MATCHERS. https://variousburglarious.com/2017/01/18/finding-global-variables-with-clang-ast-matchers/. Accessed: 2018-11-14.

[19] Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. Dependency parsing. *Synthesis Lectures on Human Language Technologies* 1, 1 (2009), 1–127.

[20] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. 2004. Guiding a reinforcement learner with natural language

advice: Initial results in RoboCup soccer. In *The AAAI-2004 workshop on supervisory control of learning and adaptive systems*. San Jose, CA.

[21] Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 826–836.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[23] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 193–206.

[24] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment* 8, 1 (2014), 73–84.

[25] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D Ernst. 2017. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01* (2017).

[26] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. *arXiv preprint arXiv:1802.08979* (2018).

[27] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[28] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv preprint arXiv:1608.03000* (2016).

[29] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 260–270.

[30] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60. http://www.aclweb.org/anthology/P/P14/P14-5010

[31] Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating Programming by Example and Natural Language Programming.. In *AAAI*.

[32] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.

[33] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[34] Zifan Nan, Hui Guan, and Xipeng Shen. 2020. HISyn: human learning-inspired natural language programming. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE),*.

[35] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[36] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. *arXiv preprint arXiv:1802.04335* (2018).

[37] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 878–888.

[38] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean-Code Search and Idiomatic Snippet Synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 357–367.

[39] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples.. In *IJCAI*. 792–800.

[40] Marta Recasens, Marie-Catherine de Marneffe, and Christopher Potts. 2013. The life and death of discourse entities: Identifying singleton mentions. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 627–633.

[41] Armando Solar-Lezama and Rastislav Bodik. 2008. *Program synthesis by sketching*. Citeseer.

[42] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. 2017. Building natural language interfaces to web apis. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 177–186.

[43] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 54–57.

[44] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 63.

[45] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[46] Zexuan Zhong, Jiaqi Guo, Wei Yang, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. Generating Regular Expressions from Natural Language Specifications: Are We There Yet?. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.