User-Customizable Transpilation of Scripting Languages

BO WANG, National University of Singapore, Singapore
AASHISH KOLLURI, National University of Singapore, Singapore
IVICA NIKOLIĆ, National University of Singapore, Singapore
TEODORA BALUTA, National University of Singapore, Singapore
PRATEEK SAXENA, National University of Singapore, Singapore

A transpiler converts code from one programming language to another. Many practical uses of transpilers require the user to be able to guide or *customize* the program produced from a given input program. This customizability is important for satisfying many application-specific goals for the produced code such as ensuring performance, readability, ease of exposition or maintainability, compatibility with external environment or analysis tools, and so on. Conventional transpilers are deterministic rule-driven systems often written without offering customizability per user and per program. Recent advances in transpilers based on neural networks offer some customizability to users, e.g. through interactive prompts, but they are still difficult to precisely control the production of a desired output. Both conventional and neural transpilation also suffer from the "last mile" problem: they produce correct code on *average*, i.e., on most parts of a given program, but not necessarily for all parts of it. We propose a new transpilation approach that offers fine-grained customizability and reusability of transpilation rules created by others, without burdening the user to understand the global semantics of the given source program. Our approach is *mostly automatic* and *incremental*, i.e., constructs translation rules needed to transpile the given program as per the user's guidance piece-by-piece. Users can rely on existing transpilation rules to translate most of the program correctly while focusing their effort locally, only on parts that are incorrect or need customization. This improves the correctness of the end result. We implement the transpiler as a tool called DuoGlot, which translates Python to Javascript programs, and evaluate it on the popular GeeksforGeeks benchmarks. DuoGlot achieves 90% translation accuracy and so it outperforms all existing translators (both handcrafted and neural-based), while it produces readable code. We evaluate DuoGlot on two additional benchmarks, containing more challenging and longer programs, and similarly observe improved accuracy compared to the other transpilers.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**; **General programming languages**; **Source code generation**.

Additional Key Words and Phrases: Program Translation, Program Synthesis

1 INTRODUCTION

Transpilers are designed to automatically translate source code written in one programming language to its equivalent code in another. They have many applications including migrating

Authors' addresses: Bo Wang, National University of Singapore, Singapore, bo_wang@u.nus.edu; Aashish Kolluri, National University of Singapore, Singapore, aashishk@u.nus.edu; Ivica Nikolić, National University of Singapore, Singapore, inikolic@nus.edu.sg; Teodora Baluta, National University of Singapore, Singapore, teodora.baluta@u.nus.edu; Prateek Saxena, National University of Singapore, Singapore, prateeks@comp.nus.edu.sg.

codebases to newer languages and compiling code written in a newly-designed language to a well-supported language. To illustrate, the Common Bank of Australia is reported to have spent 750 million dollars over 5 years to replace its codebase written in COBOL [Irrera 2017]. This problem is exacerbated when languages are frequently updated to newer versions that selectively break backward compatibility, such as in the case of Python v2 to v3 [van Rossum 2009] and Tensorflow v1 to v2 [Google 2021]. In all of these situations, taking assistance from transpilers could improve productivity and reduce costs. Transpilers have other applications, such as in education, where students can learn new programming languages by writing example code in languages they may be familiar with [Pasternak et al. 2017].

Several ways of designing transpilers have been proposed. The conventional way is for human experts to manually write transpilers as a collection of translation rules [Cordy 2006; Terekhov and Verhoef 2000]. On the other hand, neural machine translation (NMT) is used to learn translation patterns or models from a large corpus of code from source and target languages—a promising approach to reduce human effort in writing transpilers [Mariano et al. 2022; Roziere et al. 2020, 2021; Xinyun et al. 2018]. All existing approaches suffer from at least one of two major drawbacks. The first is related to the *monolithic costs* required for development of transpilers. It takes considerable time and expertise to write translation rules that are sufficiently complete to transpile a significant fraction of programs in the source language for conventional transpilers. The upfront costs of designing and integrating such monolithic transpilers into the development pipelines can outweigh the benefits. Often users are interested in translating only certain programs and are not willing to invest the effort needed to write general transpilers. NMT transpilers also incur large monolithic costs. They require hundreds of thousands lines of code in the source and target languages as examples, access to large general language models such as GPT-3/Codex [Brown et al. 2020; Chen et al. 2021], and are computationally expensive to train. Without sufficient training data and computational costs incurred, neural machine translators can have low transpilation accuracy. The second problem is that of *inflexibility*. Existing transpilers assume that all users prefer the same translations of the same source program. There are multiple ways to translate the same code and developers often have many soft constraints to satisfy (library dependencies, preferred paradigms, readability, performance, etc) when choosing a specific way of translation. Often, there is a gap between the user's intent and the translation produced by existing transpilers because they have been designed to work in generic contexts rather than the one the user wants.

Our approach. In this paper, we propose a new approach to designing transpilers that tackle the above drawbacks. We advocate that transpilers should treat *user-customizability* as a first-class design objective. In our proposed approach, users can selectively write or reuse translation rules that are sufficient to transpile their specific program of interest correctly. The power of the approach comes from two sub-features. The rules are created *incrementally* and *mostly automatically*. By incremental, we mean that the user can select the starting translation rule set, potentially written by others, and incrementally add to it until the given program is correctly translated. By mostly automatic, we mean that the transpiler tries to minimize the user effort by learning the transpilation rules from code snippets given as "hints" by the user. The process should be data efficient, i.e., learning a new rule should require very few pairs of source-target code snippets as examples.

We build a prototype system called DᴜᴏGʟᴏᴛ to transpile programs between two popular scripting languages: Python to JavaScript. We choose Python and JavaScript for illustration of the broader concept and its practical feasibility, though we expect supporting other untyped scripting languages to be similarly feasible. DᴜᴏGʟᴏᴛ takes as input a Python program and produces as output a JavaScript program. The goal is to produce a correctly translated program, i.e., to generate target code with the same input-output behavior as the source program under given unit tests.

Several design decisions have made this approach practical. First, DuoGlot translates the abstract-syntax tree (AST) representation of one program to its AST representation in another language, similar to some of the existing transpilers. Converting code to AST using standard compilers and vice versa using pretty-printers is practical as tools for this are commonly available for many popular languages. Second, our transpiler is a "stateless" transducer. By avoiding explicit state transition rules, DuoGlot reduces overfitting and makes rules learned from code examples more usable, i.e., the rules created with DuoGlot can be applied to a similar statement elsewhere in the same program, or imported when translating a similar benchmark or a library. Lastly, the transducer is designed to permit non-determinism, i.e., it offers several choices at a given point during translation. This is key to making the transpilation adaptive to the program being translated, as it is forced to pick between translation rules dynamically. Lastly, all rules created are open to inspection by the user, making the approach more interpretable compared to NMT.

We demonstrate DuoGlot's performance by translating Python code to JavaScript on one existing and two new benchmarks. First, we evaluate on the popular benchmarks maintained by the TransCoder project [MetaResearch 2022; Roziere et al. 2020, 2021]. The benchmarks have 699 Python programs (1-40 lines) with unit tests. We compare DuoGlot to 3 manually written rule-based transpilers available publicly and to Codex, a modern neural-based transpiler that supports Python to JavaScript. DuoGlot translates 90% of the benchmarks with just 142 rules learned from pairs of code snippets (one or two for each rule) provided by the user. All other evaluated tools correctly translate at least 14% fewer programs. Furthermore, as an illustration of how the rules can be customized for soft constraints, we measure the readability of the transpiled code using the code bloat metric (an increase in the number of characters). Less code bloat is better [Alkhatib 1992]. DuoGlot produces code that has good readability, comparable to that of Codex, and 1.6× to 8× better than the hand-crafted transpilers we study. Qualitatively, we analyze the manual effort involved in working with DuoGlot, providing examples and a rough categorization. We argue that most of the rules are rather simple and so are the required code snippets.

We further assess DuoGlot's strengths and limitations with two cases studies, each focused on running the tool on new benchmarks. The first case study is about translating more authentic and complex programs, with potentially insufficient unit tests. We collect solutions to LeetCode programming challenges[1] written in Python (each has a few unit tests in their problem description) and translate them to JavaScript. We end up with 1067 such Python programs, with an average length of 15 lines. Due to increased program complexity, no other transpiler is able to achieve accuracy higher than 50%. DuoGlot achieves 75.4% accuracy with 110 additional on top of the previous 142 rules, and requires 1.75 seconds per program. In the second case study we test the performance on longer code. We collect benchmarks from the book "Cracking the coding interview"[McDowell 2015] – in total 25 Python programs, each with 54 to 160 lines of code. Once again, DuoGlot has the best accuracy among the transpilers, while increasing its average translation time to 6.8 seconds per program. It achieves 56% accuracy with 197 rules (142 previous and 55 additional), which is higher than the accuracy of the other transpilers (12% by Codex and 44% by Transcrypt). The two case studies reveal some of the limitations of DuoGlot as well. First, despite being more correct than other transpilers, DuoGlot falls short of the ideal 100% accuracy. Second, when the unit tests do not provide full coverage, DuoGlot may have unpredictable accuracy on the uncovered code. Third, DuoGlot is scalable, but with some degradation of accuracy and efficiency. Despite being mostly generic, the above limitations indicate that DuoGlot may be improved, and we leave this as an open problem.

---

[1]https://leetcode.com/problemset/all/
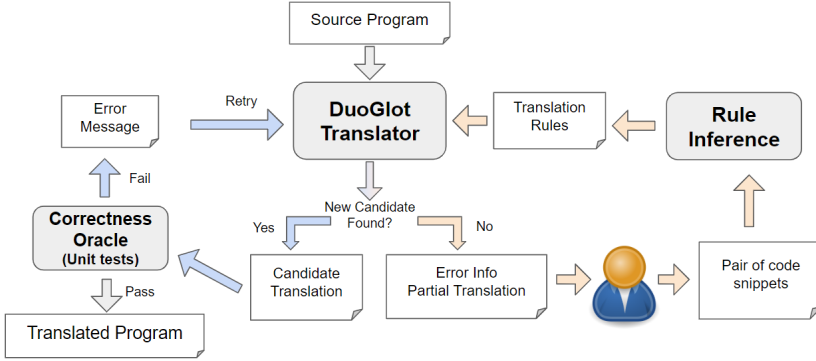
2   PROBLEM SETUP AND DUOGLOT'S OVERVIEW

Fig. 1. Workflow of Incremental Translation. The loop in blue arrows on the left is the machine loop (frequent case) and the loop in yellow arrows on the right is the user loop (less frequent case, incremental).

A user utilizes the transpiler to translate a program written in source language to a program written in target language. We design a new transpiler called DUOGLOT that translates programs from Python (source) to JavaScript (target). We assume that unit tests for both the source and the target programs are available to validate the correctness of the translation. Test-based validation is used in practice, for instance in programming contests like LeetCode. If there are unit tests only for the source code, it is relatively easy to translate them to the target language as demonstrated previously [Roziere et al. 2020]. When transpilers cannot translate a particular source program they usually abort or output a wrong target program. An interesting alternative, however, would be to infer a new rule that handles the problematic part of the program by prompting users for a fix. This is the approach we take in DUOGLOT by asking users for code snippets in both languages as hints for fixing the translation. The paradigm of interacting with users to achieve the required functionality is used in program synthesis [Gulwani et al. 2017] and verification [Bader et al. 2018]. For instance, popular programming-by-example synthesis tools, such as Flashfill [Polozov and Gulwani 2015], receive input-output examples from the user to make the search for a correct program tractable.

The workflow for our transpiler DUOGLOT is shown in Figure 1. Our transpiler puts the user in control of the transpilation process. DUOGLOT is an incremental transpiler. It tries to learn a new rule each time it cannot translate some part of the code and thus slowly grows its translation *rule set*. To learn each such new rule, it interacts with the user and prompts for pairs of code snippets in source and target languages that help to infer the rule. The rule focuses the user's attention locally on one part of the code, so the user does not need to understand the global semantics of the source program. These user-provided snippets aid rule inference, while letting the user customize the translation directly with the provided snippets. Note the user is not expected to be an expert—they do not need to learn a third language or know the internals of DUOGLOT.

At the beginning, DUOGLOT starts the translation of the source program with a set of general translation rules to bootstrap the translation procedure. These rules capture the most prominent constructs of the languages such as identifiers, literals, basic assignments, if statements, and so on. All rules learned during previous translations are also added to the rule set and the user has complete control over which rules it selects. DUOGLOT then searches for a candidate source program translation using the provided rules. If the search concludes without finding a correct translation,

DuoGlot locates the place of code that failed to translate and prompts the user for help to handle the failed statement or expression. The user then provides short code snippets that demonstrate the correct translation and the preferred translation style for such a statement or expression. At this point, DuoGlot executes its rule inference procedure to learn a rule, by looking at syntactic patterns of the provided snippets. It adds this newly learned rule to its rule set so that it can directly apply the learned rule when the same syntactic pattern appears elsewhere in the future. Using the updated rule set, DuoGlot retries the search for more candidate translations, until the translation task is finished.

DuoGlot is *mostly* automatic. When the user provides code snippets as hints for a fix, DuoGlot does much of the heavy lifting in learning the translation rule internally from the given hints. Further, during the transpilation process as new candidate translations get created, DuoGlot incrementally checks their correctness by running unit tests. If the translation turns out to be incorrect then an error message with the error location (when available) is automatically fed back. Hence the correctness oracle not only validates but also serves as a test-repair loop. This helps to generate correct translations efficiently by pruning away bad translations eagerly. To our knowledge, existing transpiler designs lack such loops that enforce correct translations.

Motivating Example. To illustrate the workflow of our transpiler, consider the Python program given in Figure 2. The program outputs the average frequency of a set of words as they appear in a set of lines. Assume we start DuoGlot with a small rule set insufficient to translate the whole program. For instance, there is no rule to translate Python's list comprehension used in line 2. Our tool detects this, notifies the user there is a problem at line 2, and asks for code snippets. The user can reply with simple snippets like the ones in Figure 3. Note that list comprehension can be translated in different ways in JavaScript, and the user may instead prefer and provide the alternative style from Figure 4. (This is the customization feature of the transpiler—the user determines the preferred style through the provided code.) Subsequently, DuoGlot's rule learning procedure automatically infers a translation rule from these code snippets and adds it to the rule set. The user, if desired, may directly inspect the rule.

```
1   def f_gold(lines, word_list):
2     trwords = [[y.upper() for y in x.split(" ")] for x in lines]
3     count_dict = {}
4     for line_split in trwords:
5       for word in line_split:
6         if word in word_list:
7           if word not in count_dict:
8             count_dict[word] = 0
9             word_list.append(word)
10          count_dict[word] += 1
11    for i in range(len(word_list)):
12      word = word_list[i]
13      msg = "found" if word in count_dict else "not found"
14      print(i, word, msg)
15    count_sum = 0
16    for word in count_dict: count_sum += count_dict[word]
17    return count_sum // len(word_list)
```

Fig. 2. Python program to translate.

```
# Python                    // JavaScript
[x for x in []]             Array.from([]).map(x => x);
[y+y for y in a]            Array.from(a).map(y => y+y);
```

Fig. 3. Pairs of source-target code snippets for translation of list comprehension.

```
# Python                    // JavaScript
[x for x in []]             [].reduce(function (arr, x)
[y+y for y in a]            {arr.push(x); return arr;}, []);
                            a.reduce(function (arr, y)
                            {arr.push(y+y); return arr;}, []);
```

Fig. 4. Pairs of source-target code snippets for alternative translation of list comprehension.

Then DuoGlot retries the translation and finds the candidate translation for the list comprehension. After applying this rule, it generates a candidate translation 1 (the first two lines are shown in Figure 5) where it assigns the list comprehension to the `trwords` variable, but without declaring the variable. Note, there are two competing rules in the bootstrapped rule set for a Python expression with `=` operator such as `a=b`. The first rule would keep the expression as is in the JavaScript translation. The second

rule would translate it to a declaration in JavaScript, `let a=b`. The two rules arise since Python does not have a separate representation for assignments and declarations whereas JavaScript does. However, the application of the first rule leads to the candidate translation 1 which fails the tests and the correctness oracle reports that `trwords` at line 2 is not defined. DuoGlot will then look up its rule set and will figure out that it has another rule to apply at line 2 (which translates Python assignment to JavaScript declaration). Thus, it will produce candidate translation 2.

This iterative test-driven procedure will continue for 13 rounds[2] in our example until a candidate translation that passes all the unit tests is found. Unlike in candidate 1, in candidate 13 all the syntax patterns are translated correctly including `for x in y` to `for (let x of y)`, `for (let x in y)` or `for(let i = 0; i < ..., i++)`. The final JavaScript program is given in Figure 6. The translation is provided in the user interface (UI) where the user can hover on a piece of translated code to find out the part of source it comes from and the used translation rule.

Fig. 5. Candidate translation 1 has an error on line 2. DuoGlot will automatically fix such an error with the test-repair loop.

Comparison to Codex and Transcrypt. To highlight the benefits of our approach, we also compare the translation of Figure 2 program to two state-of-the-art transpilers, Codex [Chen et al. 2021] and Transcrypt [3] (see Figure 7 for their translated JavaScript program). Codex is a machine-learning-based language model for code-related tasks used by Github Copilot, while Transcrypt is a popular hand-crafted rule-based transpiler reportedly used in production [Koeninger 2020]. We observe that Codex frequently confuses semantically similar expressions, operators and variables, or misses out part of the source code. Let us consider the 5 errors in Codex translation highlighted in red. At line 2, Codex ignores the `toupper()` function call inside the nested list comprehension. At line 8, it translates the expression `word in word_list` as it is but `word_list` is not a dictionary and the correct translation is `word_list.indexOf(word)>= 0`. At line 19, the translation is not syntactically valid because

Fig. 6. Translated JavaScript program (Candidate 13). When hovering over the translated code in DuoGlot UI, the corresponding piece of the source is highlighted.

it should be a ternary expression rather than an expression followed by a statement. At line 20, it translates `print` as it is, while at line 26, it confuses `//` with `/`. There is no easy way to predict whether and why Codex will make a mistake because there are *no* interpretable rules or explanations for its translation. Overall Codex's output is readable, however, it makes frequent mistakes

---

[2]It happens to be 13 rounds in this particular example.
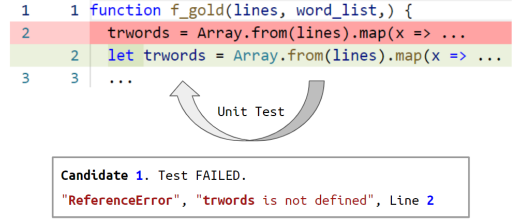[3]Python 3.7 to JavaScript compiler. https://github.com/qquick/Transcrypt
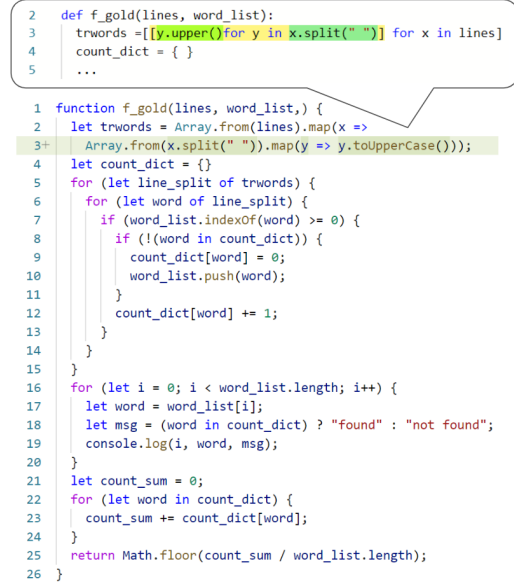
with semantically similar operators. A human working with Codex may be able to fix these issues, nonetheless, the Codex approach requires a huge training dataset and computational resources to arrive at this result.

```
 1  function f_gold(lines, word_list) {
 2-   var trwords = lines.map(function(x) { return x.split(" "); });
 3    var count_dict = {};
 4    for (var i = 0; i < trwords.length; i++) {
 5      var line_split = trwords[i];
 6      for (var j = 0; j < line_split.length; j++) {
 7        var word = line_split[j];
 8-       if (word in word_list) {
 9          if (!(word in count_dict)) {
10            word_list.push(word);
11            count_dict[word] = 0;
12          }
13          count_dict[word] += 1;
14        }
15      }
16    }
17    for (var i = 0; i < word_list.length; i++) {
18      var word = word_list[i];
19-     var msg = "found" if (word in count_dict) else "not found";
20-     print(i, word, msg);
21    }
22    var count_sum = 0;
23    for (var word in count_dict) {
24      count_sum += count_dict[word];
25    }
26-   return count_sum / word_list.length;
27  }
```

Codex

```
    object, ord, pow, print, property, py_TypeError, py_iter, py_metat
    py_reversed, py_typeof, range, repr, round, set, setattr, sorted,
    zip} from './org.transcrypt.__runtime__.js';
 3  var __name__ = '__main__';
 4  export var f_gold = function (lines, word_list) {
 5    var trwords = (function () {
 6      var __accu0__ = [];
 7      for (var x of lines) {
 8        __accu0__.append ((function () {
 9          var __accu1__ = [];
10          for (var y of x.py_split (' ')) {
11            __accu1__.append (y.upper ());
12          }
13          return __accu1__;
14        }) ());
15      }
16      return __accu0__;
17    }) ();
18    var count_dict = dict ({});
19    for (var line_split of trwords) {
20      for (var word of line_split) {
21        if (__in__ (word, word_list)) {
22          if (!__in__ (word, count_dict)) {
23            count_dict [word] = 0;
24            word_list.append (word);
25          }
26          count_dict [word]++;
27        }
28      }
29    }
30    for (var i = 0; i < len (word_list); i++) {
31      var word = word_list [i];
32      var msg = (__in__ (word, count_dict) ? 'found' : 'not found');
33      print (i, word, msg);
34    }
35    var count_sum = 0;
36-   for (var word of count_dict) {
37      count_sum += count_dict [word];
38    }
39    return Math.floor (count_sum / len (word_list));
40  };
```

Transcript

Fig. 7. Codex translation of the code in Figure 2 has multiple errors, mostly due to confusing semantically similar operators and expressions. The translation by Transcrypt emulates many Python's built-in functions, but still has one error due to the type difference between `Object` and `Array` and it is much more verbose.

Unlike Codex or DuoGlot, Transcrypt imports a custom runtime library for emulating many common Python functions and operators, as shown at the top-right of Figure 7. However, it still has an error at line 36 where it fails to infer that `count_dict` is an object rather than an array, so `for (let x in y)` should be used instead of `for (let x of y)`. This highlights a key issue with designing a transpiler that aims to solve the general transpilation problem all at once: it is hard to accurately know the variable's type (or possible values), which is critical when translating to a different language in a different runtime. On the contrary, explicit type inference is not even required in the increment transpilation paradigm we present. The test-driven feedback loop of DuoGlot guides it towards the correct translation.

In summary, our transpilation paradigm provides the following benefits:

(1) **Customizability**. DuoGlot adopts user-provided translation rules. It can be customized for different coding styles, APIs, language versions, etc. Moreover, DuoGlot grants fine-grained customization while learning rules from code snippets, allowing users to specify the translation rules for all language constructs. It allows them to modify the translation rules and tinker with different rule sets by splitting or merging them. Moreover, the customization is optional, and users may choose to reuse the rules supplied by others, partially or in full. None of the existing transpilers provide such customizability, in part due to their non-incremental

nature. The manual monolithic ones demand expert knowledge while the ML-based ones might need re-training.

(2) **Low manual effort.** DuoGlot *automatically* learns the rules from small code snippets provided by the user. This alleviates the effort of manually crafting them. Further, DuoGlot engages the user to provide code snippets only when learning a new rule, not already present in the knowledge base. Its incremental approach requires learning only rules needed to perform the translation task at hand. Therefore, by design, it requires a much smaller number of code snippets than, for instance, training a general ML model. The manual effort required post-translation is low as well since DuoGlot is optimized to generate correct and readable code along with an explanation. In our experiments, DuoGlot obtains correct translations for 90% of the main benchmarks, in total 5297 lines of code, with just 360 lines of user-provided code snippets in total.

3 DUOGLOT: CHALLENGES AND APPROACH

The core component of DuoGlot is a tree transducer ([Comon et al. 2008], Chapter 6) which reads an input abstract syntax tree (AST) of the source program and translates it to the target AST. The target AST has enough information to be converted back to a syntactically-valid target program using a pretty printer. This transducer can be seen as matching each sub-tree in the input AST and mapping it to a sub-tree in the target AST. The translation, therefore, proceeds as a sequence of steps where in each step the transducer uses a *translation rule* to output the translated AST. The transduction proceeds recursively top-down, i.e., it starts from the root of the input AST, applies the translation rule to the tree, and then moves down to translate some of its subtrees. Such transducers are called extended left-hand side (extended LHS) tree transducers [Graehl et al. 2008]. Translation rules are written in a domain-specific language (DSL), which we will detail later in Section 4.1. DuoGlot addresses three key challenges to make the transpilation practical.

First, DuoGlot needs to learn rules that are general and reusable, therefore avoiding overfitting to the code samples provided by the user. Suppose the code samples `a = 1; a = 2` (Python) and `let a = 1; a = 2` (JavaScript) are provided to learn a rule that handles assignment. DuoGlot should not learn a rule that says "assign value to a variable only after it is declared *and assigned the value 1*". This is one form of overfitting and many such examples exist. To address this challenge, DuoGlot's extended LHS tree transducer uses only a *single state*[4]. Such a transducer does not preserve any context or history across the sequence of translation steps, i.e., it applies the same translation to a sub-tree of the input AST regardless of how other sub-trees in the input AST have been translated. So, DuoGlot is forced to use the same translation for, say `a=2`, irrespective of how `a=1` was translated earlier. Therefore, it cannot have a rule that says "translate Python assignment sub-expressions to JavaScript assignment *only if the variable has been declared with value 1* or something else otherwise". This fundamentally addresses part of the overfitting problem that may occur due to learning a rule that overfits to the given code snippets.

The second challenge is that of handling ambiguity during transduction. There are two sources of ambiguity: 1) the transducer does not have enough history or context to pick between rules; and 2) there are inherently more than one functionally correct mappings to translate a given code snippet from Python to JavaScript. To address the first source of ambiguity, the transducer needs to be *non-deterministic*, i.e., for the same input AST sub-tree there can be more than one rule option for the transducer to eventually apply. The decision of which rule to use, however, is not going to be manually specified in the state machine in the transducer since the single-state property of the

---

[4]For brevity, from now on we omit "extended LHS" when describing our tree transducer.

Fig. 8. (Left) Syntactically valid and invalid translations. DuoGlot checks the validity of every transduction step and rejects the invalid translations on the fly. (Middle) Syntactically valid, but not necessarily semantically valid translations. The second translation is deemed correct when the dynamic type of `y` is a set. DuoGlot checks for semantic validity by running the unit tests every time a syntactically-valid candidate translation is generated. (Right) Syntactically and semantically valid translations. In such a case, DuoGlot selects one based on pre-determined order of rules that can be chosen arbitrarily to minimize the user effort.

transducer prevents that. Instead, the transducer will be forced to dynamically try out different rules at each step, and repeatedly will do so until it finds the choice that works.

Resorting to non-determinism by itself may lead to combinatorial explosion as there are exponentially many rule combinations to be tried—several options exist at each step of the top-down traversal of the transducer on the input AST. The key observation is that among the many syntactic candidates at each step of the traversal, there are only a few syntactically correct choices and even fewer that eventually pass the given tests. Therefore, DuoGlot eagerly validates the candidates using a syntax checker at each step, and a semantic correctness checker that uses unit tests after a sequence of steps. This *incremental* validation significantly reduces the number of valid translation rule candidates that remain after a few steps of transduction. For example, a transduction step translating `a[0] = 5` to `let . = .` followed by a transduction step translating `a[0]` to `.[.]` will be rejected eagerly by the syntax checker because `let a[0] = 5` is not syntactically valid. Similarly, the semantic correctness checker helps. The example in Figure 8 (middle) shows translation of the Python code `if x in y` to JavaScript. There are three possible syntactically valid translations in JavaScript: `y.indexOf(x)>= 0`, `y.has(x)`, and `x in y`. However, only one of them may be semantically correct, depending on dynamic type of `y` (which needs to be a set). DuoGlot's semantic checker implements a test-retry procedure to filter out such invalid translation choices. The procedure is invoked every time a syntactically valid candidate translation is generated so that it can run the unit tests to verify the correctness. Therefore, the transducer will produce a full translation by choosing the first option and by running the code with unit tests. If the runtime type of `y` is not an array, then an error from that line will be fed back to DuoGlot so that it can immediately move to the next option. We emphasize that this is done automatically, without any user interaction, and on average requires from one to a few seconds to repair a dozen of errors.

The second type of ambiguity (multiple functionally correct translations) is illustrated in Figure 8 (right). Such ambiguity can be resolved in two ways: ask the user which rule they prefer to apply, or choose the rules according to an arbitrarily pre-defined order. DUOGLOT allows both, and to minimize user interaction, by default it uses the latter. Specifically, it picks translation rules at each transduction step in a pre-determined order and moves forward until it either translates the whole program to a semantically correct translation or it gets stuck and backtracks.
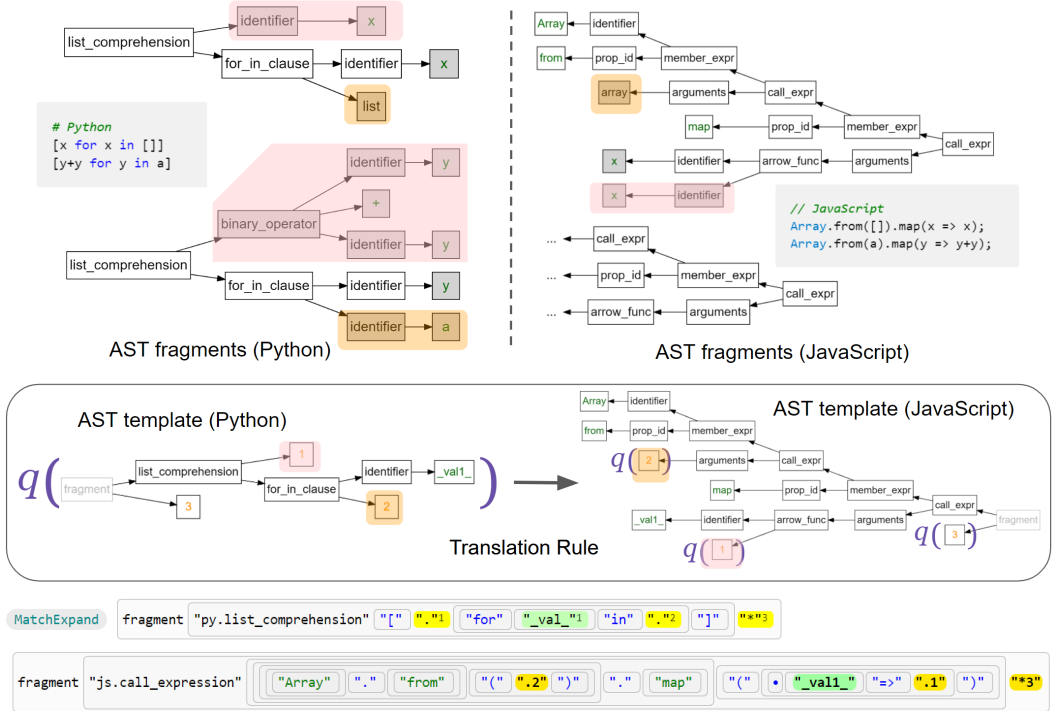

Fig. 9. The translation rule for translating list comprehension (bottom) is inferred from the AST fragments obtained from user-provided Python (left) and JavaScript (right) code snippets. The grey-shaded nodes are the non-common subtrees in the AST fragments, while the unshaded ones are the common ones. The pink-shaded subtrees are matched to the same hole (1) in the resulting AST template for each respective language. Similarly, the yellow-shaded subtrees are matched to same hole in their respective ASTs (2). q(...) refers to subtrees of the source AST to be transduced.

When DUOGLOT has no valid rule left to try to translate the input AST, it interacts with the user to help derive a rule. It asks the user to provide code snippets in source and target languages. The third challenge DUOGLOT addresses is how to infer new rules fully automatically or with a minimal manual effort. For instance, in Figures 3 and 4 we show user-provided replies to a rule inference query for line 2 of Figure 2. It is critical to understand that the query is localized at a single context in the source program, i.e., DUOGLOT prompts the user to specify how to translate one construct found in the code. The user can provide snippets easily because it is not necessary to understand the global context of the construct's use in the program.

DUOGLOT automatically infers the translation rules by matching patterns observed in the ASTs of code snippets. The procedure is deterministic and works well in practice. Figure 9 shows an example of how DUOGLOT's rule inference engine learns a translation rule from the snippets illustrated

in Figure 3. Recall that in our running example we have two list comprehensions in Python for which we want to obtain the corresponding JavaScript translations. Briefly, DuoGlot identifies an AST template that summarizes the two AST fragments corresponding to the list comprehensions in the Python snippets. The learned AST template consists of nodes that are common to both the AST fragments (not shaded boxes in Figure 9) and holes that abstract away the uncommon parts between the AST fragments (grey-shaded boxes in Figure 9).

The same procedure creates the AST template for the JavaScript snippets provided, as shown in the right part of Figure 9. The AST templates' holes are represented by numbers in the bottom part of the figure. The translation rule is then simply the transform of the Python AST template to JavaScript AST template, i.e., the fixed AST nodes of the source AST will be translated to the fixed nodes of the target AST. The translation will leave the Python fragments matching the holes to be translated in the subsequent steps of the transducer. We leave the details of how the AST templates are created and the handling of holes to Section 4. Figure 9 also shows the learned rule in our DSL which can be inspected and optionally edited by the user to directly customize the inferred rule.

4 DUOGLOT: KEY COMPONENTS

DuoGlot's four key components, the transducer (Section 4.1), the syntax and semantic checkers (Section 4.2), and the rule inference procedure (Section 4.3) allow efficient translation of a program by incrementally learning the required rules from small code snippets. All of the key components are designed to work with any pair of dynamically typed scripting languages and DuoGlot demonstrates this for Python and JavaScript.

4.1 Non-Deterministic Single-State Tree Transducer

The transducer translates different parts of the source AST recursively by applying the translation rules. We design a domain-specific language (DSL) for writing the code translation rules, as shown in Figure 10. DuoGlot requires a set of translation rules, which corresponds to the start symbol **Ruleset**. The **Ruleset** is a sequence of rules where each rule has a source and target pattern. The source pattern (**SrcPattern**) matches a source AST fragment which could be the whole tree or a subtree of the source AST. Similarly, the target pattern (**TrgPattern**) matches a target AST fragment. Therefore, when a source AST fragment matches with the source pattern in the current application of **Rule**, it can be translated into the corresponding target AST using the target pattern. Each pattern is a sequence of one or more **NodeMatcher**s

```
       Ruleset ::= Rule+
          Rule ::= (MatchExpand SrcPattern TrgPattern)
    SrcPattern ::= (fragment NodeMatcher+)
    TrgPattern ::= (fragment NodeExpander+)
   NodeMatcher ::= NTMatcher | TMatcher |
                   NTMatcherTpl | TMatcherTpl |
                   Capture
  NodeExpander ::= NTExpander | TExpander |
                   NTExpanderTpl | TExpanderTpl |
                   Reference
     NTMatcher ::= (NTSymbol NodeMatcher+)
    NTExpander ::= (NTSymbol NodeExpander+)
      TMatcher ::= (str String) | (nostr) | (val String)
     TExpander ::= (str String) | (nostr) | (val String)
  NTMatcherTpl ::= ("_nt_" NodeMatcher+)
 NTExpanderTpl ::= ("_ntIdx_" NodeExpander+)
   TMatcherTpl ::= "_str_" | "_val_"
  TExpanderTpl ::= "_strIdx_" | "_valIdx_"
       Capture ::= "." | "*"
     Reference ::= ".Idx" | "*Idx"
      NTSymbol ::= "Lang.NTName"
```

String: Arbitrary string   **Idx**: Positive integer
Lang: Name of the source or target programming language
NTName: A valid non-terminal in the corresponding language

Fig. 10. Translation Rule DSL for the tree transducer.

which describes the "nodes" in the corresponding AST fragment. A node can be a non-terminal symbol (such as forStatement and ifStatement), a terminal symbol (any string), or another fragment altogether. Therefore, the fragment present in the current rule can be translated recursively using the rule that applies for that fragment. The recursive fragments are denoted by one of the placeholders ".", and "*". The former matches a single arbitrary AST node representing a

non-terminal symbol whereas the latter represents a sequence of AST nodes including non-terminal and terminal symbols. In the source pattern, these placeholders are represented by the **Capture** rule, whereas in the target pattern by the **Reference** rule. Each **Reference** in the target pattern corresponds to one of the **Capture**s in the source pattern. Therefore, the **Reference**s are just **Capture**s that are left as-is by the current rule until the next rule can proceed to match and expand (translate) them.

Formally, a transducer can be characterized by the tuple $(Q, \Sigma, \Gamma, I, \Delta)$ where Q is a set of states of the transducer, Σ and Γ are ranked alphabets for source and target languages, $I \subset Q$ is a set of initial states, and Δ is a set of translation rules. Since our transducer is single-state there is only one state $Q = q$ and $I = q$. This tree transducer takes as input an AST tree in alphabet Σ and generates an AST tree in alphabet Γ using the rules specified by Δ. Each rule in Δ takes the form $q(F(x_1, x_2, ..., x_n)) \rightarrow G(q(x_{i_1}), q(x_{i_2}), ..., q(x_{i_m}))$, where F is a multi-level tree on Σ with n subtrees represented as $x_1, \cdots, x_n$, and G is a multi-level tree on $\Gamma \cup (Q \times \{x_1, ..., x_n\})$ with m subtrees to be translated under state q, represented as $q(x_{i_k})$ and $i_k \in \{1, .., n\}$. Each x_i is the placeholder matched by **Capture**, whereas $q(x_{i_j})$ is the one matched by **Reference**.

For example, consider two simple languages of simple arithmetic expressions: the source language has alphabet $\Sigma = \{``\times", ``+", a, b, c\}$, and the target language has alphabet $\Gamma = \{\text{Add}, \text{Mult}, A, B, C\}$. Given a rule $q(``\times"(``+"(x_1, x_2), x_3)) \rightarrow \text{Add}(\text{Mult}(q(x_1), q(x_3)), \text{Mult}(q(x_2), q(x_3)))$ that matches on multiple nodes directly, in one step the transducer can translate the source tree $``\times"(``+"(a, b), c)$ in Σ to a transduced tree in in Γ in the form $\text{Add}(\text{Mult}(q(a), q(c)), \text{Mult}(q(b), q(c)))$. When 3 additional rules are provided as $q(a) \rightarrow A$, $q(b) \rightarrow B$ and $q(c) \rightarrow C$, then the transduction will finish and will result in $\text{Add}(\text{Mult}(A, C), \text{Mult}(B, C))$ in alphabet Γ. Observe that our DSL is non-deterministic as it supports learning other rules (such as $\delta_2 = q(``\times"(x_1, x_2)) \rightarrow \text{Mult}(...)$) for the same or covered LHS. Therefore, it supports ambiguous translations due to differences in the handling of language constructs and customized rules for each user based on their coding styles.

Our transducer is designed to minimize learning new rules. First, as shown in the previous example, our transducer can apply the rule over multiple nodes in the source pattern rather than just one at a time: the rule is directly applied over both nodes "$\times$" and "$+$" at the same time rather than applying on "$\times$" first and then on the "$+$". In the latter case, the final translation would have been $\text{Mult}(\text{Add}(A, B), C)$ and would have required one additional rule over the former. Second, to translate all symbols in the source alphabet Σ, every symbol would have to be addressed by some translation rule. This can be tedious if the alphabet is huge. Instead, we write (or infer) a template rule that can encompass many similar operators, identifiers, literals and so on. For example, a template node "_str_" (see nodes with suffix "**Tpl**" in Figure 10) can be created for all operators "<, >, ≤, ≥" to replace the rules corresponding to each individual operator by just one umbrella rule containing the template node.

4.2 Syntax and Semantic Checkers

If R is the set of rules that are necessary and sufficient to obtain a valid translation, then the goal is to search for a trace of rules $\tau : [r_1, r_2, \cdots, r_m], r_i \in R$ that can be applied to the source AST. This trace is conventionally called the leftmost derivation. The search for such a trace is more complicated if multiple rules can be applied for the same AST fragment which is usually the case. If there are even 2 rules for each node in the AST then the total number of rule traces quickly raise up to $2^{\#\text{nodes}}$. Further, DuoGlot does not know R beforehand as it starts with a small set of initial rules $R_I \subset R$. Therefore, DuoGlot's incremental procedure has to search for R along with the search for the correct leftmost derivation τ to apply.

To facilitate this search, DuoGlot uses a syntax checker to enumerate only the ASTs that lead to syntactically correct translations and uses a semantic checker to further guide the search toward

finding a semantically correct translation. When DuoGlot cannot find more rule options to apply, it prompts the user to provide code snippets to learn a new rule (later described in Section 4.3).

Syntax Checker. This component uses a pushdown automaton to keep track of the result of every transduction and reject the ones that result in syntactically invalid translations. Specifically, the automaton is parameterized by the target grammar and the transductions of every intermediate step are provided as inputs to the automaton. The automaton consumes the provided inputs and checks whether the partially translated subtree can be expressed using the target grammar. If the automaton results in an error state, then the current (latest) transduction is rejected along with all further transductions that would build on top of the current transduction. If the automaton accepts a fully transduced AST, then a parse tree, with more information than the AST, can be reconstructed from the automaton's logs. This parse tree is pretty printed and outputted as a candidate translation.

We use parsing expression grammars (PEGs) [Ford 2004] to define the target language syntax. A PEG can be seen as a CFG with production rule priorities. Such grammars are often easy to write and widely used in practice [van Rossum et al. 2020]. With the PEG grammar and additional annotations about how the AST is mapped to the parse tree, the automaton automatically and incrementally checks an AST's syntactic validity. Formally, the non-deterministic pushdown automaton M used in DuoGlot is defined as a tuple $(Q', \Sigma', \Gamma', \Delta', q_0', Z', F')$, where $Q' = \{$Start, Accept, Error$\}$ is a finite set of states, $q_0' =$ Start is the start state, $F = \{$Accept$\}$ is the accepting state, Z' is the initial stack containing the start symbol of the target PEG grammar, Σ' is the input alphabet for representing AST as a sequence of tokens, Γ' is the stack alphabet containing terminals and non-terminals with annotated information of those symbols, and Δ' is a set of transition instructions. The input sequence of the automaton is the preorder traversal of a partially-translated AST. The leaf nodes of the AST representing terminal symbols are allowed to be missing in the input sequence. The automaton will only enter the accept state if the stack is empty when the end of the input has been reached. We omit the details of the alphabet and transition instructions to construct such an automaton as it follows a standard textbook procedure [Sipser 1996].

Semantic Checker. The semantic checker further helps the search for the correct translation by implementing a test-retry procedure. A simple strategy to find the correct translation among the syntactically valid candidate translations generated by the transducer would be to test all of them one by one using the unit tests. There are two issues with this strategy. First, the number of syntactically valid translations can also be potentially exponential in the code size. Second, if none of the enumerated candidate translations is correct, then the transducer might have to learn a new rule to do the right translation. But, it will only know this after testing all the enumerated candidate translations. To deal with these issues, DuoGlot tests the candidate translations as they are being generated using the provided unit tests. When a candidate translation fails the tests, in many cases such as undeclared variables, mishandled dynamic types, and unsupported operations the location of the failure is evident. In such cases, DuoGlot applies a new rule at that location to generate a new candidate translation. In essence, DuoGlot sequentially repairs a portion of the code in a candidate translation and continues testing it for the next incorrect portion of the code. In practice, we observe that this strategy often finds the correct translation quickly.

It is also possible that there are no runtime errors but the outputs of the unit tests on the translated code are different from their counterparts on the source code. This kind of problem can be caused by subtle semantic differences among the built-in functions (such as `int(x)` vs `parseInt(x)` on strings), built-in operators (such as integer overflow), variable scoping (referring to the wrong variable), runtime behaviors (such as iteration order of a dictionary), and so on. In this case, if there is no line number to debug, the searcher will simply coordinate with the core to retry all possible translations until it reaches some pre-determined retry limit of the test-retry loop or returns the

---

Algorithm 1 Rule Inference Algorithm

1: **procedure** RULEINFER(SrcASTFragments, TarASTFragments)
2:      LHS = SIMULTANEOUSTRAVERSAL(SrcASTFragments)   ▷ AST template for source
3:      src_PHs, src_tpl_PHs = GETPLACEHOLDERS(LHS)
4:      RHS = SIMULTANEOUSTRAVERSAL(TarASTFragments)   ▷ AST template for target
5:      tar_PHs, tar_tpl_PHs = GETPLACEHOLDERS(RHS)
6:      **for** tar_PH **in** tar_PHs **do**   ▷ Compute pairwise tree-edit distance source-target
7:          refer_idx = argmin_i(TREEEDITDISTANCE(src_PHs[i], tar_PH))
8:          tar_PH.SETREFERENCEIDX(refer_idx)   ▷ Find the smallest distance for each target PH
9:      **for** tar_tpl_PH **in** tar_tpl_PHs **do**   ▷ Match terminals
10:         tpl_idx = argmin_i((src_tpl_PHs[i] == tar_tpl_PH) ? 0 : 1)
11:         tar_PH.SETTEMPLATEIDX(tpl_idx == None ? "Unexpected" : tpl_idx)
12:     **return** CREATERULE(LHS, RHS)   ▷ Wrap LHS, RHS in the DSL format rule

---

first translation that passes unit tests. As future work, other fault localization techniques [Artzi et al. 2010; Ocariza Jr et al. 2016; Shen et al. 2021] could improve our approach for such cases.

Search Optimization. To make the search and check more efficient, we use lazily constructed transduction trees, dynamic programming, and caching of states of the pushdown automaton for efficient backtracking. With such optimization, the execution time to search for new candidate translations is negligible compared to the running time of the test-retry procedure. The precise details of the optimization are provided in the supplementary material.

4.3 Inferring Translation Rules

After the steps of the syntactic and semantic check, when DUOGLOT does not find a suitable rule to apply, it proceeds to infer a new rule. The rule inference problem consists of producing a pair of syntactic patterns and a corresponding translation rule from the user-provided code pairs of source-target language snippets. DUOGLOT uses one or at most two pairs of snippets. The inferred rule has the source pattern (AST template) on the left-hand side and a target pattern (AST template) on the right-hand side, as shown in Figure 9.

The rule inference problem consists of three sub-problems. The first problem is to extract the ASTs for the instances of the pattern that appear in both the source and target code snippets. The second problem is to extract one AST template that summarizes both the ASTs in each code snippet. The third problem is to generate a rule that translates the AST template from the source snippets to its corresponding one from the target snippets.

From Code Snippets to ASTs. To address the first problem, DUOGLOT assumes that a user provides only one instance capturing the syntactic pattern of interest per line and any additional code not matching that syntactic pattern is minimal. Further, it assumes that all of the instances in both source and target snippets are either expressions or statements as it is unlikely to translate between expressions and statements. DUOGLOT then automatically extracts the AST for each instance, representing an expression or a statement, for every code snippet. For example, in Figure 9, the ASTs are automatically extracted from Python and JavaScript expressions from each line. If the extraction fails due to identifying unintended instances, DUOGLOT also allows users to highlight the intended instances that capture the syntactic patterns.

Extracting AST Templates. Given the resulting ASTs from source and target code snippets, our rule inference procedure infers a translation rule. The algorithm for rule inference is given in

Algorithm 1. To find the AST template for the ASTs in the source code snippets the algorithm finds the largest common AST fragment between the source ASTs. Specifically, our algorithm at line 2 calls the SIMULTANEOUSTRAVERSAL subprocedure[5] which matches the nodes of the two ASTs by using simultaneous pre-order traversals on both of them. The subtrees that are different will be abstracted away and replaced by placeholders (or template placeholders if the differing subtree is a terminal) in the extracted AST template (see line 3). The source AST template is the LHS of the rule being inferred (see left of the translation rule in Figure 9). After that, the same SIMULTANEOUSTRAVERSAL procedure is applied to target ASTs (line 4) to find the AST template for the RHS of the rule.

AST Templates to Rules. Recall that according to our rule DSL (see Section 4.1) the placeholders on the LHS are **Capture**s and the placeholders in the RHS are **Reference**s. The **Reference**s and template expanders in the RHS are still placeholders and have not been connected to **Capture**s and template matchers in the LHS yet. For each template expander, the connection to template matchers is simply computed by string comparisons (see lines 9-11). We next connect the **Reference**s with **Capture**s in the template AST (see lines 6-8). The idea is to connect a **Reference** to the **Capture** that is most similar to it based on the subtrees they encode. Each **Reference** encodes two subtrees, one each from the target ASTs and each **Capture** encodes two subtrees, one each from the source ASTs. The similarity between a **Capture** and a **Reference** is computed by summing up the pairwise similarities between their corresponding subtrees. We use tree-edit distance as a similarity metric to compute the similarity between two subtrees. The tree-edit distance is affected by the subtree structure, node names, and node values in the terminals. The tree edit distance is computed using the APTED algorithm [Pawlik and Augsten 2016] and the distance between nodes is defined as Levenshtein string edit distance for node names and terminal node values [Levenshtein et al. 1966]. The obtained rule for our running example is given in Figure 9. The connections between **Reference**s and **Capture**s are inferred based on an assumption that the translation of a specific subtree in the source AST has the smallest tree edit distance to its corresponding subtree in the translated AST compared to all other translated subtrees. In practice, we find that this assumption holds for almost all of the code snippets we provided. When it does not hold, the user may manually choose corresponding **Reference**s of **Capture**s in a rule editor.

To summarize, it is up to the user to decide if the rule-inference procedure is automatic or manually guided. Furthermore, the rule-inference procedure is not particularly specific to Python and JavaScript, and in a similar fashion can work with other languages as all it needs is the ability to construct ASTs out of code snippets provided in the two languages.

5 IMPLEMENTATION

DUOGLOT is open source [Wang 2023]. It is implemented in a modular fashion. DUOGLOT's core module implements the tree transducer and the syntax checker with optimizations mentioned in supplementary materials. The core module exposes an interface for interacting with external procedures to select rules during the search for correct translation. DUOGLOT also contains peripherals that enable automating the test-driven incremental rule learning. Furthermore, it supports a plug-and-play user interface that allows users to control and customize all phases of translation.

5.1 DUOGLOT Core

Input and Output. DUOGLOT's core focuses on translating the AST of the source code to an AST in the target language. It requires two inputs: the AST of the source code and a set of rules to start the

---

[5]Refer to the supplementary material for details.

translation procedure (see Figure 11). The AST for source code is generated by a Tree-sitter parser [6] for the source language. DuoGlot also requires the PEG grammar definition for the target language which can be obtained from the Tree-sitter package for the target language. DuoGlot finds a trace of rules that transduce the input AST to a syntactically-valid AST in the target language. It outputs such a candidate AST along with a mapping of rules that have been used to generate it.
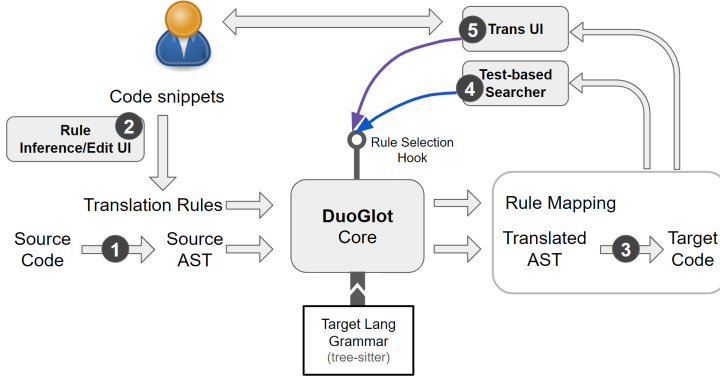

Fig. 11. DuoGlot's modules and inputs/outputs.

Rule Selection Hook. The core implements a non-deterministic tree transducer so there might be multiple rules that are applicable at the same translation step. Some applications are wrong or may not be the preferred ones, depending on the source program, user's intention and the unit tests. Therefore, the hook allows for arbitrary external modules or search heuristics to interact with DuoGlot's core to select or prioritize rules that are applicable at the same translation step. The user can also choose the rules that can lead to an intended translation.

Parametrization. The syntax checker inside the core is parameterized by a parser expression grammar (PEG) definition for the target language. We implement the checker to accept Tree-sitter's grammar DSL for that purpose. The tree-sitter grammar definitions for many common programming languages are already used in production [Brunsfeld 2018]. Therefore, DuoGlot's core can be easily extended to support other scripting languages with existing Tree-sitter grammar definitions.

5.2 DuoGlot **Peripherals**

DuoGlot's core outputs a candidate syntactically-valid AST and does not contain the test-repair procedure or the modules to convert between AST and exact code. These features are implemented in the peripheral modules (see Figure 11).

Parser and Pretty Printer. Circles 1 and 3 in Figure 11 represent the parser for the source language and pretty-printer for the target language respectively. For the parser, we use off-the-shelf Tree-sitter parser to get AST from source code. The pretty-printer is used to convert the translated AST, in the Tree-sitter format, to code in the target language. There are no existing implementations for such pretty-printers. So we implement a meta pretty-printer that is parameterized by the Tree-sitter's PEG for the target language. In addition, we also handle some corner cases for each language such as terminals and non-context-free special cases (such as Python's indentation).

Test-based Searcher. When the DuoGlot's core generates a candidate translation it is passed on to the test-based searcher. The searcher runs the unit tests to verify its correctness. Circle 4 in Figure 11 shows the test-based searcher implementing the testing of the candidate translations

---

[6]Tree-sitter is a popular parser-generator used in production. It accepts PEG grammar definition.

using the unit tests and error analysis. If the translation passes all the unit tests, the searcher will output the translation. If the translation fails unit tests with a fault line number, the searcher will guide its search in the following way. It will first look at the rule mapping to compute the segment of the derivation history that corresponds to the faulty line of code. Then the searcher will choose other possible rule selections in that segment and send the choices back to the DUOGLOT core. Note that there could be multiple rule applications corresponding to one line of code. Therefore, in the worst case, the searcher will have to try all possible combinations of applicable rules on that line. Hence, the searcher will first choose to change only one rule at a time, then proceed to change two rules and so on. When the core receives the updated choices, it will produce another candidate translated AST that obeys the new rules selected by the searcher. The new candidate translation will be tested again. If the error on that line cannot be fixed after trying all alternatives on the fault line, the searcher will get stuck and prompt the user for new code snippets.

We impose a limit on number of test-retry iterations, called the retry limit. For instance, in our main evaluation we set the retry limit to 20. So within these, say 20, retries, 1) the DUOGLOT may produce a translation that passes all unit tests, 2) it may get stuck because it has exhausted all rules to apply at a line, 3) all the 20 candidate translations raise runtime errors, or 4) the last candidate translation does not have runtime errors but still fails to have the same outputs as the source code. Only when the DUOGLOT gets stuck (second scenario), the user is prompted for new code snippets to learn a new rule and it restarts its test-retry procedure again.

Trans UI (User-based Rule Selection). Circle 5 in Figure 11 represents a component of DUOGLOT's user interface. The component allows users to see the translation procedure. Users can inspect a candidate translation and its rule mapping. Users can also change the rules at each translation step during the translation procedure to obtain a different candidate translation. A screenshot of this component is given in the supplementary material.

Rule Inference/Edit UI. Circle 2 in Figure 11 represents the rule inference/edit component of DUOGLOT's UI. Here, users input the code snippets and inspect the automatically inferred rules. Expert users may also manually edit the rules if they want. A screenshot of this component is provided in the supplementary material.

6 EVALUATION

We show that DUOGLOT is one of the best practical tools for translating between untyped scripting languages such as Python and JavaScript. We plan to submit an artifact with our code and benchmarks. Our evaluation is structured as follows.

(1) **Quantitative (Sections 6.2 and 6.3)**: We measure the number of benchmarks that DUOGLOT can solve correctly and the number of rules it requires to solve them. We then compare DUOGLOT to several existing translators on correct translations and their readability.

(2) **Qualitative (Section 6.4)**: We manually categorize the rules based on their complexity and identify how the rule complexity affects performance. We discuss as well the additional human effort required by DUOGLOT as compared to other state-of-the-art transpilers.

(3) **Case Studies (Section 6.5 and 6.6)**: We present two case studies to evaluate the abilities and limitations of our tool in scenarios where the test coverage varies and the code is either more complex or longer.

6.1 Evaluation Setup

We evaluate DUOGLOT for translating code from Python to JavaScript on popular benchmarks.
Benchmarks. We use GeeksForGeeks (GFG) benchmarks [MetaResearch 2022] from the Transcoder project [Roziere et al. 2020] as our main evaluation benchmarks, and two additional sets of programs

for our case studies (refer to Section 6.5 and 6.6). The GFG benchmarks consists of 702 standalone Python scripts with unit tests, each having a target function for translation. We remove 3 scripts that we could not run (contain errors), resulting in 699 Python scripts to evaluate on. The target functions vary in lengths in the range of 1-40 lines (8.53 on average), or 23-974 non-space characters (154 on average).

Unit Tests. The benchmarks provide unit tests for the source Python code (which achieve 99% line coverage), but not for the target JavaScript. However, Python unit tests are straightforward to translate to JavaScript, either with a simple ad-hoc translator for unit tests as done in previous works [Roziere et al. 2020] or with a simplified version of DuoGlot (without test-repair strategy). We use the latter and manually verify that all of the unit tests have been correctly translated.

DuoGlot Setup. To translate Python to JavaScript, we create a pipeline using DuoGlot Core and DuoGlot Peripherals described in the previous section. We turn off the part of the user interface that allows manually choosing alternative rules and set the retry limit for the test-retry loop (running unit tests) to 20 iterations. We choose 20 as we want DuoGlot either to finish translating the source code within 1-2 seconds, or if it fails, to ask for new code snippets. At the beginning, DuoGlot starts with 44 base rules for direct mapping of the most common AST nodes, such as identifiers, number literals, string literals, assignments, etc. from Python to JavaScript. We include these base rules by default (even though end users can also create or modify them), to avoid repeated work of redefining them in all current and future applications of DuoGlot.

Library Calls. The GFG benchmarks depend only on built-in Python libraries and do not make calls to third-party libraries. Limiting calls only to built-in libraries is common when testing Python to JavaScript translations as it allows effective comparison of core capabilities of transpilers. Note, DuoGlot does not have fundamental limitations in handling third-party libraries (such as NumPy and pandas), however, it will require the existence of similar libraries in JavaScript.

Evaluating Accuracy. To confirm the correctness of the translations we use unit tests. We compare the outputs of the source and the translated programs on unit tests, including their intermediate logs (prints). We say a translation is correct only if the outputs match on all unit tests. Accuracy then is the percentage of benchmarks that have been correctly translated.

Evaluating Readability. We assess the readability based on the translated code size [Weinberg 1971] and assume smaller size is better because it preserves high-level abstractions as it precludes translations that have many additional unnecessary variables or redundant code. We prefer this to other readability metrics (such as [Buse and Weimer 2008]), because our main interest is to stay at the same abstraction level as the source program, rather than to focus on human factors such as identifier naming, quality of comments, etc. We express the readability as the bloating ratio between translated code and the source code of the number of non-whitespace characters.

Compared Transpilers. We compare DuoGlot with other state-of-the-art transpilers, both ML-based and hand-crafted. Among ML-based transpilers, we evaluate Codex [Chen et al. 2021] (the API used by Github Copilot) and TransCoder [Roziere et al. 2020]. For hand-crafted transpilers, we search for open source projects on GitHub that are relevant to Python to JavaScript translation. We choose Transcrypt (2.5k stars[7]), Javascripthon[8] (824 stars), and py2js[9] (90 stars) as they also aim to produce correct and readable JavaScript code. To avoid unfair comparison, we use the following criteria that generally *overestimate* the performance of some of these tools:

---

[7]Number of stars means number of developers on GitHub that keep track of the project.
[8]Python 3 to ES6 Javascript translator. https://github.com/metapensiero/metapensiero.pj
[9]Python to JavaScript translator. https://github.com/qsnake/py2js

- **TransCoder**: We take the 10 most probable translations (i.e., with beam size 10) and run tests on all 10 translations. We count it as successful if any of the 10 translations can pass all the unit tests.
- **Codex**: We take one possible translation from Codex API. We clean up the translation by removing the additional code or text after the first JavaScript function in the translation. If there is a typo in the translated function name, we fix it by string replacement. We count it as successful if the cleaned and fixed translation can pass all the unit tests.
- **Transcrypt, Javascripthon, and py2js**: We run them once for each benchmark to get the translated code. We count it as successful if the translation passes all the unit tests.

System Specification. We use a desktop with an Intel i7-9700 8-core CPU and 32GB of RAM. The DuoGlot Core, test-based searcher, and the evaluation script for the non-ML-based tools are run on one CPU core each. For TransCoder, we use a machine with AMD Threadripper 3970X 32-Core Processor with 64GB of RAM and a GeForce RTX 3090 GPU. For Codex, we use OpenAI's API (`code-davinci-001`) directly without any local computation.

6.2 Performance of DuoGlot on GFG Benchmarks

DuoGlot increases its accuracy by inferring rules learned from user-provided code snippets. With no learned rules (only the bootstrapped 44 rules mentioned in the setup), its accuracy is 0% on the 699 benchmarks, but this grows to 90% with 142 total rules. We plot the relation between the number of rules and the percentage of benchmarks that can be solved by a specific number of rules in Figure 12. As evident, the accuracy quickly increases from 0% to around 60% as the number of total rules reaches 80, and 80% with 98 rules. After that, the accuracy increases slowly and it reaches the top 90% at 142 rules.
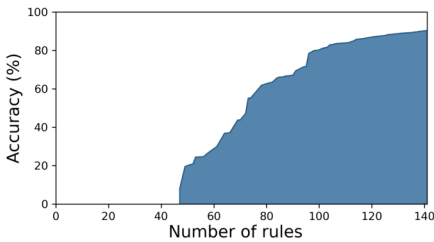

Fig. 12. Accuracy of the translated benchmarks from GeeksforGeeks as a function of the number of used rules. For the first 80 rules, the accuracy increases from 0% to 60%.
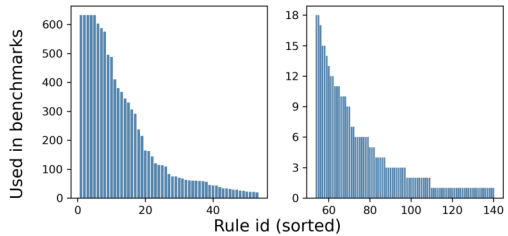
Fig. 13. Distribution of the 142 rules across the translated benchmarks. The top 15 most popular rules are presented in around half of the benchmarks, and all of these rules are the base rules.

We provide as well the distribution of the rules across the translated benchmarks, refer to Figure 13. The top 15 most popular rules are present in up to one-half of the benchmarks. As expected, all of these rules are from the bootstrap set (base rules). The usage of the remaining rules varies, from hundreds of benchmarks for the more used, to only a handful for the less applied.

6.3 Comparison to Other Transpilers on GFG Benchmarks

Further, we provide comparisons of the features of all transpilers and refer the reader to Table 1.
Accuracy Comparison. The accuracy of all the transpilers under the considered evaluation setting is shown in Table 1. DuoGlot achieves better performance than all hand-crafted transpilers such as Transcrypt, Javascripthon, and py2js. Furthermore, it also outperforms Codex even after fixing minor typos in the translations. For TransCoder, a direct comparison is not possible, so for reference

we only list TransCoder's performance from Python either to C++ or to Java. Due to the lack of data and resources to re-train TransCoder for JavaScript target, we can only speculate that when trained on JavaScript, TransCoder's performance on Python-JavaScript translations will be comparable to that of its Python-C++ translation.

Table 1. Comparison of accuracy, readability (bloating ratio) and execution times of different transpilers.

Translate Python to Javascript	**Accuracy**	**Readability**	**Time**
DuoGlot (142 rules)	90%	1.34x	0.46s
Codex [10]	35%	1.2x	7.33s
Codex (fixed typos) [11]	74%	1.2x	7.33s
Transcrypt	76%	10.7x[12]	0.16s
Javascripthon	43%	2.1x	0.06s
py2js	21%	2.2x	0.03s
Other Settings	**Accuracy**	**Readability**	**Time**
TransCoder (to Java, beam size 10)	67%	-	1.36s
TransCoder (to C++, beam size 10)	79%	-	2.85s

Readability Comparison. DuoGlot provides as well one of the top readabilities (quantified as the bloating ratio) among all the tested transpilers (refer to Table 1). On average across all accurately translated benchmarks, DuoGlot increases the size of the original Python program into the corresponding JavaScript program by a factor of 1.34× (varies in the range 1.06-2.22). That is best only next to Codex's 1.2× readability. On the other hand, the remaining transpilers have a substantially larger bloating ratio: 10.7× for Transcrypt without considering the size of the API emulation library, 2.1× for JavaScripthon, and 2.2× for py2js.

Time Comparison. Once DuoGlot obtains all the rules to translate a benchmark, the total average time spent to translate a benchmark including testing is 0.46s. The average time for the other manual transpilers is as follows: py2js requires 0.03s, JavaScripthon 0.06s, and Transcrypt 0.16s. The time for TransCoder to perform Python to C++/Java translation, with beam size 10, is 2.85/1.36s on average per benchmark. Finally, API calls to OpenAI's Codex require 7.43s on average per benchmark, including the round trip time. Thus the execution time of DuoGlot is between hand-crafted and ML-based transpilers. Note, the former translates faster due to the lack of a test-repair loop. This loop in DuoGlot, however, provides a time-accuracy tradeoff: at the expense of extra computations, significantly improves the correctness of the translation. As such, it is arguably beneficial as the execution time is still very practical.

6.4 Rules Classification and Manual Effort

DuoGlot achieves 90% accuracy on the tested benchmarks with 142 rules. Aside from the 44 base rules, we can categorize the remaining 98 rules based on their complexity into three classes: Simple, Medium and Complex. The complexity is determined by the number of occurrences of **Capture**s and **Reference**s (refer to Section 3): the higher the number of occurrences, the more complex code snippets (in terms of length) are required to infer the rule. Simple rules have 2-6 occurrences, Medium 7-9, while Complex rules have more than 9 occurrences of **Capture/Reference**. We

---

[10]The model we are using is `code-davinci-001` and the prompt we are using is '##### Translate this function from Python into JavaScript\n### Python\n\n<Python Code>\n \n### JavaScript\n"use strict";\n'

[11]For many of the benchmarks translated by Codex, the function name is wrong and we fix it by string replacement.

[12]Transcrypt prepends a lot of "imports" in its translations for emulating Python functions in JavaScript.

Table 2. Examples of snippets for rules from Base, Simple, Medium, and Complex categories

Category	Python code snippets	JavaScript code snippets
Base	`None`	`null;`
Base	`a = 5` `[c, d] = [e, f]`	`a = 5;` `[c, d] = [e, f];`
Simple	`not x` `not y == 0`	`!(x);` `!(y === 0);`
Simple	`a in xs` `2 in []`	`xs.indexOf(a) >= 0;` `[].indexOf(2) >= 0;`
Medium	`list1.remove("bob")` `[1,2,3].remove(1)`	`list1.splice(list1.indexOf("bob"), 1);` `[1,2,3].splice([1,2,3].indexOf(1), 1);`

Medium	`for x in range(10): pass`
	`for y in range(a):`
	`    break`
		`for (let x = 0; x < 10; x++) {}`
		`for (let y = 0; y < a; y++) {break;}`

Complex	`{x["id"]:x for x in xs}`
	`{y:y*y for y in [1,2,3]}`
		`Array.from(xs).map((x) => [x["id"], x])`
		`  .reduce((a, b) => (a[b[0]] = b[1], a), {});`
		`Array.from([1,2,3]).map((y) => [y, y*y])`
		`  .reduce((a, b) => (a[b[0]] = b[1], a), {});`

emphasize that the complexity of the rules does not affect the number of needed code snippets for their inference, which is always one or two. In total, among the 98 rules, 82 (84%) are Simple rules, 9 (9%) are Medium, and 7 (7%) are Complex rules. Hence, among all rules, an overwhelming majority are simple. In Table 2, we provide examples of rules from each category, described with their inferring code snippets.

The manual effort required to run DuoGlot can be divided into two types. The first type dominates the effort and consists of providing the aforementioned code snippets used for inference of new rules for the incremental transpiler. Across all rules, we provided in total only 360 lines of snippets (for *both* Python and JavaScript) containing 5389 non-empty characters. The second type of effort is the occasional manual adjustment of the rule inference, when the expected rule differs from the inferred. We performed this adjustment for 15 rules. Each adjustment consists of a handful of clicks (depending on the rule complexity) in the rule editor to identify the connections between **Reference**s and **Capture**s, as specified in Section 4.3. The second type of effort can be eliminated completely, with more advanced implementation techniques. Finally, we emphasize that the manual effort is a one-time cost per rule, and distinct rule sets can be merged and expanded.

6.5 Case Study: Translating Solutions for LeetCode Programming Exercises

LeetCode[13] is a popular, multi-million user online platform for coding contests and problems. There are more than 2000 problems, to which users can submit solutions in several supported programming languages. Leetcode checks the solutions on correctness internally with multiple unit tests, while allowing public access only to a handful of unit tests per problem. Users usually share their solutions on other platforms such as Github, in particular https://github.com/doocs/leetcode has 15k+ stars. Most of these user-provided solutions are written in Python, but fewer in JavaScript. Hence, programmers in JavaScript may be at a disadvantage as they cannot fully benefit from the numerous Python solutions.

---

[13]https://leetcode.com/problemset/all/, https://www.linkedin.com/company/leet-code

[14]Offline accuracy / online accuracy. Offline accuracy is the test accuracy on example tests. Online accuracy is computed as 98% confidence interval from a sample of 100 programs manually submitted to LeetCode.

Table 3. DuoGlot's Performance in Case Studies

Case Study: LeetCode	Accuracy[14]	Time
DuoGlot (142 + 5 rules)	37.2% / -	-
DuoGlot (142 + 16 rules)	**51.7%** / -	-
DuoGlot (142 + 41 rules)	**61.4%** / -	-
DuoGlot (142 + 110 rules)	**75.4% / 71.6±5%**	1.75s
Codex (fixed typos)	50.0% / -	5.23s
Transcript	48.7% / -	**0.24s**
Javascripthon	0% / -	-
py2js	0% / -	-

Case Study: CtCI	Accuracy	Time
DuoGlot (142 + 34 rules)	28%	-
DuoGlot (142 + 55 rules)	**56%**	6.78s
Codex (fixed typos)	12%	37.6s
Transcript	44%	**0.25s**
Javascripthon	8%	-
py2js	0%	-

We use DuoGlot to make available the solutions in JavaScript. We take 1414 Python solutions from the aforementioned Github page, filter out 347 solutions[15], and end up with 1067 Python programs, each with a few available unit tests. For comparison reasons, we translate these programs with the other transpilers too and report the results in Table 3. JavaScripthon and py2js cannot translate any of the programs because those programs rely on the latest syntax and APIs (Python 3.9) which are not supported by these two transpilers. Transcript and Codex translate around half of the programs. DuoGlot initialized by the 142 rules for GFG benchmarks cannot translate any program, however, once we add 16 rules its accuracy quickly surpasses Transcript and Codex. By adding more rules, it achieves even higher accuracy: 61.4% with additional 41 rules, and 75.4% with 110 rules. Its average translation time is 1.75 seconds per program.

The above JavaScript translations provided by DuoGlot are based only on a few available offline unit tests (2.4 test examples per program on average). These unit tests achieve 100% line coverage for 892 of the 1067 programs and 99% line coverage averaged over all programs. To make sure the translations pass an actual LeetCode online assessment, we randomly sample 100 JavaScript programs from the 804 offline-correct translations and manually submit them to LeetCode. The site reported that 95 out of 100 passed all of the internal unit tests. This implies that with confidence of 98% [16], the online accuracy on all programs is (71.6 ± 5) %, thus much higher than even the offline accuracy of any other compared transpiler. We can speculate that the close match of online to offline accuracy occurs because the incremental paradigm results in a search space that is small enough and highly constrained by the source, hence several good examples might already be sufficient to filter out all the wrong translations in this space.

We also inspect the rules created for the LeetCode case study and code patterns that cause failures. Some representative examples of code snippets are listed in Table 4. We find that most rules are general and should be reusable in a different code translation task. A handful of the rules, while still being reusable, are specific to the code patterns in those LeetCode programs. For cases where a correct translation rule would be bespoke to the specific target program, we treat them as failures instead of adding specific one-shot rules (see Table 4 for examples). We will discuss the failures in detail in section 7.4.

6.6 Case Study: Translating Longer Code

We test the scalability of DuoGlot by translating longer programs. For this purpose, we use Python programs[17] from the book "Cracking the Coding Interview" [McDowell 2015]. We collect

---

[15]They are not self-contained, i.e. they cannot run as standalone programs.

[16]It is computed as the 98% credible interval. Assuming a uniform prior over online accuracy (possible value varying from 0 to offline accuracy), it is the posterior probability that the accuracy lies in that interval.

[17]Programs were collected from the book's official GitHub repository.

Table 4. Examples of code snippets in the LeetCode case study

Patterns for more general rules (Can be reused elsewhere)	
Python code snippets	JavaScript code snippets

<pre># Single function 	
accumulate([1,2])
accumulate(a)</pre>
	<pre>Array.from([1,2]).map((sum => value => sum += value)(0))
	Array.from(a).map((sum => value => sum += value)(0))</pre>

<pre># Function with a specific argument 	
sorted(arr, reverse=True)
sorted([1,2], reverse=True)</pre>
	<pre>arr.slice().sort((a, b) => b - a)
	[1,2].slice().sort((a, b) => b - a)</pre>

<pre># List comprehension with enumerate 	
[v*i for i, v in enumerate(nums) if v != i + 1]
[b for k, b in enumerate(f()) if True]</pre>
	<pre>[...nums].map((v, i) => [v, i]).filter(
	    ([v, i]) => v != i + 1).map(([v, i]) => v*i)
	[...f()].map((b, k) => [b, k]).filter(
	    ([b, k]) => true).map(([b, k]) => b)</pre>

<pre># (Lazy) generator expression with filter 	
(x for x in y if True)
(a+b for a,b in [t1,t2] if 1 > 0)</pre>
	<pre>({[Symbol.iterator]: function* () {
	    for (let x of y) if (true) yield x;}})
	({[Symbol.iterator]: function* () {
	    for (let [a,b] of [t1,t2]) if (1 > 0) yield a+b;}})</pre>

Patterns for more specific rules (Reuseable rules within LeetCode programs)	
Python code snippets	JavaScript code snippets

<pre># Specific to how those LeetCode programs 	
# process Regex match results (xxx.regs[0][1]) 	
x.regs[0][1]
result[y].regs[0][1]</pre>
	<pre>(x.index + x[0].length)
	(result[y].index + result[y][0].length)</pre>

<pre># Sometimes LeetCode programmer added unnecessary 	
# parentheses around the function `ord`. 	
(ord)(x)
(ord)("s")</pre>
	<pre>x.charCodeAt(0)
	"s".charCodeAt(0)</pre>

Patterns that are considered as failures (No reusable rules)	
Python code snippets	Reasons of failures

<pre>from heapq import heapify, heappush, heappop</pre>	<pre>// !!! there's no built-in support for heap in JavaScript
	// !!! A third-party library is needed</pre>

<pre>ans = ans * pow(a, e, MOD) % MOD</pre>	<pre>// !!! numbers are too large and result is inaccurate
	// in JavaScript (Different runtime behavior)
	// !!! A different implementation is needed</pre>

<pre>return '{:02x}'.format(17 * y)</pre>	<pre>// !!! format string is one leaf node (no sub structure)
	// in AST. (DSL expressiveness)
	// !!! We consider such cases as failures instead of
	// adding one-shot rules.</pre>

all programs with more than 50 lines of code, in total 25 programs, of which 19 have 54 − 100 lines and 6 have 100 − 160 lines. On average the 25 programs have 88 lines (cf. to 15 lines of LeetCode programs), and some of them have multiple functions and class definitions. The overall line coverage of the unit tests is 88%.

As before, we start DuoGlot with 142 rules obtained from the GFG benchmarks, and gradually add new rules. The final results and comparison to other transpilers are shown in Table 3. With 197 (142 + 55) rules, DuoGlot translates 14 out of the 25 programs. On the other hand, Codex is only able to translate three programs. DuoGlot needs 6.78s on average to search for a translation that passes tests, which is expected, since the longer the code, the more options DuoGlot needs to check to find the correct translation. The longest time is 25s for DuoGlot to translate a 114-line Python program. At the same time, Codex needs 37.6s on average because of the large number of tokens to decode. Besides DuoGlot and Codex, Transcrypt is able to translate 11 of the 25 programs, but the translation is only emulating Python class definitions into bloated JavaScript wrapper functions which are not native JavaScript classes thus sacrificing readability and maintainability.

Since the test coverage is 88%, we also manually check the translation of the uncovered parts of the code. It turns out that the remaining 12% are mostly functions for debug printing, overridden function stubs and occasionally unused dead code. While not affecting the functionality of the translation, DuoGlot cannot automatically fix errors in the 12% uncovered parts.

7 DISCUSSIONS

Let us take a look at application domain, comparison with compilers, and extensibility of DuoGlot.

7.1 DuoGlot Application Domain and Usage Model

Code translation in different application domains can vary drastically in terms of challenges and techniques. For use cases like porting code from Python to JavaScript, the translation requires moving across execution environments (e.g., from Python VM to node.js runtime) and so the sets of available APIs and libraries are not the same. This can be more challenging than translating within the same execution environment, such as CoffeeScript to JavaScript (within node.js), Visual Basic to C# (within .NET), and Kotlin to Java (within JVM). In the latter cases, when available APIs and runtime behaviors are not changing, the translation is mostly focused on language features rather than data types and APIs, thus compilers with a small set of rules might work well for such tasks. On the other hand, for code migration across execution environments and dependencies, existing tools are mostly ad-hoc or less accurate. Examples of transpilers from both application domains are shown in Table 5).

Table 5. Example code translation tools for different application domains

across env.	e.g., DuoGlot, TSS[18], Transcrypt, Codex, TransCoder
same env.	e.g., TypeScript / CoffeeScript (node.js), .NET Reflector (.NET), Fernflower (JVM)

The usage model of DuoGlot includes applications in line with Section 6.5, i.e. to achieve availability of JavaScript code from Python code. It can be used by developers interactively to reduce the manual effort during code translation, or it can be packed with rulesets created by developers as a push-button translator for users that are not familiar with the source language or the target language. We emphasize that the translation rules of DuoGlot need to be created once if no customization is required, and can be used in further translations. Thus the difficulty barrier of using the tool is low. For instance, DuoGlot can be used to translate programming challenges for a tutorial website. The translation rules can be created once by the developer of the website. The end-users familiar only with some languages can use DuoGlot to automatically translate solutions from other languages, unfamiliar to them. So, from this example, we can see that the rule creator and the end user do not have to be the same entity.

When the unit tests are insufficient, additional correctness oracles might be needed. In the aforementioned example, if the users are not confident about the correctness of the translation, they can submit it to the tutorial website online judge system to get feedback, and in case of reported failure, they can run DuoGlot with additional unit tests.

7.2 Comparing DuoGlot's Translation Rules With Rule-based Transpilers

Traditional transpilers [Behnel et al. 2011; Bierman et al. 2014; Burnham 2015; Cordy 2006] may differ in the used data structures and abstraction of rules (see detailed discussions in supplementary materials). However, it is common that their translation processes are driven by manually-specified state machines that analyze the code and check the applicable conditions of transformation rules.

---

[18]Tangible Software Solutions: Source Code Converters, https://www.tangiblesoftwaresolutions.com/converters.html

Such state machines are typically implemented as visitors[19] of AST trees. In contrast, DuoGlot's translation process is driven by grammar and unit test feedback rather than hand-crafted code analysis. Unlike the other transpilers whose rule compositions depend on opaque internal states of AST visitors, DuoGlot has transparent behaviors because rule compositions are not associated with any manually-specified state change. To adapt to the context and to reduce errors, DuoGlot adaptively reverts incorrect rule compositions based on the feedback from the test-driven loop, rather than using a manually-written deterministic decision procedure like compilers.

7.3 Extending to Other Languages

The current design of DuoGlot can be adapted to language pairs with common memory management, data types, and programming paradigms. For example, Python, JavaScript, Lua, and ActionScript have similarities in those three aspects, despite targeting different execution environments with different APIs. To extend DuoGlot to other pairs of languages, one needs to provide a tree-sitter grammar[20] (already available for many popular languages) and a pretty printer for the language's tree-sitter AST.

To extend to less similar language pairs, we speculate that the current DSL and purely syntax-based solutions are not sufficient. For example, for a pair of a dynamically-typed and a statically-typed language, DuoGlot may struggle to produce a type-consistent translation, or correct translations may be out of the search space if the code is JIT-unfriendly (one variable is assigned to many different types at runtime [Gong et al. 2015]). Another challenging case is translating languages with garbage collection (such as Go) to languages with manual memory management (such as C/C++). Without pointer analysis, DuoGlot may introduce memory errors.

7.4 Limitations and Future Work

Let us summarize DuoGlot's limitations and take a glance at potential future work.

Insufficient unit tests. When the unit tests provide only partial code coverage, DuoGlot with the test-based correctness oracle is not able to get test feedback on the uncovered code lines to repair errors. Companies such as Google provide sufficient unit tests [Ivanković et al. 2019] (80% ~90% on average), however, this is not always the case for open-source projects. While some popular Python projects have 90% coverage [Zhai et al. 2019], for other codebases it is more difficult to write unit tests (e.g., for user interface code) [Fard and Mesbah 2017]. To tackle the problem of insufficient unit tests, one may use automated test generators to increase the coverage. An alternative is to use static analyzers and formal verification tools as correctness oracles to provide additional feedback to DuoGlot, however, for dynamically-typed languages, such techniques might be limited, especially when the input specifications are not available. In the worst case, the user might need to review the uncovered part of the translation and manually choose the correct rules.

Differences in execution environments. Execution environments may exercise different behavior on similar types. E.g. when iterating over a dictionary, Python virtual machine may use insertion order, but the order of JavaScript's node.js runtime is not the same [21]. So, if Python code depends on the iteration order, it cannot be translated straightforward. The solution adopted by Transcrypt is to emulate Python's data types in JavaScript, but it may not be suitable for code migration due to non-idiomatic code and performance penalties.

Expressiveness of DSL and human effort. The rule DSL of DuoGlot, which directly maps to tree transducer, cannot match on sub-tokens inside leaf AST nodes. For example, it cannot translate

---

[19]https://en.wikipedia.org/wiki/Visitor_pattern

[20]Some example tree-sitter grammars: tree-sitter-python and tree-sitter-javascript. The tree-sitter grammar might need some adjustments if external scanners are used.

[21]https://stackoverflow.com/questions/5525795/does-javascript-guarantee-object-property-order

`"score: %d"` into `"score: {}"` in a general way because `"%d"` is inside the leaf node. It also cannot perform complex computations such as reordering keyword arguments in a function call based on the function's definition. A temporary solution without modifying the system is to use one-shot rules for those corner cases, but such a trick will not reduce human effort and will result in many non-reusable rules. A more expressive rule DSL might support more complex transformations but rule inference might also become more challenging. We leave as an open problem the design of better translation DSL with rule inference that may further minimize overall human effort.

Scalability. When the code is longer, as shown in the CtCI case study from Section 6.6, all Python to JavaScript transpilers have lower accuracy due to accumulated translation errors and corner cases. Specifically, for DuoGlot half of the failed translations from CtCI contain corner cases that cannot be automatically fixed, while for the correct translations the number of retries increases (e.g, up to 25 seconds and 52 retries for 114 lines of code). So, longer code might need to be split into sub-tasks for DuoGlot to handle it efficiently. In some extreme cases, when the running time of the program is high (e.g. so-called `pytest` unit testing framework), despite the small number of retries, DuoGlot would require a larger amount of time to run. Better divide-and-conquer strategy, smarter rule selection, or faster correctness checks may increase DuoGlot's scalability and all of these constitute compelling future work.

8 RELATED WORK

There are a few broad approaches used for code translations that we summarize below.

Statistical and neural machine learning based translation. Various statistical machine translation techniques have been proposed. [Karaivanov et al. 2014], [Nguyen et al. 2013] and [Nguyen et al. 2015] use phrased-based statistical translation. [Graehl et al. 2008] proposed a training algorithm for tree transducers. [Nguyen et al. 2016] use statistical methods to find mappings between APIs. Recently, neural-based translators have been gaining in popularity. [Xinyun et al. 2018] use a tree-to-tree neural network to translate JavaScript to CoffeeScript. [Roziere et al. 2020] proposed TransCoder that learns to translate code using unsupervised learning. Besides the specialized translators, large language models (such as Codex [Chen et al. 2021]) are trained for many code-related tasks, including translation. However, all of the statistical methods require a large amount of data to train. In our approach, we decouple the syntactic and semantic aspects of the translation. This allows us to learn customized syntactic patterns from a few short code snippets and search for semantically correct translations on the fly.

Term-rewriting systems. Conventionally, source code translation can be accomplished with manually written rule-based term rewriting systems and pattern matching. StringTemplate[22] is a popular framework for rule-based code generation and translation. TXL [Cordy 2006] is a general-purpose source transformation language based on union grammar techniques and manually-written transformation rules. SrcML [Collard et al. 2013] is another system for lightweight source transformation within the same language. Some computer algebra systems (such as Mathematica [Wellin et al. 2005]) have expressive pattern language to transform mathematical expressions between alphabets or to export to LaTeX. All these transformation tools require experts to manually write the rules, and syntactic or semantic correctness is typically challenging when translating between languages with different grammar and semantics.

Program Synthesis. An alternative approach is to use program synthesis to solve code translation and transformation problems. [Kamil et al. 2016] proposed verified lifting to synthesize a semantically equivalent parallel code from a piece of low-level sequential code based on SAT/SMT solving.

---

[22]https://www.stringtemplate.org/index.html

[Mariano et al. 2022] propose a neural-guided program synthesizer with similar functionalities using a cognate grammar network. However, only short programs can be transformed this way due to scalability issues in typical program synthesizers. [Miltner et al. 2019; Rolim et al. 2017] synthesize short code-transformation programs doing automated code refactoring for several statements within the same language, which is orthogonal to our work as we translate complete programs across different languages.

9 CONCLUSION

We propose a new design for transpilation with a focus on democratizing the process of building and using such transpilers. With our design, the end user would be able to build custom transpilers that are tuned to their code style, different APIs and preferred language features with little manual effort. The key design principle is to incrementally learn the rules that are only required to translate the program at hand from user-provided code snippets. Through the snippets, the users may fully customize the translation rules and tailor such rules to their needs. The manual effort is also limited to specifying a few short code snippets only once per rule that can then be reused in the future. Internally, the design formalizes the translation problem as a search in a program space structured by non-deterministic tree transducers, customized by transduction rules, and constrained by grammar and unit tests.

We have also provided a practical instantiation of our design called DuoGlot that translates from Python to JavaScript. DuoGlot learns 142 rules from 360 lines of snippets to reach 90% accuracy on GeeksForGeeks benchmarks, which is of 5297 lines in total (14.7× of the code snippets). As a comparison, none of the other transpilers achieves an accuracy higher than 76%. Similarly, none provides flexible customizability or non-expert upgradability of translation rules. DuoGlot can be fairly easily adapted to translate between other untyped scripting languages.

10 DATA AVAILABILITY STATEMENT

The artifact containing the code and the related benchmarks for DuoGlot is permanently available on Zenodo [Wang et al. 2023]. The latest version of the artifact and supplementary materials can be found on Github [Wang 2023].

ACKNOWLEDGMENTS

REFERENCES

Ghazi Alkhatib. 1992. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice* 4, 2 (1992), 83–104. https://doi.org/10.1002/smr.4360040203

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Practical fault localization for dynamic web applications. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 265–274. https://doi.org/10.1145/1806799.1806840

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46. https://doi.org/10.1007/978-3-319-73721-8_2

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan,

Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

Max Brunsfeld. 2018. *Atom understands your code better than ever before.* https://github.blog/2018-10-31-atoms-new-parsing-system/

Trevor Burnham. 2015. *Coffeescript: accelerated Javascript development.* Pragmatic Bookshelf. https://pragprog.com/titles/tbcoffee2/coffeescript/

Raymond PL Buse and Westley R Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis.* 121–130. https://doi.org/10.1145/1390630.1390647

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). https://doi.org/10.48550/arXiv.2107.03374

Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance.* IEEE, 516–519. https://doi.org/10.1109/ICSM.2013.85

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree automata techniques and applications. https://hal.inria.fr/hal-03367725

James R Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210.

Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (un) covered parts. In *2017 IEEE international conference on software testing, verification and validation (ICST).* IEEE, 230–240. https://doi.org/10.1109/ICST.2017.28

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 111–122. https://doi.org/10.1145/982962.964011

Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: pinpointing JIT-unfriendly JavaScript code. In *ESEC/FSE (ESEC/FSE 2015).* Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2786805.2786831

Google. 2021. *TensorFlow 1.x vs TensorFlow 2 - Behaviors and APIs.* https://www.tensorflow.org/guide/migrate/tf1_vs_tf2

Jonathan Graehl, Kevin Knight, and Jonathan May. 2008. Training tree transducers. *Computational Linguistics* 34, 3 (2008), 391–427. https://doi.org/10.1162/coli.2008.07-051-R2-03-57

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Anna Irrera. 2017. *Banks scramble to fix old systems as IT 'cowboys' ride into sunset.* https://www.reuters.com/article/us-usa-banks-cobol-idUSKBN17C0D8

Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 955–963. https://doi.org/10.1145/3338906.3340459

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726. https://doi.org/10.1145/2908080.2908117

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* 173–184. https://doi.org/10.1145/2661136.2661148

Cody Koeninger. 2020. *Cloudflare Workers Announces Broad Language Support.* https://blog.cloudflare.com/cloudflare-workers-announces-broad-language-support/

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 71 (apr 2022), 27 pages. https://doi.org/10.1145/3527315

Gayle Laakmann McDowell. 2015. *Cracking the coding interview: 189 programming questions and solutions.* CareerCup, LLC.

MetaResearch. 2022. *GeeksForGeeks benchmark.* https://github.com/facebookresearch/CodeGen/tree/659963b195f5edaa42502d97605386b7fecbaa62/data/transcoder_evaluation_gfg

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360569

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 651–654. https://doi.org/

10.1145/2491411.2494584

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596. https://doi.org/10.1109/ASE.2015.74

Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758. https://doi.org/10.1145/2889160.2892661

Frolin S Ocariza Jr, Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. 2016. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability* 26, 1 (2016), 69–88. https://doi.org/10.1002/stvr.1576

Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)*. 21–24. https://doi.org/10.1109/BLOCKS.2017.8120404

Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Vol. 50. 107–126. https://doi.org/10.1145/2858965.2814310

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415. https://doi.org/10.1109/ICSE.2017.44

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611. https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. *arXiv preprint arXiv:2110.06773* (2021). https://doi.org/10.48550/arXiv.2110.06773

Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 537–549. https://doi.org/10.1145/3433210.3437528

Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

Andrey A Terekhov and Chris Verhoef. 2000. The realities of language conversions. *IEEE Software* 17, 6 (2000), 111–124. https://doi.org/10.1109/52.895180

Guido van Rossum. 2009. *What's New In Python 3.0*. https://docs.python.org/release/3.0.1/whatsnew/3.0.html

Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. 2020. PEP 617 – New PEG parser for CPython | peps.python.org. https://peps.python.org/pep-0617/

Bo Wang. 2023. *DuoGlot: A User-Customizable Code Translator*. https://github.com/HALOCORE/DuoGlot

Bo Wang, Aashish Kolluri, Ivica Nikolić, Teodora Baluta, and Prateek Saxena. 2023. *DuoGlot: User-Customizable Transpilation of Scripting Languages (Artifact)*. https://doi.org/10.5281/zenodo.7709003 This artifact is also available on Github: https://github.com/HALOCORE/DuoGlot.

Gerald M Weinberg. 1971. *The psychology of computer programming*. Vol. 29. Van Nostrand Reinhold New York.

Paul R Wellin, Richard J Gaylord, and Samuel N Kamin. 2005. *An introduction to programming with Mathematica®*. Cambridge University Press. https://doi.org/10.1017/CBO9780511801303

Chen Xinyun, Liu Chang, Song Dawn, et al. 2018. Tree-to-tree neural networks for program translation. *NeurIPS* (2018). https://proceedings.neurips.cc/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf

Hongyu Zhai, Casey Casalnuovo, and Prem Devanbu. 2019. Test coverage in python programs. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 116–120. https://doi.org/10.1109/MSR.2019.00027