Automated Transpilation of Imperative to Functional Code using Neural-Guided Program Synthesis

BENJAMIN MARIANO, University of Texas at Austin, USA
YANJU CHEN, University of California, Santa Barbara, USA
YU FENG, University of California, Santa Barbara, USA
GREG DURRETT, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA

While many mainstream languages such as Java, Python, and C# increasingly incorporate functional APIs to simplify programming and improve parallelization/performance, there are no effective techniques that can be used to *automatically* translate existing imperative code to functional variants using these APIs. Motivated by this problem, this paper presents a transpilation approach based on inductive program synthesis for modernizing existing code. Our method is based on the observation that the overwhelming majority of source/target programs in this setting satisfy an assumption that we call *trace-compatibility*: not only do the programs share syntactically identical low-level expressions, but these expressions also take the same values in corresponding execution traces. Our method leverages this observation to design a new neural-guided synthesis algorithm that (1) uses a novel neural architecture called *cognate grammar network (CGN)* and (2) leverages a form of concolic execution to prune partial programs based on *intermediate values* that arise during a computation. We have implemented our approach in a tool called NGST2 and use it to translate imperative Java and Python code to functional variants that use the `Stream` and `functools` APIs respectively. Our experiments show that NGST2 significantly outperforms several baselines and that our proposed neural architecture and pruning techniques are vital for achieving good results.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Search-based software engineering**.

Additional Key Words and Phrases: transpilation, program synthesis, neural networks

ACM Reference Format:
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 (April 2022), 27 pages. https://doi.org/10.1145/3527315

1 INTRODUCTION

In recent years, a number of mainstream programming languages have introduced functional APIs to offer users the benefits of functional programming within imperative languages. For instance, Java 8 introduced the notion of streams, which provides an API for processing sequences of elements using common functional operators like map and filter. Similarly, Python offers an expansive functional API incorporating functional operators like map, reduce, and list comprehensions.

Authors' addresses: Benjamin Mariano, bmariano@cs.utexas.edu, University of Texas at Austin, USA; Yanju Chen, yanju@cs.ucsb.edu, University of California, Santa Barbara, USA; Yu Feng, yufeng@cs.ucsb.edu, University of California, Santa Barbara, USA; Greg Durrett, gdurrett@cs.utexas.edu, University of Texas at Austin, USA; Işil Dillig, isil@cs.utexas.edu, University of Texas at Austin, USA.

2475-1421/2022/4-ART71
https://doi.org/10.1145/3527315

While these functional APIs provide a number of benefits such as convenient parallelization [Khatchadourian et al. 2020], succinct definitions, and easier readability, converting imperative code to functional APIs can be challenging, particularly for developers accustomed to imperative languages. Recognizing this drawback, prior efforts introduced rule-based translators to automatically refactor imperative code into (equivalent) functional versions [Gyori et al. 2013], and such approaches have even been incorporated into mainstream IDEs like NetBeans. However, since manually crafting rule-based translators for *arbitrary* imperative code is infeasible, existing approaches can only handle stylized code snippets over commonly occurring imperative code patterns.

In this paper, we propose an alternative solution to this transpilation problem based on *counterexample guided inductive synthesis (CEGIS)* [Solar-Lezama et al. 2007]. Our approach incorporates a novel inductive synthesis engine that combines neural networks (for guiding search) with powerful bidirectional pruning rules based on concolic execution. At a high level, our solution is based on two key observations:

- While the high-level structure of the code is quite different between the imperative and functional versions, the source and target programs actually share many syntactically identical low-level expressions (e.g., arguments of API calls). We formalize this observation using a new notion that we call *cognate grammars* and leverage it to reduce the size of the search space.
- Beyond being syntactically identical, the source and target programs enjoy a property that we call *trace-compatibility*: shared expressions in the two programs always take the same values when executing the source and target programs on the same inputs.

Guided by these observations, we have developed a new neural-guided inductive synthesizer called NGST2[1] (see Figure 1) that (1) incorporates a new neural architecture called a *cognate grammar network (CGN)* to guide the search, and (2) a powerful pruning system that significantly reduces the search space using the trace compatibility observation. Our proposed CGN model is an adaptation of the existing *abstract syntax network (ASN)* [Rabinovich et al. 2017] (which is a top-down, tree-structured model that is suitable for reasoning about programs), but it leverages the syntactically shared terms between the source and target programs to make more accurate predictions. We train the CGN model off-line on pairs of equivalent (but synthetic) imperative and functional programs and then use it on-line at synthesis time to guide a top-down enumerative search engine. Given a partial abstract syntax tree (AST) where some of the nodes are non-terminal symbols in the target grammar, our approach uses the CGN to predict which grammar productions to use for expanding the non-terminals. This results in a refined AST which is then checked under the trace compatibility assumption. In particular, the goal of this check is to determine whether a given partial program produces any intermediate values that are inconsistent with the source program. If so, we can prune all completions of the partial program from the search space. In contrast to prior pruning approaches used in inductive program synthesis [Albarghouthi et al. 2013; Alur et al. 2015; Feser et al. 2015], the key novelty of our approach is to leverage *intermediate* values (as opposed to just input-output examples) to perform more aggressive pruning.

We evaluate NGST2 in two different settings that require translating imperative code snippets to equivalent functional variants. In the first setting, we use NGST2 to migrate imperative Java programs to the functional Java Stream API. In our second client, we use NGST2 to translate imperative Python code to a functional style using the `functools` API and other Pythonic constructs, such as list comprehensions. Overall, NGST2 is able to automate 80% of these challenging source-to-source translation tasks and significantly outperforms existing techniques and simpler baselines/ablations. These experimental results demonstrate the effectiveness of our trace compatibility checking rules

[1]Stands for *Neural Guided Source To Source Translation*

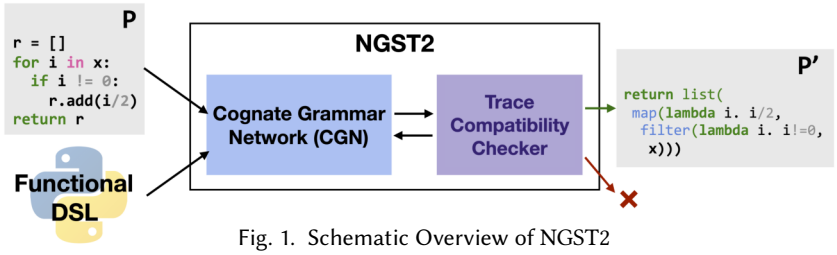


Fig. 1. Schematic Overview of NGST2

```
1   public List<String> getUserRoles(String uID, List<Policy> policies) {
2       List<String> roles = new ArrayList<>();
3       for (Policy policy : policies) {
4           for (Role role : policy.getRoles()) {
5               if (role.getIDs().contains(uID))
6                   roles.add(role.getName());
7           }
8       }
9       return roles;
10  }
```

Fig. 2. Imperative Java Program

as well as the proposed neural CGN architecture and provide evidence that this work advances the state-of-the-art in imperative-to-functional code translation.

In summary, this paper makes the following key contributions:

(1) We propose the first broadly applicable solution for translating imperative code to equivalent versions using functional APIs.
(2) We identify and formalize two key characteristics of this problem (namely, *cognate grammars* and *trace compatibility*) and propose a new neural-guided inductive synthesizer that leverages these properties.
(3) We propose a new neural architecture called *cognate grammar network (CGN)* that can be used for source-to-source translation tasks involving cognate grammars with shared terms.
(4) We propose a novel pruning technique — based on the trace compatibility assumption — for detecting partial programs that produce intermediate values that are inconsistent with the source program. Our pruning technique is based on concolic execution and leverages bidirectional reasoning to reduce the overhead of trace compatibility checking.
(5) We implement our algorithm in a tool called NGST2 and evaluate it on real-world programs from two different domains and show that it significantly outperforms existing approaches.

2 OVERVIEW

We give a high-level overview of our method with the aid of the motivating example shown in Figure 2. Here, the `getUserRoles` procedure takes as input a user ID (`uID`) and a list of policies and returns all the "roles" (e.g., beneficiary, policy owner, etc.) that the specified user has across all policies. Figure 2 performs this computation in an imperative style using a doubly nested loop, but it is possible to express the same program logic in a functional style using the Java Stream API. As shown in Figure 3, the functional implementation first converts the input policy to a stream

```
1  public List<String> getUserRoles(String uID, List<Policy> policies) {
2    return policies.stream()
3              .flatMap(policy -> policy.getRoles().stream())
4              .filter(role -> role.getIDs().contains(uID))
5              .map(role -> role.getName())
6              .collect(Collectors.toList());
7  }
```

Fig. 3. Java Stream Program

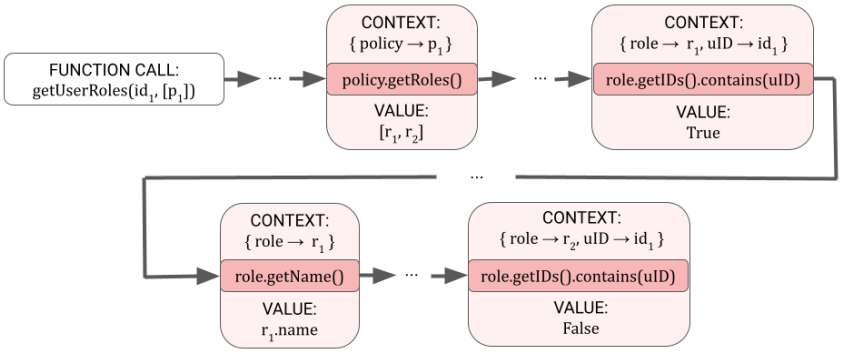


Fig. 4. Execution of Imperative Program through Shared Expressions

and then obtains the desired result using the higher-order combinators `map`, `filter`, and `flatMap`[2] provided by the Stream API.

While the high-level structure of the code is clearly quite different between Figures 2 and 3, we notice that many of the low-level expressions (highlighted in pink) are actually shared between the two programs. In fact, on deeper inspection, we realize that the relationship between these expressions goes beyond just surface syntax. As shown in the sample execution traces in Figures 4 and 5, these shared expressions also take the same set of values in two corresponding executions of the imperative and functional programs.

As illustrated by this example, code snippets written using functional APIs exhibit close syntactic *and* semantic ties to their corresponding imperative version. Our method takes advantage of such relationships between the two programs to dramatically simplify the underlying synthesis task. In particular, we formalize the syntactic relationship between the source and target programs through the concept of *cognate grammars* and express their semantic ties using the notion of *trace compatibility*.

2.1 Leveraging Syntactic Ties via CGN

Intuitively, the syntactic relationship between the two program versions is extremely useful for restricting the search space that a program synthesizer needs to explore. In particular, once the synthesizer produces a "sketch" of the functional program with a certain choice of functional combinators, the arguments of those combinators are not *arbitrary* expressions but rather small snippets taken from the original imperative program. Thus, this observation immediately gives us a way to reduce the search space.

[2] `flatMap` maps each element of the input to a stream and then flattens the result to a single stream.

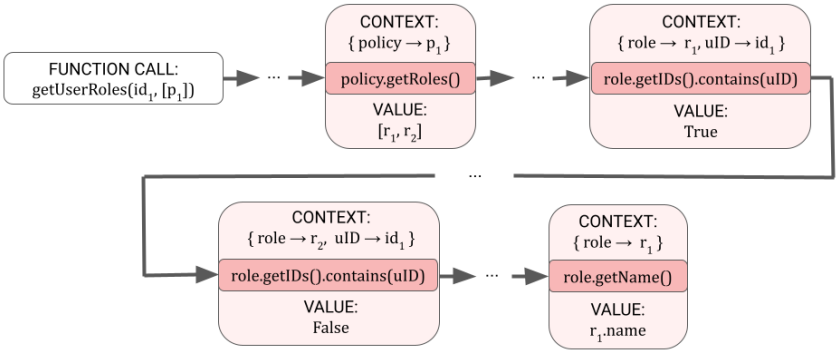


Fig. 5. Execution of Functional Program through Shared Expressions

```
return ?0.flatMap(policy -> ?1)
       .map(role -> ?2)
```

Fig. 6. Partial program encountered during search.

In addition, our method also uses the syntactic relationship between the two programs to more effectively *guide* search. In particular, similar to many prior techniques [Balog et al. 2016; Ye et al. 2020], we want to leverage a neural model to predict more likely target programs based on the source program. However, existing techniques for neural-guided synthesis do not exploit the syntactic similarity between source and target expressions and predict *grammar productions* instead of *concrete expressions* from the source program. In contrast, we need a method that can predict grammar productions when generating the high-level structure of the program but that *switches to predicting source expression when filling in the remaining holes.*

To address this problem, we formulate the source and target languages as a pair of *cognate grammars* where some of the non-terminals are *shared* between the two languages. Then, based on this concept, we propose a new neural architecture called *cognate grammar network (CGN)* to perform predictions over such grammars. Our proposed architecture extends the existing *abstract syntax network* model [Rabinovich et al. 2017] (for generating syntactically valid programs) with a so-called *pointer network* [Jia and Liang 2016; See et al. 2017] in order to make useful predictions when expanding shared non-terminals.

2.2 Leveraging Semantic Ties

While our proposed CGN architecture makes it possible to exploit syntactic similarities between the source and target programs, we also want to exploit the *semantic ties* between them. To see what we mean by this, consider the partial program shown in Figure 6. Here, $?_0$, $?_1$, and $?_2$ represent unknown expressions, and the fact that $?_1$ and $?_2$ are highlighted in pink indicates that they must be instantiated with one of the highlighted expressions from the source program in Figure 2. Even though $?_0$ is completely unconstrained (i.e., it can be any arbitrary expression), we can use our trace compatibility assumption to prune this partial program using the following chain of inferences for the program executions pictured in Figures 4 and 5:

- Among the four source expressions that $?_1$ could be instantiated with, the only one that would even type check is `policy.getRoles()`, which only takes on value $[r_1, r_2]$ in the execution trace shown in Figure 4.

- Thus, given the semantics of `flatMap`, we can conclude that the expression $?_0$.`flatMap(...)` must produce a list which contains $[r_1, r_2]$ as a sublist when executed on the same input as the imperative program.
- Now, reasoning backwards from the return value, we know that the output for the partial program on this trace needs to be the list $[r_1]$.
- But, no matter how we instantiate $?_2$, we can never obtain the list $[r_1]$ when invoking `map` on an input list containing $[r_1, r_2]$ as a sublist (as `map` is length preserving). Thus, we conclude that no completion of this partial program can be equivalent to the imperative source program from Figure 2.

Note that this kind of reasoning would *not* be possible without knowing intermediate values in the execution of the source program. In particular, if we did not know the possible values that $?_1$ could take, we would have to assume that `map` could be called on arbitrary list, which does not provide any pruning opportunities. Our synthesis engine leverages such shared intermediate values between the two programs and prunes infeasible partial programs using a form of concolic execution.

3 PRELIMINARIES AND PROBLEM STATEMENT

In this section, we first present some preliminary information and then formalize our problem.

3.1 CFGs and Partial Programs

As standard, we define the syntax of our source and target languages in terms of a context-free grammar (CFG):

Definition 3.1 (CFG). A context-free grammar $\mathcal{G}$ is a tuple (V, Σ, R, S) where V are non-terminals, Σ are terminals, R are productions, and S is the start symbol.

Given a string $s \in (\Sigma \cup V)^*$, we use the notation $s \Rightarrow s'$ to denote that s' is obtained from s by replacing one of the non-terminals N in s by a string $w \in (\Sigma \cup V)^*$ such that $N \rightarrow w$ is a production in the grammar. We use the notation $\Rightarrow^*$ to denote the transitive closure of $\Rightarrow$.

Definition 3.2 (Partial program). Let $\mathcal{G} = (V, \Sigma, R, S)$ be a context-free grammar for some programming language $\mathcal{L}$. A *partial program* in $\mathcal{L}$ is a string $\mathcal{P} \in (\Sigma \cup V)^*$ such that $S \Rightarrow^* \mathcal{P}$. We say that $\mathcal{P}$ is *complete* if $\mathcal{P}$ does not contain any non-terminals. Finally, a complete program $\mathcal{P}'$ is a *completion* of partial program $\mathcal{P}$ if $\mathcal{P} \Rightarrow^* \mathcal{P}'$.

3.2 Program Equivalence and Transpilation

The problem we consider in this paper is an instance of *transpilation*, where the goal is to translate a program $\mathcal{P}_1$ in a programming language $\mathcal{L}_1$ to a *semantically equivalent* program $\mathcal{P}_2$ in a different language $\mathcal{L}_2$. In this work, we presume program semantics are given in terms of denotational semantics $[\![\cdot]\!]_\sigma : \Sigma^* \rightarrow \mathcal{V}$ which maps a complete program $\mathcal{P} \in \Sigma^*$ to a value $v \in \mathcal{V}$ given valuation $\sigma : Var \rightarrow \mathcal{V}$ mapping variables in $\mathcal{P}$ to values.

Definition 3.3 (Equivalence). Two programs $\mathcal{P}_1, \mathcal{P}_2$ are *semantically equivalent*, denoted $\mathcal{P}_1 \equiv \mathcal{P}_2$, if, for all input valuations σ, we have $[\![\mathcal{P}_1]\!]_\sigma = [\![\mathcal{P}_2]\!]_\sigma$.[3]

In other words, we consider two programs to be equivalent if they always produce the same output when executed on the same input.

[3]Since we out technique does not target reactive programs, we assume all source programs terminate, so our equivalence definition is for terminating programs.

Definition 3.4 (Transpilation problem). Let $\mathcal{P}_s$ be a program in some source language $\mathcal{L}_s$ defined by grammar $\mathcal{G}_s$. Given a target language $\mathcal{L}_t$ defined by grammar $\mathcal{G}_t$, the transpilation problem is to find a (complete) program $\mathcal{P}_t$ in $\mathcal{L}_t$ such that $\mathcal{P}_s \equiv \mathcal{P}_t$.

3.3 Trace Compatible Transpilation

As discussed in Section 2, the transpilation problem as introduced in Definition 3.4 is too general to be amenable to an efficient solution (at least without placing severe syntactic restriction on the source or target languages). Motivated by a property we have observed in our target application domain (namely, translating imperative code to functional APIs), we first introduce the *trace compatibility* assumption and then refine the transpilation problem under this restriction.

First, since the notion of trace compatibility only makes sense for certain combinations of languages, we introduce the concept of *cognate grammars*:

Definition 3.5 (Cognate grammars). Grammars $\mathcal{G}_1 = (V_1, \Sigma_1, R_1, S_1)$ and $\mathcal{G}_2 = (V_2, \Sigma_2, R_2, S_2)$ are *cognate* iff, for every *shared non-terminal* $N \in V_1 \cap V_2$, we have $N \Rightarrow^* \tau$ in $\mathcal{G}_1$ iff $N \Rightarrow^* \tau$ in $\mathcal{G}_2$. We refer to the words $\tau \in (\Sigma_1 \cap \Sigma_2)^*$ that are derivable from the shared non-terminals as *shared terms* between $\mathcal{G}_1$ and $\mathcal{G}_2$.

In other words, two grammars are said to be *cognate* if every shared non-terminal symbol $N \in V_1 \cap V_2$ can derive the same words from N in both grammars. As mentioned in Section 1, this notion of cognate grammars makes sense in the context of translating imperative code to functional APIs because, while the high-level structure of the source and target programs may differ, the low-level expressions often tend to be shared. Thus, we can structure the grammar of the source and target languages to be cognate — i.e., low-level expressions of the same type are derived from shared non-terminal symbols, whereas high-level constructs (e.g., `for` loops, higher-order combinators, etc.) are derived from unshared non-terminals.

Beyond having cognate source and target languages, another important observation in our setting is that shared terms almost always take the same values in corresponding pairs of executions. To formalize this observation which we refer to as *trace compatibility*, we first assume a collecting semantics [Cousot and Cousot 1994; Hudak and Young 1991] $C[\![\cdot]\!]_{\mathcal{P},\sigma} : Expr \to \wp(\mathcal{V})$ which maps an expression e to the set of values which e takes during execution of $\mathcal{P}$ on σ.

Example 3.1. Consider the following program $\mathcal{P}$ that takes as input an integer n:

```
for(i:=0; i<n; i+=2) {
    print(i+1);
}
```

Here, for input valuation $\sigma : [n \mapsto 3]$, we have $C[\![i+1]\!]_{\mathcal{P},\sigma} = \{1, 3\}$ since these are the values that $i+1$ takes during this execution.

Next, we define *trace compatibility* as follows:

Definition 3.6 (Trace compatibility). Let $\mathcal{P}_s$ and $\mathcal{P}_t$ be a pair of source and target programs from cognate grammars $\mathcal{G}_s$ and $\mathcal{G}_t$. We say that $\mathcal{P}_s$ and $\mathcal{P}_t$ are *trace-compatible* on σ, denoted $\sigma \vdash \mathcal{P}_s \preceq \mathcal{P}_t$ iff for every shared term τ appearing (at least once) in $\mathcal{P}_t$ we have (1) τ appears (at least once) in $\mathcal{P}_s$, and (2) $C[\![\tau]\!]_{\mathcal{P}_t,\sigma} \subseteq C[\![\tau]\!]_{\mathcal{P}_s,\sigma}$.

In other words, trace compatibility of $\mathcal{P}_s$ and $\mathcal{P}_t$ on input σ means that (a) every shared term τ that appears in $\mathcal{P}_t$ also syntactically appears in the source program $\mathcal{P}_s$, and (b) for every value that τ can take when executing $\mathcal{P}_t$ on σ, there is a corresponding value of τ when executing $\mathcal{P}_s$ on σ.

```
1:  procedure TRANSPILE(P, M_θ, G)
2:      input: Source program P
3:      input: Neural model M_θ
4:      input: Context-free grammar G = (V, Σ, R, S)
5:      W ← {S}                          ▷ initialize worklist to S, the empty partial program
6:      E ← ∅                            ▷ initialize counterexamples to empty set
7:      while W ≠ ∅ do
8:          P' ← CHOOSEBEST(W, M_θ, P)   ▷ Dequeue top candidate from priority queue
9:          if IsComplete(P') then
10:             (ν, σ) ← IsEquivalent(P, P', E)
11:             if ν then                ▷ Candidate is equivalent
12:                 return P'
13:             E ← E ∪ {σ}              ▷ Add new counterexample
14:             continue
15:         if ¬ ISFEASIBLE(P', P, E) then   ▷ Check feasibility of partial program
16:             continue
17:         N ← ChooseNonterminal(P')
18:         for r ∈ Productions(N) do
19:             W ← W ∪ {Expand(P', r)}  ▷ Add expansions of P' to worklist
20:     return ⊥
```

Fig. 7. Top-level transpilation Algorithm

Note that if there are multiple occurrences of τ in some program $\mathcal{P}$ (i.e., for either $\mathcal{P}_s$ or $\mathcal{P}_t$), then $C[\![\tau]\!]_{\mathcal{P},\sigma}$ contains the values that *all* occurrences of τ take during execution of $\mathcal{P}$ on σ.

Example 3.2. Consider the following imperative program $\mathcal{P}_s$, which prints every odd element of input list x multiplied by 2, and the functional program $\mathcal{P}_t$ which prints *every* element multiplied by 2. We assume the term $i * 2$ (highlighted in pink) is a shared term in the grammars of $\mathcal{P}_s$ and $\mathcal{P}_t$.

```
for(int i : odd(x)) {
    print( i*2 );
}
```

```
map(x, λ i. print( i*2 ))
```

These two programs are clearly not equivalent, and they also violate trace compatibility on the input $\sigma = \{x \mapsto [2]\}$. In particular, $i * 2$ produces value 4 in $\mathcal{P}_t$ but not $\mathcal{P}_s$.

Using this formalization of trace compatibility, we next define the *trace compatible transpilation* problem that we address in the rest of this paper:

Definition 3.7. (Trace compatible transpilation) Let $\mathcal{L}_s$ and $\mathcal{L}_t$ be a pair of source and target languages defined by two cognate context-free grammars. Given a program $\mathcal{P}_s$ in $\mathcal{L}_s$, the *trace compatible transpilation problem* is to find a program $\mathcal{P}_t$ in $\mathcal{L}_t$ such that (1) $\mathcal{P}_s \equiv \mathcal{P}_t$, and (2) for all input valuations σ, we have $\sigma \vdash \mathcal{P}_s \backsimeq \mathcal{P}_t$.

4 NEURAL-GUIDED TRANSPILATION ALGORITHM

In this section, we discuss our transpilation algorithm based on neural-guided inductive program synthesis. We first describe our high-level approach and then explain the key pruning that exploits the trace-compatibility assumption.

```
1: procedure IsFeasible(P, P', E)
2:     input: Source program P
3:     input: Candidate program P'
4:     input: Counterexamples E
5:     for σ ∈ E do
6:         if σ ⊢ P' ≠ P then
7:             return false
8:     return true
```

Fig. 8. Algorithm for checking trace compatibility

4.1 Top-Level Algorithm

Figure 7 summarizes our transpilation technique based on inductive program synthesis. The Transpile procedure takes as input (i) the source program $\mathcal{P}$, (ii) a trained neural model $\mathcal{M}_\theta$ (described in the next section), and (iii) a context-free grammar describing the target language. The output of Transpile is either a program $\mathcal{P}'$ that is equivalent to the source program $\mathcal{P}$ or $\perp$ if no equivalent program is found. At a high-level, the Transpile algorithm is an instantiation of the counterexample-guided inductive synthesis paradigm but (1) uses a new neural architecture to guide its search, and (2) prunes the search space by using program analysis techniques that exploit the trace-compatibility assumption.

Internally, Transpile maintains a worklist $\mathcal{W}$ of partial programs and, as standard in CEGIS, a set of counterexamples $\mathcal{E}$. After initializing the worklist to an empty partial program (line 5), the algorithm enters a loop in which it dequeues a partial program from $\mathcal{W}$ and expands it using one of the productions in $\mathcal{G}$. Specifically, at line 8, it invokes a procedure called ChooseBest that returns the "best" partial program according to the neural model $\mathcal{M}_\theta$ as follows:

$$\arg\max_{\mathcal{P}' \in \mathcal{W}} \mathcal{M}_\theta(\mathcal{P}' \mid \mathcal{P}) \tag{1}$$

If the chosen program $\mathcal{P}'$ is complete (i.e., it has no non-terminals), then Transpile invokes an equivalence checking engine to test whether $\mathcal{P}$ and $\mathcal{P}'$ are equivalent (line 10) and returns $\mathcal{P}'$ if they are (line 12). In particular, the procedure IsEquivalent takes as input two programs and the counterexample set and returns a tuple (ν, σ) where ν is a boolean indicating whether the two programs are equivalent and σ is a counterexample if they are not. Since our approach is based on CEGIS, the returned counterexample σ is added to counterexample set $\mathcal{E}$ and the algorithm moves on to the next partial program in the worklist (lines 13-14). Note that IsEquivalent first checks equality on the examples in the counterexample set before invoking a stronger verifier – if one of these checks fails, the returned counterexample σ is simply the counterexample from $\mathcal{E}$ which caused the failed check.

On the other hand, if $\mathcal{P}'$ is a partial program with remaining non-terminals, our algorithm checks whether $\mathcal{P}'$ is feasible under the trace compatibility assumption. In particular, the procedure IsFeasible (discussed in detail in the next subsection) takes as input $\mathcal{P}, \mathcal{P}'$ as well as the counterexamples $\mathcal{E}$ and checks whether they are trace-compatible on inputs $\mathcal{E}$ (line 15). If they are not, the algorithm prunes the search space by not considering expansions of $\mathcal{P}'$. Otherwise, it chooses a non-terminal N used in $\mathcal{P}'$ and adds all expansions of $\mathcal{P}'$ obtained by replacing N with some $w \in (\Sigma \cup V)^*$ (for $N \rightarrow w \in R$) in $\mathcal{P}'$ to the worklist.

4.2 Checking Trace Compatibility

As illustrated by the discussion in Section 4.1, one of the novel aspects of our approach is its ability to rule out partial programs based on the trace-compatibility assumption. In this subsection, we discuss the IsFeasible procedure, summarized in Figure 8, for checking whether a partial program $\mathcal{P}'$ has any completion that is trace-compatible with $\mathcal{P}$ on inputs $\mathcal{E}$.

In more detail, IsFeasible iterates over all inputs $\sigma \in \mathcal{E}$ and checks whether the partial program $\mathcal{P}'$ satisfies the trace compatibility judgment $\sigma \vdash \mathcal{P}' \backsimeq \mathcal{P}$ which indicates that $\mathcal{P}'$ *may* have a completion that is trace compatible with $\mathcal{P}$ on $\mathcal{E}$. More importantly, if $\sigma \vdash \mathcal{P}' \not\backsimeq \mathcal{P}$, this means that *no* completion of $\mathcal{P}'$ is trace compatible with $\mathcal{P}$ and $\mathcal{P}'$ can be pruned from the search space. Thus, the IsFeasible procedure returns false if it finds any input σ for which $\sigma \vdash \mathcal{P}' \not\backsimeq \mathcal{P}$ (line 7).

In the remainder of this subsection, we discuss our implementation of the $\sigma \vdash \mathcal{P}' \backsimeq \mathcal{P}$ judgment. At a high-level, our technique for checking trace compatibility need to achieve two important goals:

- **Pruning power:** In order to have good pruning power, our rules for checking trace compatibility need to be sufficiently precise. That is, if there is no completion of $\mathcal{P}'$ that is trace compatible with $\mathcal{P}$ on input σ, our inference rules should derive $\sigma \vdash \mathcal{P}' \not\backsimeq \mathcal{P}$ most of the time.
- **Low overhead:** Since our transpilation algorithm invokes the trace compatability checker *many* times, it is crucial that this procedure has low overhead. Thus, it must be efficient to check whether $\sigma \vdash \mathcal{P}' \not\backsimeq \mathcal{P}$.

With these goals in mind, we discuss our trace compatibility checking procedure in two steps. First, we describe inference rules that achieve the first goal (i.e., they are very precise and conceptually easy to understand); however, they fall short of our second goal. To address this shortcoming, we next describe more complex *bidirectional* rules that achieve better scalability without sacrificing precision.

4.2.1 Uni-directional Rules for Trace Compatibility. Given a partial program $\mathcal{P}'$ and an input valuation σ, our rules over-approximate the possible outputs of $\mathcal{P}'$ using a form of *concolic execution* [Sen et al. 2005]. The basic idea is to introduce *symbolic variables* to represent the unknown value of unshared non-terminals. On the other hand, for shared non-terminals, we know which *concrete values* they can take when executing $\mathcal{P}'$ on σ. We then propagate this mix of concrete and symbolic values using concolic execution and then check (using an SMT solver) whether it is possible for $\mathcal{P}'$ to produce the output of $\mathcal{P}$ on σ. If not, $\mathcal{P}'$ and $\mathcal{P}$ are guaranteed *not* to be equivalent under the trace compatibility assumption, so we can safely prune $\mathcal{P}'$ from the search space.

To describe our pruning rules, we assume that the target language $\mathcal{L}_t$ is an instance of the following meta-grammar that takes as arguments (1) F_H, a set of higher-order functions, (2) F, a set of first-order functions, (3) V, a set of variables, and (4) C, a set of constants:

$$E \longrightarrow F_H(E_1, \lambda V.\ E_2) \mid F(E_1, \ldots, E_N) \mid V \mid C$$

We also assume that we have access to the abstract semantics [Cousot and Cousot 1992] for each function (including both first-order and higher-order) in the target language. Given some construct f, we write $[\![f]\!]^\sharp$ to denote the abstract semantics of f.

With these assumptions in place, we can now explain the inference rules from Figure 9. All rules with the exception of the first two derive judgments of the form:

$$\mathcal{P}, \sigma \vdash \mathcal{P}' \uparrow \psi$$

where ψ is a symbolic expression (i.e., an SMT term[4]). The meaning of this judgment is that, given source program $\mathcal{P}$ and input valuation σ, partial program $\mathcal{P}'$ can produce symbolic expression ψ

[4]The exact logical theory used in the SMT encoding depends on the abstract semantics, so we don't specify what logical theory these symbolic expressions belong to.

Uni-directional Trace-Compatibility Checking Rules ↑

$$\frac{\sigma \nvdash \mathcal{P}' \preceq \mathcal{P}}{\sigma \vdash \mathcal{P}' \npreceq \mathcal{P}} \npreceq\text{-}\uparrow \qquad \frac{\mathcal{P}, \sigma \vdash \mathcal{P}' \uparrow \psi \qquad \mathsf{SAT}(\psi = [\![\mathcal{P}]\!]_\sigma)}{\sigma \vdash \mathcal{P}' \preceq \mathcal{P}} \preceq\text{-}\uparrow$$

$$\frac{\neg\mathsf{Shared}(N) \qquad \mathsf{FreshVar}(v)}{\mathcal{P}, \sigma \vdash N \uparrow v} \textsc{NTerm-}\uparrow$$

$$\frac{\mathsf{Shared}(N) \qquad N \Rightarrow^* w \qquad w \in \mathcal{P} \qquad c \in C[\![w]\!]_{\mathcal{P},\sigma}}{\mathcal{P}, \sigma \vdash N \uparrow c} \textsc{S-NTerm-}\uparrow$$

$$\frac{\mathsf{Variable}(v) \qquad \sigma[v] = \psi}{\mathcal{P}, \sigma \vdash v \uparrow \psi} \textsc{VarTerm-}\uparrow$$

$$\frac{\mathsf{FirstOrder}(f) \qquad \overset{\mathsf{Function}(f)}{\forall i.\, \mathcal{P}, \sigma \vdash \tilde{e}_i \uparrow \psi_i} \qquad [\![f]\!]^\sharp(\psi_1, ..., \psi_n) = \psi}{\mathcal{P}, \sigma \vdash f(\tilde{e}_1, ..., \tilde{e}_n) \uparrow \psi} \textsc{FirstOrderTerm-}\uparrow$$

$$\frac{\begin{array}{c}\mathsf{Function}(f) \qquad \mathsf{HigherOrder}(f) \\ \mathcal{P}, \sigma \vdash \tilde{e}_1 \uparrow [\psi_1, ..., \psi_k] \qquad \mathcal{P}, \sigma[i \mapsto \psi_i] \vdash \tilde{e}_2 \uparrow \psi_i' \\ [\![f]\!]^\sharp([\psi_1, ..., \psi_k], \{\psi_1 \mapsto \psi_1', ..., \psi_k \mapsto \psi_k'\}) = \psi\end{array}}{\mathcal{P}, \sigma \vdash f(\tilde{e}_1,\ \lambda i.\, \tilde{e}_2) \uparrow \psi} \textsc{HigherOrderTerm-}\uparrow$$

Fig. 9. Uni-directional rules for checking trace compatibility between partial program in target language and a program in the source language. We assume that lambda bindings in higher-order functions do not shadow input variables.

under the trace compatibility assumption. Thus, according to the second rule in Figure 9, $\mathcal{P}$ and $\mathcal{P}'$ are trace compatible on valuation σ if $\mathcal{P}'$ can produce an expression ψ that is consistent with the output of $\mathcal{P}$ on σ (i.e., we have $\mathsf{SAT}(\psi = [\![\mathcal{P}]\!]_\sigma)$). Furthermore, according to the first rule, $\mathcal{P}$ and $\mathcal{P}'$ are *not* trace compatible under σ if we cannot derive $\sigma \vdash \mathcal{P} \preceq \mathcal{P}'$ using the second rule.

Next, we explain the rules in Figure 9 that derive judgments of the form $\mathcal{P}, \sigma \vdash \mathcal{P}' \uparrow \psi$. The rule labeled NTerm-↑ is for unshared non-terminals and introduces a fresh variable v to represent the unknown value of expressions derived for non-terminal N. The S-NTerm-↑ rule is for *shared* non-terminals N and leverages the trace compatibility assumption. In particular, under this assumption, N can only take on values observed when executing some term w (for $N \Rightarrow^* w$) within $\mathcal{P}$ on input valuation σ. Thus, we "look up" the values that w can take in $\mathcal{P}$ and conclude that N can take value c only if w can evaluate to c when executing $\mathcal{P}$ on input σ.

The last three rules in Figure 9 deal with terminal symbols in the grammar. The rule labeled VarTerm-↑ is for variables and states that variable v evaluates to $\sigma[v]$. The next two rules are for functions but differentiate between first-order and higher-order combinators.[5] For first order functions, we first evaluate the arguments $\tilde{e}_1, \ldots, \tilde{e}_n$ through recursive invocation of the inference rules; let $\psi_1, \ldots, \psi_n$ be the evaluation result. We then symbolically execute function f on these arguments to obtain the symbolic expression ψ for $f(\tilde{e}_1, \ldots, \tilde{e}_n)$.

[5]We view constants as nullary functions; so there is no special rule for constants.

The final rule is for higher-order functions, which, for simplicity, we assume to be of arity two, with the first argument being a list and the second argument being a lambda abstraction. Here, we first evaluate the first argument $\tilde{e}_1$; suppose that the result is a list of length k with symbolic elements $\psi_1, \ldots, \psi_k$. When evaluating the second argument $\lambda i.\tilde{e}_2$, we first bind i to ψ_i and then evaluate $\tilde{e}_2$ as ψ_i'. Finally, since f takes $[\psi_1, \ldots, \psi_n]$ as its first input and the function $\{\psi_1 \mapsto \psi_1', \ldots, \psi_k \mapsto \psi_k'\}$ as its second input, we symbolically evaluate f on these arguments to obtain the final result ψ.

Example 4.1. Consider the following imperative program $\mathcal{P}_s$, which takes in two lists of integers, x_1 and x_2, and produces an output list of all integers i_1 in x_1 for which $i_1 * i_2 + 1$ is prime for some i_2 in x_2. Suppose that the synthesizer proposes the incorrect partial functional program $\mathcal{P}_t$, where I is a shared non-terminal representing integer expressions.

Source program $\mathcal{P}_s$:

```
r = []
for(int i1 : x1) :
    for(int i2 : x2) :
        if prime(i1 * i2 + 1) :
            r.add(i1)
return r
```

Target partial program $\mathcal{P}_t$:

$$\textbf{map}(L_1,\ \lambda\ i_1.\ I_1 * (I_2 + I_3)))$$

For input $\sigma = \{x_1 \mapsto [1, 2, 3, 4, 5], x_2 \mapsto [11, 70, 61, 72, 61]\}$, $\mathcal{P}_s$ produces the output $[1, 1, 2, 3, 4]$, and we can prune $\mathcal{P}_t$ using these input-output examples and the trace compatibility assumption. In particular, each occurrence of shared non-terminal I can take values of $i_1, i_2, i_1 * i_2$, and $i_1 * i_2 + 1$ in the source program. Since these expressions only take positive values, we infer that $I_1 * (I_2 + I_3)$ cannot be 1. However, using the semantics of `map`, this would imply that the output list cannot contain 1, thereby contradicting the fact that the output list is $[1, 1, 2, 3, 4]$.

4.2.2 Bidirectional Rules for Checking Trace Compatibility. While the inference rules from Section 4.2.1 are sufficient for precisely checking trace compatibility, they can be very inefficient to execute. To gain some intuition, consider a term $f(\tilde{e}_1, \ldots, \tilde{e}_k)$ and suppose that each $\tilde{e}_i$ can evaluate to n different symbolic expressions. Then, there are a total of n^k different expressions that f can evaluate to. Thus, as this example illustrates, checking trace compatibility can be exponential in the size of the partial program being evaluated.

To mitigate this problem, we augment our trace compatibility checking rules using backwards reasoning. The idea is to use the inverse semantics of constructs in the target language to obtain a specification φ of each sub-term being evaluated. Then, if a sub-term evaluates to an expression ψ that is not consistent with φ (i.e., $\mathsf{Unsat}(\varphi(\psi))$, we know that ψ is infeasible and we need not propagate it forwards. Hence, the use of backwards reasoning allows us to control the number of symbolic expressions being propagated forwards and prevents this exponential blow-up in practice.

With this intuition in mind, we now explain the bidirectional trace compatibility checking rules shown in Figure 10 which derive judgments of the following form:

$$\mathcal{P}, \sigma, \varphi \vdash \mathcal{P}' \updownarrow \psi$$

The meaning of this judgment is that $\mathcal{P}'$ can evaluate to ψ under the assumption that (1) $\mathcal{P}'$ satisfies φ and (2) $\mathcal{P}'$ is trace compatible with $\mathcal{P}$ on valuation σ. As expected, these rules are a refinement of those from Figure 9 and the differences from Figure 9 are indicated in blue. The rule $\preceq$-$\updownarrow$ is the same as $\preceq$-$\uparrow$, but it initializes the specification φ of $\mathcal{P}'$ to state that $\mathcal{P}'$ must produce the same output as $\mathcal{P}$ when executed on σ. The rules labeled NTerm-$\updownarrow$, S-NTerm-$\updownarrow$, and VarTerm-$\updownarrow$ are exactly the

Bidirectional Trace-Compatibility Checking Rules ↕

$$\frac{\sigma \not\vdash \mathcal{P}' \backsimeq \mathcal{P}}{\sigma \vdash \mathcal{P}' \not\backsimeq \mathcal{P}} \not\backsimeq\text{-}\updownarrow \qquad \frac{\mathcal{P}, \sigma, y = [\![\mathcal{P}]\!]_\sigma \vdash \mathcal{P}' \updownarrow \psi \qquad \text{SAT}(\psi = [\![\mathcal{P}]\!]_\sigma)}{\sigma \vdash \mathcal{P}' \backsimeq \mathcal{P}} \backsimeq\text{-}\updownarrow$$

$$\frac{\neg\text{Shared}(N) \qquad \text{FreshVar}(v) \qquad \text{SAT}(\varphi[v])}{\mathcal{P}, \sigma, \varphi \vdash N \updownarrow v} \textsc{NTerm-}\updownarrow$$

$$\frac{\begin{array}{c}\text{Shared}(N) \\ N \Rightarrow^* w \qquad w \in \mathcal{P} \qquad c \in C[\![w]\!]_{\mathcal{P},\sigma} \qquad \text{SAT}(\varphi[c])\end{array}}{\mathcal{P}, \sigma, \varphi \vdash N \updownarrow c} \textsc{S-NTerm-}\updownarrow$$

$$\frac{\text{Variable}(v) \qquad \sigma[v] = \psi \qquad \text{SAT}(\varphi[\psi])}{\mathcal{P}, \sigma, \varphi \vdash v \updownarrow \psi} \textsc{VarTerm-}\updownarrow$$

$$\frac{\begin{array}{c}\text{Function}(f) \\ \text{FirstOrder}(f) \qquad \forall i.\, \mathcal{P}, \sigma, [\![f]\!]^{\sharp^{-i}}(\varphi, \psi_1, ..., \psi_{i-1}) \vdash \tilde{e}_i \updownarrow \psi_i \\ [\![f]\!]^\sharp(\psi_1, ..., \psi_n) = \psi \qquad \text{SAT}(\varphi[\psi])\end{array}}{\mathcal{P}, \sigma, \varphi \vdash f(\tilde{e}_1, ..., \tilde{e}_n) \updownarrow \psi} \textsc{FirstOrderTerm-}\updownarrow$$

$$\frac{\begin{array}{c}\text{Function}(f) \\ \text{HigherOrder}(f) \qquad \mathcal{P}, \sigma, [\![f]\!]^{\sharp^{-1}}(\varphi) \vdash \tilde{e}_1 \updownarrow [\psi_1, ..., \psi_k] \\ \mathcal{P}, \sigma[i \mapsto \psi_i], [\![f]\!]^{\sharp^{-2}}(\varphi, \psi_i) \vdash \tilde{e}_2 \updownarrow \psi_i' \\ [\![f]\!]^\sharp([\psi_1, ..., \psi_k], \{\psi_1 \mapsto \psi_1', ..., \psi_k \mapsto \psi_k'\}) = \psi \\ \text{SAT}(\varphi[\psi])\end{array}}{\mathcal{P}, \sigma, \varphi \vdash f(\tilde{e}_1, \lambda i.\, \tilde{e}_2) \updownarrow \psi} \textsc{HigherOrderTerm-}\updownarrow$$

Fig. 10. Bidirectional Rules

same as the corresponding rules from Figure 9 with the only difference being the consistency check between the specification and the result of the concolic evaluation.

The last two rules for function terms are slightly more involved and make use of the inverse semantics. In particular, in the rule labeled FirstOrderTerm-↕, we first compute the specification for argument $\tilde{e}_i$ using the inverse semantics of f with the respect to its i'th argument. Specifically, we use the notation $[\![f]\!]^{\sharp^{-i}}$ to denote the function that takes as input the specification for the whole term and the (symbolic) values for expressions $\psi_1, \ldots, \psi_{i-1}$ and yields the specification for the i'th argument. Then, when evaluating ψ_i, we use $[\![f]\!]^{\sharp^{-i}}(\varphi, \psi_1, \ldots, \psi_{i-1})$ as the specification.

Example 4.2. Consider the term $+(N_1, N_2)$ and suppose that we have:

$$\mathcal{P}, \sigma, [\![+]\!]^{\sharp^{-1}}(y > 0) \vdash N_1 \updownarrow 1$$

Note, the argument $y > 0$ of $[\![+]\!]^{\sharp^{-1}}$ is the specification for the entire term, i.e., $+(N_1, N_2) > 0$. Then, using the inverse semantics of $+$, we can infer the specification $y > -1$ for the second argument N_2. This specification can be used to prune all negative values for N_2.

The next rule for higher-order combinators is similar and uses the inverse semantics in exactly the same way. In particular, note that the satisfiability check reduces the space of possible evaluation results ψ for $f(e_1, \lambda i_1.e_2)$ and therefore reduces the number of terms propagated forward.

The following theorem states that our pruning rules only rule out partial programs that are not trace compatible with the source program on valuation σ.

Theorem 4.3. *Suppose that $\sigma \vdash \mathcal{P} \neq \mathcal{P}'$ and both the forward semantics $[\![\cdot]\!]^\sharp$ and backward semantics $[\![\cdot]\!]^{\sharp -i}$ provided are sound. Then, there is no completion $\mathcal{P}_t$ of $\mathcal{P}'$ that is trace-compatible with $\mathcal{P}$ on input σ where $[\![\mathcal{P}_t]\!]_\sigma = [\![\mathcal{P}]\!]_\sigma$.*

Proof. Please see the extended version of the paper [Mariano et al. 2022] for full proofs. □

5 COGNATE GRAMMAR NETWORK

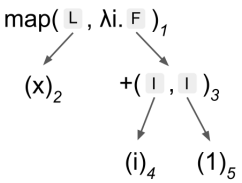
The transpilation algorithm from Figure 7 uses a neural model to guide its search. In this section, we discuss the *cognate grammar network* (CGN) that serves as our custom neural model in this context. In what follows, we first explain the *abstract syntax network* (ASN) model on which CGN is based [Rabinovich et al. 2017], then describe how CGN extends the ASN architecture with a copying mechanism [Jia and Liang 2016; See et al. 2017] to handle shared nonterminals, and, finally, we give a brief overview of our training phase.

5.1 ASN Preliminaries

Our goal is to model the distribution $p(\mathcal{P}_t \mid \mathcal{P}_s)$ over programs $\mathcal{P}_t$ in the target language given programs $\mathcal{P}_s$ in the source language. An abstract syntax network models this distribution by decomposing the probability of a complete target program $\mathcal{P}_t$ into a series of probabilities for each AST node in $\mathcal{P}_t$.

Notation. To formalize this decomposition, we first introduce some notation. Following [Ye et al. 2020], given node n in (complete) program $\mathcal{P}$ (simply denoted $n \in \mathcal{P}$), we define an AST path $\pi(\mathcal{P}, n) = ((n_1, i_1), \ldots, (n_k, i_k))$ to be a sequence of (node, index) pairs where node n_{j+1} is the i_j'th child of node n_j, and the i_k'th child of node n_k is n. In other words, $\pi(\mathcal{P}, n)$ is the path from the root node of the AST to node n. For each node n, we use the notation $\mathcal{R}(n)$ to denote the CFG production used to assign a terminal symbol to n.

Example 5.1. The AST for the simple functional program $\mathsf{map}(x, \lambda i.i + 1)$ is pictured below. The name for each node is given as a subscript.



The ast path to node n_4, denoted $\pi(\mathcal{P}, n_4)$, is $((n_1, 2), (n_3, 1))$ and the production rule used to assign n_4, $\mathcal{R}(n_4)$, is $I \rightarrow i$.

The ASN $\mathcal{M}_\theta$ models $p(\mathcal{P}_t \mid \mathcal{P}_s)$ using the distribution $\mathcal{M}_\theta(\mathcal{R}(n) \mid \pi(\mathcal{P}_t, n), \mathcal{P}_s)$ over grammar productions $\mathcal{R}(n)$ for each node n in $\mathcal{P}_t$ given the AST path π to n and the source program $\mathcal{P}_s$. In particular, the probability $\mathcal{M}_\theta(\mathcal{P}_t \mid \mathcal{P}_s)$ is computed as follows:

$$\mathcal{M}_\theta(\mathcal{P}_t \mid \mathcal{P}_s) = \prod_{n \in \mathcal{P}_t} \mathcal{M}_\theta(\mathcal{R}(n) \mid \pi(\mathcal{P}_t, n), \mathcal{P}_s) \tag{2}$$

In other words, the probability of program $\mathcal{P}_t$ given $\mathcal{P}_s$ is given as the product of the probability of each grammar rule used to construct $\mathcal{P}_t$.

Following past work [Ye et al. 2020], we compute Equation 2 by first encoding the source program $\mathcal{P}_s$ with a BiLSTM [Hochreiter and Schmidhuber 1997] over its linearized representation $l(\mathcal{P}_s)$ to form an embedding $h_p = \text{BiLSTM}(l(\mathcal{P}_s))$. In this work, $l(\mathcal{P}_s)$ simply treats $\mathcal{P}_s$ as a sequence of tokens, where each token corresponds to a keyword or symbol from $\mathcal{P}_s$ (e.g., `for`, `if`, `=`, etc.). We then use an LSTM[6] to produce embedding $h_n = \text{LSTM}(h_p, \pi(\mathcal{P}_t, n))$ for node n of the target program AST, which will be used to form the distribution over production rules at this node. In particular, given embedding h_n, the probability for each production rule at node n is computed using a feedforward neural network (FFNN) module with attention over the input source program:

$$\mathcal{M}_\theta(\cdot \mid \pi(\mathcal{P}_t, n), \mathcal{P}_s) = \text{softmax}(\text{FFNN}(h_n; \text{Attn}(h_n, \text{BiLSTM}(l(\mathcal{P}_s))))) \tag{3}$$

Attention is a standard tool in sequence-to-sequence models for making information from the input more accessible to the decoder model [Bahdanau et al. 2015; Luong et al. 2015]. In our case, the attention layer computes a distribution over tokens in the linearized source program $l(\mathcal{P}_s)$ for each node n, indicating which input tokens contribute most (and least) to the probability of production rules used at n.

5.2 Proposed CGN Model

While the ASN model allows us to model $p(\mathcal{P}_t \mid \mathcal{P}_s)$, it is not necessarily well suited to our problem setting. In particular, source and target programs in our context are generated using cognate grammars, where some terms are shared between the two programs. For nodes corresponding to shared non-terminals, we want our model to produce a probability distribution over *terms* in the source program as opposed to *production rules* in the grammar. Thus, we propose the *cognate grammar network (CGN)* model, which comprises (1) a *copy* component for handling nodes that correspond to shared non-terminals, and (2) an ASN component for handling nodes corresponding to unshared non-terminals. In essence, the copy component directly *copies* terms from the source program to the target program in the same way that pointer networks copy portions of input text during text summarization [See et al. 2017], while the ASN component builds target program terms from grammar rules in the target language.

In more detail, for each node n that *does not* correspond to a shared non-terminal (i.e., the LHS of $\mathcal{R}(n)$ is *not* a shared non-terminal), the ASN component of the CGN simply uses Equation 3 for determining the probability of a production rule at n. However, for each node n which corresponds to a shared non-terminal N, the copy component of the CGN computes the distribution $p(t \mid \pi(\mathcal{P}_t, n), \mathcal{P}_s)$ over terms t from the source program derivable from N ($N \Rightarrow^* t$) given the AST path to n and source program $\mathcal{P}_s$. To compute this, we must first compute an encoding for the entire source program term t, which (potentially) consists of multiple tokens, each with their own separate LSTM hidden state embedding. Following work on encoding spans in constituency parsing models [Kitaev and Klein 2018], we compute the encoding of t as the difference of the LSTM hidden states corresponding to the start and end tokens of t as follows:

$$\text{enc}(t) = \frac{1}{2}(s_S - s_E) \tag{4}$$

where s_S and s_E are the LSTM hidden states for the start and end tokens respectively, and $\frac{1}{2}$ is a scaling factor.

[6]Note that this is a unidirectional LSTM, since we are encoding a sequential "history" of the node to predict its children. The most recent ancestors in the tree are what matter the most.

To compute the probability of copying some source program term t, we learn a weight matrix W which is combined with the term encoding $\text{enc}(t)$ and target program node embedding h_n as follows:

$$\mathcal{M}_\theta(t \mid \pi(\mathcal{P}_t, n), \mathcal{P}_s) = \text{softmax}(h_n^\top W \text{enc}(t)) \tag{5}$$

5.3 Training

Our model is trained on a labeled dataset of paired programs $\mathcal{D} = (\mathcal{P}_s, \mathcal{P}_t)$. As is standard for this type of model, we maximize the conditional log probability of the target program given the source program:

$$\mathcal{L}(\theta) = \sum_{i=1}^{|\mathcal{D}|} \log \mathcal{M}_\theta(\mathcal{P}_t \mid \mathcal{P}_s) \tag{6}$$

It should be noted that we can directly minimize this quantity, as the copy component is invoked if and only if a node corresponds to a shared non-terminal and the corresponding source-side shared terminal is unambiguous. In other words, the ground-truth decision for each production rule choice $\mathcal{R}(n)$ or term choice t at node n is unambiguous, regardless of whether n corresponds to a shared or unshared non-terminal.

Similar to prior work [Ye et al. 2020], we train the networks using *complete* programs as opposed to the partial programs seen at test time. This is possible because both the ASN and CGN compute a probability distribution based on the source program $\mathcal{P}_s$ and AST path $\pi(\mathcal{P}_t, n)$ to the node n being expanded (Equations 3 and 5). Thus, when learning these probability distributions from complete program pairs $(\mathcal{P}_s, \mathcal{P}_t)$, we train the network for each node n in $\mathcal{P}_t$ where the ground-truth label $\mathcal{R}(n)$ (or t for CGN) is given by the choice of grammar rule (or term for CGN) used to expand n in $\mathcal{P}_t$ and the embeddings for $\pi(\mathcal{P}_t, n)$ and $\mathcal{P}_s$ are computed as described in Section 5.1.

6 IMPLEMENTATION AND INSTANTIATIONS

We implemented our transpilation algorithm in a tool called NGST2, which is parameterized over the source and target language grammars. In this section, we first discuss salient aspects of our implementation and then describe how to instantiate NGST2 for a pair of source/target languages.

6.1 NGST2 Implementation

NGST2 is implemented in Python and performs several optimizations over the algorithm presented in Sections 4 and 5.

Training. For training our CGN, we use the Adam optimizer [Kingma and Ba 2015], with early stopping based on the accuracy achieved on the validation set and a dropout mechanism [Srivastava et al. 2014] to prevent model overfitting. Additionally, we apply the gradient norm clipping method [Pascanu et al. 2013] to stabilize training.

Training data. As there are no existing datasets for our target application domains, we generated synthetic source/target program pairs to train the CGN. Our data generation procedure relies on a key insight: *while translating imperative code to functional APIs is challenging, translating functional APIs to imperative code is straightforward.* In particular, this *reverse* translation problem can be achieved with a small set of composable translation rules. We propose a translation function $\text{translate}(e, r)$ that converts a functional expression e into equivalent imperative code c with return value saved to variable r. For example, consider the following implementation of translate for the functional operator *map*:

$$
\begin{array}{ll}
\text{Application } A & \rightarrow \mathsf{map}(A, \lambda V.A) \mid \mathsf{filter}(A, \lambda V.E) \mid \mathsf{flatmap}(A, \lambda V.A) \mid \\
& \quad \mathsf{find}(E, A, \lambda V.E) \mid \mathsf{fold}(E, A, \lambda V, V.E) \mid E \\
\text{Expression } E & \rightarrow V \mid C \mid f(\overline{E}) \\
\text{Variable } V & \rightarrow x \mid i_1 \mid \ldots \mid i_m \\
\text{Constant } C & \rightarrow k \in \mathbb{Z} \mid b \in \mathbb{B}
\end{array}
$$

Fig. 11. Target language λ_{str} where f denotes a first-order function

```
translate(map(e1, λi. e2), r) =      translate(e1, r1)
                                     for i ∈ r1:
                                       translate(e2, r2)
                                       r.add(r2)
```

This translation rule simply inlines the result of translating subexpressions e_1 and e_2, and rules for other higher-order operators follow a similar pattern. Thus, to generate training data, we randomly sample functional programs and use the the translator described above to generate matching imperative programs. Note that using this same approach in reverse is not feasible – i.e., applying these translation rules in reverse would not yield a solution to our problem. This is because these rules construct a *separate loop* for each higher-order operator in a functional expression. Thus, to apply the rules in reverse (i.e., to apply compositional rules to imperative code), each loop would need to correspond to exactly one higher-order operator, which is rarely the case. For example, the three higher-order operators in Figure 3 correspond to only two loops in Figure 2.

Equivalence checker. Our implementation utilizes a (bounded) verifier implemented on top of Rosette [Torlak and Bodik 2013] to check equivalence between the source and target programs. Specifically, the checker takes as input two programs $\mathcal{P}_s, \mathcal{P}_t$ as well as a symbolic input x and verification bound K that determines how many times loops are unrolled. It then symbolically executes $\mathcal{P}_t$ and $\mathcal{P}_s$ on x, resulting in symbolic output states y_t and y_s. If y_t and y_s are proven equal, $\mathcal{P}_t$ and $\mathcal{P}_s$ are known to be equivalent up to bound K. While the use of a bounded verifier can, in principle, result in NGST2 producing the wrong target program, we have not found this to be an issue in practice.

6.2 Instantiating NGST2

We have instantiated NGST2 for two pairs of source/target languages. Our first client involves translating standard Java code to functional equivalents written using the Stream API, and the second requires converting imperative Python to code snippets using the `functools` API. To give the reader a sense of what it takes to customize NGST2, we describe how we instantiated it in the context of our Java client.

Grammar and shared non-terminals. Since most readers are familiar with standard Java, we only present our target language, which is shown in Figure 11 and which we refer to as λ_{str} in the rest of this section. λ_{str} allows functional programs composed of common higher-order functional operators, such as map, filter, and fold. In addition, λ_{str} also contains two other operators find and flatmap. As its name indicates, $\mathsf{flatmap}(e_1, \lambda i.e_2)$ is similar to map in that it takes each element of e_1 and applies e_2 to it. However, for flatmap, e_2 is a function which produces a list, and the final result is flattened to a single list. For example, $\mathsf{flatmap}(x,\ \lambda i.\ \mathsf{factors}(i))$ will produce a list of all factors of integers appearing in the list x. As for $\mathsf{find}(e_1, e_2, \lambda i.e_3)$, it returns the first element of list e_2 satisfying condition e_3. If no such element is found, the default value e_1 is returned. For example, $\mathsf{find}(0,\ x,\ \lambda i.\ i\%2 == 1)$ finds the first odd element of x, or 0 if none is found.

Forward Semantics $[\![\cdot]\!]^\sharp$

$$
\begin{array}{lcl}
[\![\text{map}]\!]^\sharp([\psi_1,\ldots,\psi_k], \uplus\{\psi_i \mapsto \psi_i'\}) & \triangleq & [\psi_1',\ldots,\psi_k'] \\
[\![\text{flatmap}]\!]^\sharp([\psi_1,\ldots,\psi_k], \uplus\{\psi_i \to [\psi_{i1}',\ldots,\psi_{il_i}']\}) & \triangleq & [\psi_{11}',\ldots,\psi_{1l_1}',\ldots,\psi_{k1}',\ldots,\psi_{kl_k}'] \\
[\![\text{filter}]\!]^\sharp([\psi_1,\ldots,\psi_k], \uplus\{\psi_i \to \psi_i'\}) & \triangleq & \text{append}(\bar{\psi}_1,...,\bar{\psi}_k) \\
 & & \quad \text{where } \bar{\psi}_i = (\psi_i' \ ?\ \psi_i : \bot) \\
[\![\text{find}]\!]^\sharp(\psi_0, [\psi_1,\ldots,\psi_k], f) & \triangleq & (\text{len}(l) = 0\ ?\ \psi_0 : \text{hd}(l)) \\
 & & \quad \text{where } l = [\![\text{filter}]\!]^\sharp([\psi_1,\ldots,\psi_k], f) \\
[\![\text{fold}]\!]^\sharp(\psi_0, [], \{\}) & \triangleq & \psi_0 \\
[\![\text{fold}]\!]^\sharp(\psi_0, [\psi_1,\ldots,\psi_k], \uplus\{(\psi_{i-1},\psi_i) \to \psi_i'\}) & \triangleq & \psi_k'
\end{array}
$$

Fig. 12. Forward semantics for λ_{str}.

Backward Semantics $[\![\cdot]\!]^{\sharp-i}$

$$
\begin{array}{lcl}
[\![\text{map}]\!]^{\sharp-1}(y = [\psi_1',\ldots,\psi_n']) & \triangleq & \text{len}(y) = n \\
[\![\text{map}]\!]^{\sharp-2}(y = [\psi_1',\ldots,\psi_n'], \psi_i) & \triangleq & y = \psi_i' \\
[\![\text{flatmap}]\!]^{\sharp-1}(y = [\psi_1',\ldots,\psi_n']) & \triangleq & \text{true} \\
[\![\text{flatmap}]\!]^{\sharp-2}(y = [\psi_1',\ldots,\psi_n'], \psi_i) & \triangleq & \text{subarr}(y, [\psi_1',\ldots,\psi_n']) \ \wedge\ \text{len}(y) \leq n \\
[\![\text{filter}]\!]^{\sharp-1}(y = [\psi_1',\ldots,\psi_n']) & \triangleq & \text{len}(y) \geq n \wedge \text{subseq}([\psi_1',\ldots,\psi_n'], y) \\
[\![\text{filter}]\!]^{\sharp-2}(y = [\psi_1',\ldots,\psi_n'], \psi_i) & \triangleq & \left(\bigwedge_{j=1}^{n} \psi_i \neq \psi_j'\right) \to y = \text{false} \\
[\![\text{find}]\!]^{\sharp-1}(y = \psi') & \triangleq & \text{true} \\
[\![\text{find}]\!]^{\sharp-2}(y = \psi', \psi_0) & \triangleq & (\psi' \neq \psi_0) \to \psi' \in y \\
[\![\text{find}]\!]^{\sharp-3}(y = \psi', \psi_0, \psi_i) & \triangleq & \psi' \neq \psi_i \to y = \text{false} \\
[\![\text{fold}]\!]^{\sharp-1}(y = \psi') & \triangleq & \text{true} \\
[\![\text{fold}]\!]^{\sharp-2}(y = \psi', \psi_0) & \triangleq & \text{true} \\
[\![\text{fold}]\!]^{\sharp-3}(y = \psi', \psi_0, \psi_i) & \triangleq & y = \psi' \ \text{ if } \psi_i \text{ final arg}
\end{array}
$$

Fig. 13. Backward semantics for λ_{str}.

In order to leverage the synthesis technique proposed in this paper, we formulate the source and target languages as *cognate grammars* as defined in Def. 3.5. Our choice of shared non-terminals is motivated by the observation from Section 1: while the high-level structure of the source and target programs are very different, low-level expressions tend to be shared. In the grammar shown in Fig 11, we designate non-terminals E, V, and C as shared, while the top-level non-terminal A is unshared.

Forward semantics. Recall that our basic rules from Figure 9 make use of the forward semantics of the target language. Figure 12 shows the semantics that we use for λ_{str} in our implementation. Since these rules are based directly on the standard semantics of `map`, `fold` etc., we do not explain them in detail. Also, these rules utilize standard auxiliary functions such as `append` and can be easily converted into an SMT encoding.

Backward semantics. Recall from Section 4.2.2 that our method also utilizes the backward semantics of each target language construct to make trace compatibility checking more practical. Figure 13 shows the inverse semantics for λ_{str}. Here, we use the notation $[\![f]\!]^{\sharp-i}(\varphi, \psi_1, \ldots \psi_{i-1})$ to indicate the specification for the i'th argument of f given the specification φ of the output and symbolic values for the first i arguments. We further assume that all specifications φ contain a single free variable y that refers to the output of the expression.

Table 1. Benchmark Information.

Client	Imperative Stats			Functional Stats		
	AST Nodes			AST Nodes		
	Avg	Med	Max	Avg	Med	Max
Stream	66	62	127	40	35	100
Pythonic	76	72	198	38	36	159

7 EXPERIMENTS

In this section, we describe the results of a series of experiments that are designed to answer the following research questions:

RQ1 How does NGST2 compare to existing techniques?
RQ2 How important are the various design decisions underlying NGST2? In particular, how important are our proposed neural architecture and pruning techniques?
RQ3 How does the trace compatibility assumption limit NGST2's ability to transpile real-world benchmarks?

All experiments are run with a 5-minute timeout on a 2019 MacBook Pro with an 8-core i9 processor and 16 GB RAM.

7.1 Benchmark Collection

As mentioned earlier in Section 6.2, we instantiated NGST2 for two types of transpilation tasks, namely (1) converting imperative Java to functional code using the `Stream` API and (2) translating Python code to a functional variant using the `functools` API. Unlike our training benchmarks that were generated synthetically, it is important to evaluate NGST2 on *test* benchmarks that are representative of real-world code. Towards this goal, we implemented a web crawler to extract real-world transpilation tasks from Github and Stackoverflow. In particular, our script looks for Github commits and Stackoverflow posts that meet the following criteria:

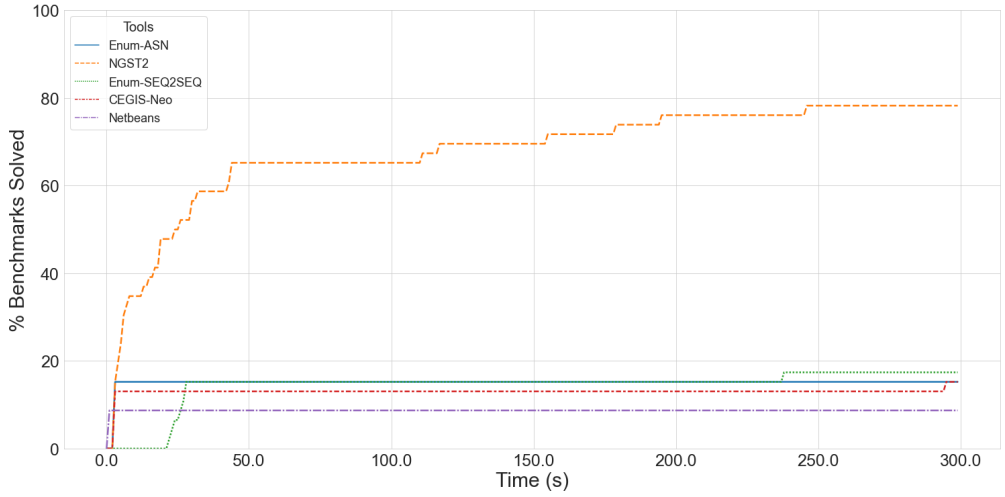
(1) Contains relevant keywords such as `Java stream`, `Pythonic loops`, or `List comprehension`.
(2) Includes both the imperative code and its functional translation in the versions before/after the commit or Stackoverflow post/answer.
(3) Passes our non-triviality checks such as the imperative version containing at least one loop, and the functional version containing at least two higher-order combinators.

Using this methodology, we obtained a total of 91 benchmarks, 46 of which are for Java and 45 of which are Python. To give the reader some intuition about the complexity of these benchmarks, Table 1 presents various statistics about the source/target versions of these programs.

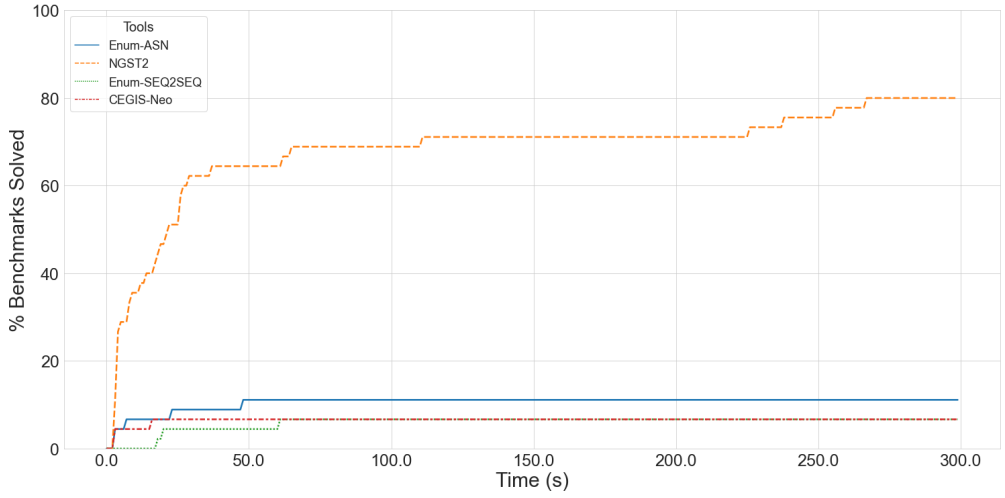
7.2 Comparison against Existing Techniques

To answer our first research question, we compared the performance of NGST2 against existing techniques. In particular, we consider the following four baselines:

- **CEGIS-Neo**, a CEGIS-based synthesizer that uses Neo [Feng et al. 2018] as the inductive synthesis backend and our equivalence checker as the verifier
- **Enum-ASN**, an enumerative synthesizer in which we enumerate syntactically valid code according to its likelihood using an abstract syntax network (ASN) model [Rabinovich et al. 2017] and then verify correctness using our equivalence checker
- **Enum-Seq2Seq**, another enumerative synthesizer based on a Sequence to Sequence (SEQ2SEQ) model [Sutskever et al. 2014] in which we enumerate programs based on the score of the SEQ2SEQ model and verify correctness using our equivalence checker



(a) Stream results.



(b) Python results.

Fig. 14. Performance comparison with existing tools.

- **Netbeans**, a translator (implemented as a Netbeans plugin) that uses a set of handcrafted rules to modernize imperative Java code to use the Stream API [Gyori et al. 2013]

Note that the first three of these baselines are applicable to both of our clients, while the last one is specific to Java. For all tools which rely on a neural component (i.e., NGST2, Enum-ASN, and Enum-Seq2Seq) we use the same training data and training methodology described in Section 6. Additionally, to set hyperparameters for these systems, we experimented with 10 different common/intuitive settings for each and used the one which produced the best results.

The results of this experiment are summarized in Figure 14. Here, the x-axis shows the time (per benchmark) and the y-axis presents the percentage of benchmarks solved within that time limit. As we can see in both figures, NGST2 significantly outperforms all baselines in both clients.

In particular, NGST2 solves at least 60% more benchmarks than any other baseline across both clients. We believe the major cause of poor performance for the other synthesizers (Enum-ASN, Enum-Seq2Seq, CEGIS-Neo) is the large search space of programs to be searched. In contrast to these tools, NGST2 leverages the trace compatibility assumption to prune large parts of the search space. The poor performance of the Netbeans plugin is likely due to an inadequate set of hand-made translation templates; as shown, in the few cases where the plugin did have the correct templates, translation was very fast.

> **Result for RQ1:** NGST2 significantly outperforms existing baselines for both the Java and Python clients. In particular, NGST2 solves 78% of the Java benchmarks, while its closest competitor (Enum-Seq2Seq) solves 17%. For the Python client, NGST2 solves 80% of benchmarks, while the best baseline (Enum-ASN) solves only 11%.

7.3 Ablation Studies

In this section, we describe a series of ablation studies that are designed to answer our second research question. In particular, we consider the following ablations of NGST2:

- **NGST2-NoPrune:** This is a variant of NGST2 that does not perform pruning based on trace compatibility. In particular, it does not leverage any of the techniques discussed in Section 4.2.
- **NGST2-Forward:** This variant of NGST2 uses the *uni-directional* pruning rules from Section 4.2; however, it does not utilize the backward semantics to reduce the overhead of pruning.
- **NGST2-NoTrace:** This variant of NGST2 uses the *bidirectional* pruning rules from Section 4.2; however, it does not utilize *intermediate* values for shared non-terminals; i.e., rule S − NTerm− $\updownarrow$ simply returns a fresh variable v like rule NTerm− $\updownarrow$.
- **ASN-Bidirec:** This ablation uses the ASN architecture instead of our proposed CGN.
- **ASN-NoTrace:** This variant of NGST2 is the same as NGST2-NoTrace, except it uses the ASN to guide search rather than the CGN.

Again, similar to our baseline comparison, we use the same training data, training methodology, and hyperparameter setting approach for all ablations.

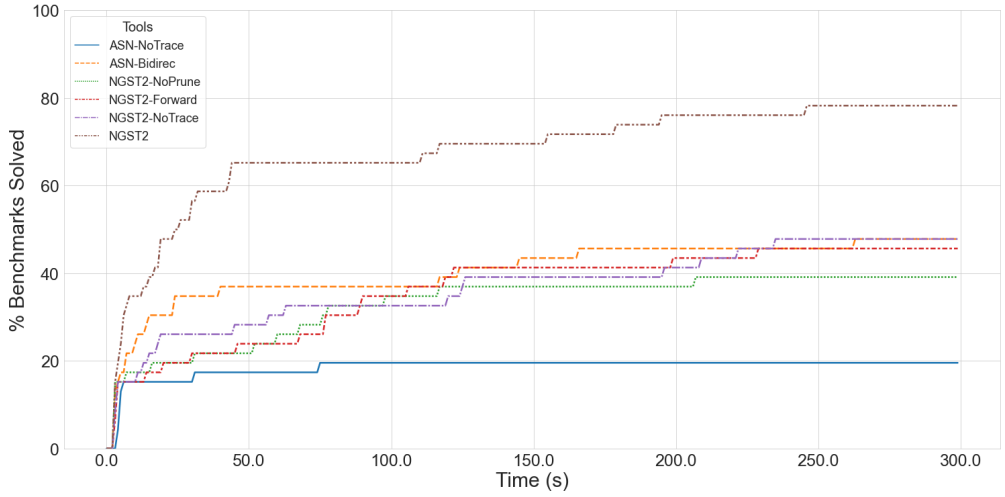
The results of these ablation studies are summarized in Figure 15 using the same type of cactus plot as in Figure 14. The main take-away from this experiment is that NGST2 significantly outperforms all other variants for both clients. In particular, NGST2 solves more benchmarks in the first 45 seconds than any other variant solves in the total 5 minutes.

In terms of the relative importance of the proposed neural architecture vs. the pruning component, the data are mixed. For the Java Stream client, ASN-Bidirec outperforms NGST2-NoPrune, while NGST2-NoPrune outperforms ASN-Bidirec for the Python client. It appears that, for some benchmarks, the CGN is more important to performance than bidirectional pruning, while for others, pruning is more important than the CGN. However, it is clear that using a combination of bidirectional pruning and the proposed CGN architecture leads to the best performance.

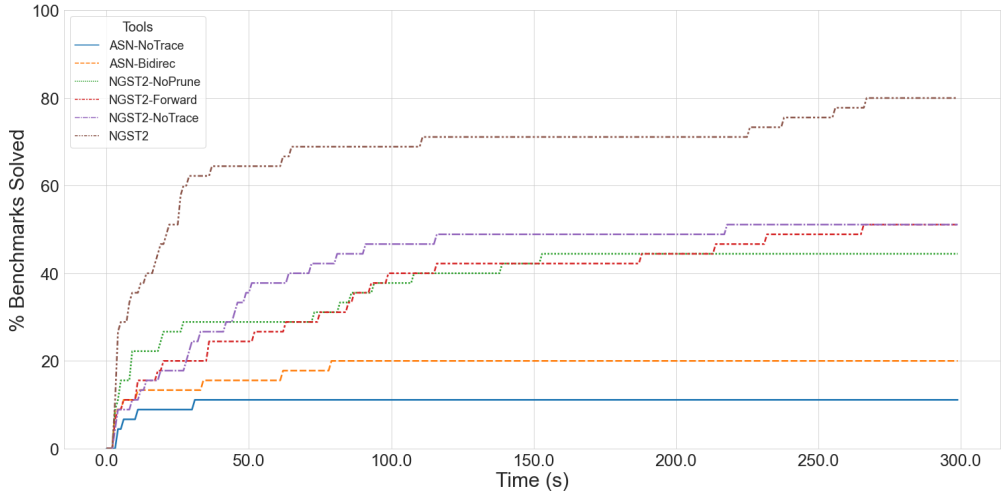
> **Result for RQ2:** Both the pruning mechanism and the CGN architecture are vital to NGST2; removing either component results in more than a 25% reduction in benchmarks solved.

7.4 Evaluation of Trace Compatibility Assumption

To answer our third research question, we performed a manual evaluation of the benchmarks that NGST2 fails to synthesize within the time limit. We found only 3 of the 91 fail to synthesize because



(a) Stream results.



(b) Python results.

Fig. 15. Performance comparison with NGST2 variants.

Input

```
def lowerIn(strs, S):
    for s in strs:
        if s in S:
            return s.lower()
    return None
```

Output

```
find(map(filter(strs,
                λ s. s in S),
         λ s. s.lower()),
     None,
     λ s. True)
```

Fig. 16. Non-trace-compatible Benchmark Example

they do not satisfy our trace compatibility assumption. However, as we discuss below, it is possible to solve all three benchmarks using a slight relaxation of the trace compatibility assumption.

To provide intuition, Figure 16 shows a simplified version of one of the three benchmarks that violate trace compatibility. Here, the imperative function iterates through each string `s` from a list of strings `strs` and returns the lower-case version of the first string which appears in `S`. The functional translation is shown next to the imperative version: it first uses the `map` and `filter` operators to get the lower case versions of all strings `s` in `S`. Then, it leverages the `find` combinator to return the first element of that list if one exists. The source and target programs shown in Figure 16 are not trace complete, as the expression `True` in the target does not appear in the source program. However, if we relax the trace compatibility assumption slightly and allow the target program to contain additional "default" constants such as True, False, and 0, then all three benchmarks can be correctly transpiled by NGST2.

Result for RQ3: Only 3 of the 91 benchmarks violate the trace-compatibility assumption. Furthermore, if we slightly relax the trace compatibility assumption to allow default constants, then those three benchmarks can also be successfully transpiled by NGST2.

8 LIMITATIONS

Our approach has some limitations based on the assumptions we made. First, we assume the target language belongs to the meta-grammar shown in Section 4.2.1. While in theory this could restrict the space of languages we can translate to, in practice we found this meta-grammar is expressive enough to allow all of the functional APIs we were interested in. Second, we assume that we have access to the forward and backward semantics of the target language constructs and that these semantics can be encoded in some first-order theory. We do not believe this to be a major limitation, as our method does not require the precise semantics of language constructs, and thus they can be over-approximated in some reasonable manner. For example, we encode the forward and backwards semantics for Java Stream operators via the rules shown in Figures 12 and 13. Finally, our proposed neural approach assumes there is sufficient data on which to train the network. However, we also do not see this as a considerable limitation – we found that easily-generated synthetic training data was sufficient for all the translation problems we considered (Section 6).

9 RELATED WORK

In this section, we survey prior work that is most closely related to our proposed approach.

Transpilation. A number of works use program synthesis techniques for transpilation across a variety of domains. For instance, Ahmad and Cheung [Ahmad and Cheung 2018] introduce Casper, a tool for synthesizing programs in the MapReduce paradigm, using imperative source implementations [7]. QBS [Cheung et al. 2013] uses synthesis to translate application code fragments to SQL queries and STNG [Kamil et al. 2016] translates low-level Fortran code into a high-level predicate language. Mariano et al. [Mariano et al. 2020] use type-directed program synthesis to generate functional summaries of loops in smart contracts.

In addition to synthesis techniques, rule-based transpilation techniques have also been proposed. Gyori et al. [Gyori et al. 2013] suggest rule-based translation of imperative Java to the functional Stream API while Khatchadourian et al [Khatchadourian et al. 2020] propose a similar method for automatically translating sequential Java streams into parallel streams. Similarly, MOLD [Radoi et al. 2014] uses rules to automatically translate Java programs to Apache Spark.

Another emerging approach to transpilation is based on machine translation, and such techniques have been used to translate from Java to C# [Koehn et al. 2007], Python2 to Python3 [Aggarwal

[7]We contacted the authors about comparing our tool with Casper, however, their tool is no longer in repair.

et al. 2015], and CoffeScript to JavaScript [Chen et al. 2018]. Recently, Transcoder [Lachaux et al. 2020] has shown great success in this space by using large pre-trained models for learning a variety of transilation tasks. To the best of our knowledge, we are the first to propose a hybrid technique that combines a neural component for guiding the search and a pruning component that leverages intermediate values.

MapReduce Synthesis. The problem solved in this work is related to the problem of MapReduce synthesis which attempts to automatically generate parallelized programs. Raychev et al. [Raychev et al. 2015] propose a technique for automatically parallelizing imperative user-defined aggregations (UDAs) using symbolic execution. Farzan and Nicolet [Farzan and Nicolet 2017] also address the problem of UDA parallelization, in their case by synthesizing joins which combine the results of executing portions of the UDA in parallel. In general, these approaches are limited to UDAs and are thus not applicable to the more general imperative loops considered in this project. Radoi et al. [Radoi et al. 2014] propose a technique for translating sequential loops into MapReduce programs via a reduction to the lambda calculus and then a series of rewrites. Similar to [Gyori et al. 2013], this approach relies on a set of hand-written rules while ours does not. Smith and Albarghouthi [Smith and Albarghouthi 2016] propose a technique for synthesizing MapReduce programs from input-output examples. Their approach requires a small fixed set of higher-order sketches (only 8 in their evaluation) which they use to significantly reduce the search space; defining such a set of sketches in our context is not feasible given the varied nature of the benchmarks we consider.

Pruning and Program Synthesis. A number of existing synthesis tools use top-down and/or bottom-up reasoning to prune the search space. Synquid [Polikarpova et al. 2016] uses a combination of top-down and bottom-up reasoning to propagate refinement type constraints to unknowns during program search. Similarly, STUN [Alur et al. 2015] uses a combination of top-down and bottom-up reasoning to decompose a single synthesis problem into multiple subproblems whose solutions are unified together. λ^2 [Feser et al. 2015] leverages a top-down deductive approach for inferring input-output examples of missing subexpressions. However, none of these approaches incorporate reasoning based on intermediate values via concolic execution as our technique does.

Copy Mechanisms. As discussed in Section 5, the CGN is an extension of the ASN [Rabinovich et al. 2017] with a copy mechanism [Jia and Liang 2016; See et al. 2017] for copying components from the input to the output. Copy mechanisms have been effectively used in a number of settings, including document summarization [See et al. 2017], dialogue generation [Gu et al. 2016], and have even been used in the context of program synthesis for copying string variables [Jia and Liang 2016] and predicting out-of-vocabulary tokens [Li et al. 2018]. However, as far as we are aware, we are the first to suggest using copy mechanisms for copying large subtrees of ASTs.

Neurosymbolic Program Synthesis. In recent years, neurosymbolic techniques have become quite popular. Notable applications of neurosymbolic program synthesis include program correction [Bhatia et al. 2018], compositional rule-based translation [Nye et al. 2020], rediscovering classic physics laws [Ellis et al. 2020], scraping website information [Chen et al. 2021], and learning regular expressions [Ye et al. 2020]. As far as we are aware, we are the first to apply neurosymbolic synthesis techniques to the problem of imperative-to-functional API translation and the only to propose symbolic reasoning based on intermediate values via concolic execution.

10 CONCLUSION AND FUTURE WORK

We have proposed a new technique, and its implementation in a tool called NGST2, for translating imperative code to functional variants. Our method is based on the assumption that the overwhelming majority of source/target programs are *trace-compatible*, meaning that low-level expressions

are not only syntactically shared but also take the same values in a pair of corresponding program traces. Our approach leverages this assumption to (1) design a new neural architecture called *cognate grammar network (CGN)* to effectively guide transpilation and (2) develop bidirectional type checking rules that allow pruning partial programs based on intermediate values that arise during a computation. We have evaluated our approach against several baselines and ablations in the context of translating imperative Java and Python code to functional APIs targeting these languages. Our results show that the proposed techniques are both useful and necessary in this context. More broadly, while NGST2 has only been evaluated in the context of imperative-to-functional translation, our techniques are applicable in any domain satisfying the trace-compatibility assumption, which is likely to hold in other contexts as well. We leave it to future work to explore the applicability of our techniques (as well as their limitations) for other transpilation tasks.

ACKNOWLEDGMENTS

We would like to thank Benjamin Sepanski, Shankara Pailoor, and Jocelyn Chen for their thoughtful feedback. This material is based upon work supported by the National Science Foundation under grant numbers CCF-1908494, CCF-1811865, CCF-1918889, DARPA under the HARDEN program, Google under the Google Faculty Research Grant, as well as both Intel and RelationalAI.

REFERENCES

Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using machine translation for converting Python 2 to Python 3 code.* Technical Report. PeerJ PrePrints. https://doi.org/10.7287/peerj.preprints.1459v1

Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data.* ACM, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification.* Springer, New York, NY, USA, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. 2015. Synthesis through unification. In *International Conference on Computer Aided Verification.* Springer, New York, NY, USA, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the International Conference on Learning Representations (ICLR).* International Conference on Learning Representations (ICLR), La Jolla, CA, USA, 1–15.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. https://doi.org/10.48550/arXiv.1611.01989 arXiv:1611.01989

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/3453483.3454047

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. arXiv:1802.03691

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14. https://doi.org/10.1145/2499370.2462180

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547. https://doi.org/10.1093/logcom/2.4.511

Patrick Cousot and Radhia Cousot. 1994. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94).* IEEE, New York, NY, USA, 95–112. https://doi.org/10.1109/ICCL.1994.288389

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. https://doi.org/10.48550/arXiv.2006.08381 arXiv:2006.08381

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. *ACM SIGPLAN Notices* 52, 6 (2017), 540–555. https://doi.org/10.1145/3140587.3062355

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435. https://doi.org/10.1145/3296979.3192382

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. https://doi.org/10.1145/2813885.2737977

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1631–1640. https://doi.org/10.18653/v1/P16-1154

Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 543–553. https://doi.org/10.1145/2491411.2491461

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Paul Hudak and Jonathan Young. 1991. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 269–290. https://doi.org/10.1145/103135.103139

Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 12–22. https://doi.org/10.18653/v1/P16-1002

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/10.1145/2980983.2908117

Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. 2020. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Science of Computer Programming* 195 (2020), 102476. https://doi.org/10.1016/j.scico.2020.102476

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*. International Conference on Learning Representations (ICLR), La Jolla, CA, USA, 1–15.

Nikita Kitaev and Dan Klein. 2018. Constituency Parsing with a Self-Attentive Encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 2676–2686. https://doi.org/10.18653/v1/P18-1249

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*. Association for Computational Linguistics, La Jolla, CA, USA, 177–180.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. https://doi.org/10.48550/arXiv.2006.03511 arXiv:2006.03511

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, New York, NY, USA, 4159–4165. https://doi.org/10.24963/ijcai.2018/578

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, 1412–1421. https://doi.org/10.18653/v1/D15-1166

Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Isil Dillig. 2022. Automated Transpilation of Imperative to Functional Code using Neural-Guided Program Synthesis (Extended Version). https://doi.org/10.48550/arXiv.2203.09452 arXiv:2203.09452

Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K Lahiri, and Isil Dillig. 2020. Demystifying Loops in Smart Contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 262–274.

Maxwell I Nye, Armando Solar-Lezama, Joshua B Tenenbaum, and Brenden M Lake. 2020. Learning compositional rules via neural program synthesis. https://doi.org/10.48550/arXiv.2003.05562 arXiv:2003.05562

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 28)*, Sanjoy Dasgupta and David McAllester (Eds.). PMLR, Atlanta, Georgia, USA, 1310–1318. http://proceedings.mlr.press/v28/pascanu13.html

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538. https://doi.org/10.1145/2980983.2908093

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. https://doi.org/10.48550/arXiv.1704.07535 arXiv:1704.07535

Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, New York, NY, USA, 909–927. https://doi.org/10.1145/2660193.2660228

Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/2815400.2815418

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1073–1083. https://doi.org/10.18653/v1/P17-1099

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272. https://doi.org/10.1145/1095430.1081750

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340. https://doi.org/10.1145/2980983.2908102

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1250734.1250754

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. https://doi.org/10.48550/arXiv.1409.3215 arXiv:1409.3215

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2020. Optimal Neural Program Synthesis from Multimodal Specifications. https://doi.org/10.48550/arXiv.2010.01678 arXiv:2010.01678