AutoPandas: Neural-Backed Generators for Program Synthesis

ROHAN BAVISHI, University of California Berkeley, USA
CAROLINE LEMIEUX, University of California Berkeley, USA
ROY FOX, University of California, Irvine, USA
KOUSHIK SEN, University of California Berkeley, USA
ION STOICA, University of California Berkeley, USA

Developers nowadays have to contend with a growing number of APIs. While in the long-term they are very useful to developers, many modern APIs, with their hundreds of functions handling many arguments, obscure documentation, and frequently changing semantics, have an incredibly steep learning curve. For APIs that perform data transformations, novices can often provide an I/O example demonstrating the desired transformation, but are stuck on how to translate it to the API. A programming-by-example synthesis engine that takes such I/O examples and directly produces programs in the target API could help such novices. Such an engine presents unique challenges due to the breadth of real-world APIs, and the often-complex constraints over function arguments. We present a generator-based synthesis approach to contend with these problems. This approach uses a *program candidate generator*, which encodes basic constraints on the space of programs. We introduce neural-backed operators which can be seamlessly integrated into the program generator. To improve the efficiency of the search, we simply use these operators at non-deterministic decision points, instead of relying on domain-specific heuristics. We implement this technique for the Python pandas library in AutoPandas. AutoPandas supports 119 pandas dataframe transformation functions. We evaluate AutoPandas on 26 real-world benchmarks and find it solves 17 of them.

CCS Concepts: • **Software and its engineering → Programming by example**; **Automatic programming**;

Additional Key Words and Phrases: pandas, python, program synthesis, programming-by-example, generators, graph neural network

ACM Reference Format:
Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2018. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 27 pages.

1 INTRODUCTION

Developers nowadays have to contend with a growing number of APIs. Many of these APIs are very useful to developers, increasing the ease of code re-use. API functions provide implementations of functionalities that are often more efficient, more correct, or produce better-looking results than what the developer can implement. This increases the productivity of developers overall by providing functions that would be time-consuming to write from scratch, either because the function has very complex logic, or because the developer has a non-traditional programming background (e.g. data science).

Authors' addresses: Rohan Bavishi, Computer Science Division, University of California Berkeley, USA, rbavishi@cs.berkeley.edu; Caroline Lemieux, Computer Science Division, University of California Berkeley, USA, clemieux@cs.berkeley.edu; Roy Fox, Department of Computer Science, University of California, Irvine, USA, royf@uci.edu; Koushik Sen, Computer Science Division, University of California Berkeley, USA, ksen@cs.berkeley.edu; Ion Stoica, Computer Science Division, University of California Berkeley, USA, istoica@cs.berkeley.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Unfortunately, in the short to medium run, figuring out how to use a given API can hamper developer productivity. Many new APIs are wide, with hundreds of functions, some of which have overlapping semantics. Further, each function can have tens of arguments governing its behavior. For example, some Python APIs such as NumPy use the flexible type system to define almost entirely different behavior for functions based on the type or arguments. The documentation of all of these factors is of varying quality. Further, modern APIs are frequently updated, so tutorials, blog posts, and other external resources on the API can quickly fall out of date. All these factors increase the difficulty of API use for the developer.

However, often times when trying to use an APIs to conduct *data transformation*, novice developers have an idea of what transformation they want to perform. The popularity of online help forums such as StackOverflow has normalized the practice of creating an *input-output* (I/O) example that clearly illustrates the transformation. By I/O examples of a transformation, we mean examples where the input is the data before the transformation (e.g. the string "Garbledy Goop"), and the output is the desired transformed data (e.g. "Goop, G.").

When the novice developer can provide such an I/O example, programming-by-example based synthesis is a compelling solution to the problem of finding a program in the API that performs the transformation. Programming-by-example refers to a class of program synthesis tools that use an I/O example as the specification to which the synthesized program must adhere [Feng et al. 2017; Gulwani 2011; Polozov and Gulwani 2015; Yaghmazadeh et al. 2017]. Existing programming-by-example engines have been highly successful in synthesizing string transformations typically used in spreadsheets [Gulwani 2011], where the engine has been integrated into Microsoft Excel. However, in most of these cases, the language in which the synthesized programs are written only consists of tens of operators or functions. This is far below the number of functions in data transformation APIs. The Python library `pandas`, which provides an API for dataframe transformations, has hundreds of functions just operating on dataframes.

Beyond the sheer number of functions in the API, finding the correct arguments for a given function is a challenge. API functions often place constraints on the arguments beyond type. In Python, they can also accept multiple types for a single argument, with different constraints for each type. A synthesis engine with no knowledge of these constraints will have a difficult time creating runnable programs.

Based on these observations, we propose a generator-based program synthesis technique. At its core, the technique utilizes a *program candidate generator*, which yields a different well-formed program in the target API each time it is called. A simple way to build a synthesis engine is then to repeatedly call the candidate program generator until it produces a program p such that $p(input) = output$ for the I/O example at hand.

The program candidate generator must encode expert domain knowledge about the API to—as much as possible—synthesize programs in the API which are *valid*. For example, when producing arguments to a function, the generator should almost never produce argument combinations which cause the function to immediately error out. Given knowledge of the API, writing such a candidate program generator is a straightforward—although perhaps tedious—effort. This means such a generator can be written by any developer who knows the API, regardless of their familiarity with program synthesis techniques.

However, if the generator takes a long time to generate a p such that $p(input) = output$, our simple synthesis engine—which just calls the generator until it yields such a p—becomes impractical. To make the generator more likely to yield such a p in reasonable time, an API expert could work with a program synthesis expert to build heuristics that prioritize more program candidates that are more likely to pass the test. Building such heuristics is extremely *tedious*, *error-prone*, *non-scalable*, and requires a long process of trial-and-error.

Our *key insight* is the following. Many of the choices which would be governed by heuristics in these generators are choices between different elements in a given domain. For example, which column name of a table to use in an argument. Instead of requiring the developer of the generator to write sophisticated heuristics over which element in the domain should be chosen, we can provide a *smart operator* which uses a *learned probabilistic model* to make the choice over the domain which is most likely, given the I/O example. Thus, the developer of the generator can write only the constraints on the program search space of which they are sure, such as constraints on the argument space of a function. The fuzzier decisions can be left to smart operators over the domain.

Our smart operators allow users to specify a selection over a given domain (i.e., select one or more elements from this set). The probabilistic models that back them thus only need to be trained for a specific selection task (i.e., given this context, which element should be selected from the set). This is in contrast to the use of probabilistic models in past work such as that of Lee et al. [2018] and Devlin et al. [2017] where these models are trained over the full language. This is not feasible for our target domain as we are dealing with large real-world APIs without designing DSLs or considering small subsets of the API.

In this paper, we propose this smart generator model approach to program synthesis for APIs. We introduce novel graph-neural-network based neural backends for 4 key *smart operators* over a domain. For example, we provide operators to select a single element, as well as a subset or sequence of elements, from a given domain. These operators can be seamlessly integrated with arbitrary Python code in the program candidate generators.

We evaluate the viability of our proposed approach with an implementation for the Python `pandas` API for dataframe manipulation. We chose `pandas` as a subject because it presents interesting new research challenges; a broad number of functions, each taking multiple arguments with dependencies, and transforms data with very complex structure (dataframe, i.e. tables). In addition, we chose `pandas` because it is a prominent and widely-used library for data manipulation in Python, used to pre-process data for statistical and machine learning application, as well as for data science in its own right. Our implementation, AutoPandas, supports 119 `pandas` operations on dataframes. We hand-wrote a program candidate generator that supports all these functions. This generator encodes, in native Python, the pre-conditions for each supported `pandas` API.

In implementing AutoPandas, we also devise a novel encoding of dataframes as graphs in our operators' neural backends. Currently, this encoding is specific to key `pandas` data-structures (dataframes, series, lists). However, graphs could be used to encode many other data structures, including: matrices (similar to dataframes), strings (each node is a character), lists, trees, maps, and objects (each object is a node and each field denotes an outgoing edge). The graph-neural-network based backends for our smart operators assume only that their input is a graph, and thus can ingest such encodings.

Overall we find that AutoPandas can efficiently solve 17 of our 26 complex real-world `pandas` benchmarks. In particular, AutoPandas can solve 4 out of 10 benchmarks requiring sequences of 3 function calls. We also analyze the accuracy of our smart operators. We find the accuracy of the smart operators to be quite high (Section 4.4) for argument prediction compared to deterministic and randomized semantics. However, while the accuracy of the smart operators to predict function sequences is much higher than random, it decreases as the function sequence length increases, highlighting room for improvement.

In summary, we make the following contributions:

- We propose a *smart generator* approach to synthesizing programs in large, real-world APIs (Sections 3.1, 3.2). Smart generators combine validity constraints on the space of programs in the API (as described by documentation), with learned probabilistic models that guide

	Date	Category	Location	Expense	Balance
0	2018-02-18	Social	Terrace	98.34	9971.66
1	2018-02-18	Lunch	Pox	245.63	9726.03
2	2018-02-24	Social	Gate 320	121.89	9604.14
3	2018-02-24	Lunch	Pox	248	9356.04

(a) An example input DataFrame.

	Lunch	Social
2018-02-18	245.63	98.34
2018-02-24	248	121.89

(b) Desired output.

Fig. 1. A DataFrame input-output example.

arbitrary choices within these constraints. These generators can be written in existing languages such as Python.

- To build these smart generators, we introduce a set of arbitrary-choice *smart operators* with *neural backends*. These smart operators seamlessly integrate with the execution of arbitrary Python code in the generators, making choices by generating neural network queries on the fly (Section 3.3.2). We additionally design novel graph neural network models for each of these smart operator backends (Section 3.3.3). These graph-based models support generators operating on complex data such as dataframes.
- We implement AutoPandas, a smart-generator based synthesis engine for the Python pandas API for dataframe transformations (Section 4). AutoPandas supports 119 pandas functions, an order of magnitude more functions than considered in prior work. AutoPandas is released as open-source.[1] AutoPandas can efficiently solve 17 out of 26 of complex real-world benchmarks using the pandas library.

2 MOTIVATION

In order to motivate our problem statement and technique, consider the following scenario. Suppose a developer or data scientist, new to the Python library pandas, needs to use pandas in order to pre-process data for a statistical or ML pipeline. This pandas novice needs to transform some input data—a pandas `DataFrame` read from a CSV file—into a different form for the target pipeline.

For example, suppose the input data is an expense table, represented by the dataframe in Figure 1a. For input into the pipeline, it needs to be transformed slightly into the dataframe in Figure 1b. For a small input table, the novice can do this by hand. But in order to scale, they need a re-usable program in the pandas API that performs this transformation.

Our goal is to automatically solve *exactly* such problems. In this paper, we propose a generator-based program synthesis technique for data transformation APIs. We evaluate its viability for the pandas API by implementing our tool AutoPandas. Given a dataframe transformation represented by an (`input`, `output`) example like the one in Figure 1, AutoPandas outputs a program p which performs the transformation, i.e., such that $p(\texttt{input}) = \texttt{output}$.

Before we dive into the details of AutoPandas, however, let us return to the simpler problem at hand. Again, a novice to the pandas API is trying to write a program that performs the transformation in Figure 1. Consider a simpler scenario, where, while the novice does not know pandas, they know some other data transformation tools. In particular, the novice knows that in Microsoft Excel, they can perform the transformation in Figure 1 using the "pivot table" functionality. The novice also notices that pandas has a `pivot` function for dataframes, and thinks they can use it.

But, even when knowing which function in pandas to use, the novice is faced with the issue of understanding what all the arguments to `pivot` *do*. For complex APIs like pandas, there are many arguments for each function, requiring substantial effort to master even one function. Resources

[1] https://github.com/rbavishi/autopandas

```
1 def find_pivot_args(input_df: pandas.DataFrame, output_df:pandas.DataFrame):
2     while True:
3         cur_kwargs = generate_pivot_args(input_df, output_df)
4         cur_out = pandas.DataFrame.pivot(input_df, **cur_kwargs)
5         if cur_out == output_df:
6             return cur_kwargs
```

Fig. 2. A procedure to find the arguments to the pandas function `pivot` that turn `input_df` into `output_df`.

explaining all the complexity of the API can overwhelm a novice wanting to perform a single transformation.

Hence, novices often resort to asking experts for help on which arguments to use. Unfortunately, this is not a perfect solution. First, if no expert is around to answer the question, a novice can get stuck on the problem for a long time. Also, the expert finds themselves constantly answering a very similar question—what `pivot` arguments should be used to get `output` from `input`? Answering this question over and over again is not scalable. This issue is not imaginary: in reality, if a basic `pivot` or `merge` question is asked on pandas StackOverflow, it is not answered, and simply marked as a duplicate of a master answer diving into the details of these functions. At the time of writing, these master answers had 350 and 246 duplicates, respectively. [2]

An alternative to redirecting the novice to the documentation would be for the API expert to write a program that outputs valid argument combinations for `pivot` on the dataframe `df`, say `generate_pivot_args(df)`. In particular, `generate_pivot_args(df)` is a *generator* that, every time it is called, yields a different valid argument combination. The novice can then use `generate_pivot_args(df)` to enumerate the argument combinations, and save the one that works for their input-output example. Figure 2 shows pseudo-code to find the correct arguments for `pivot`, given the generator `generate_pivot_args(df)`. The code simply calls `generate_pivot_args(df)` (Line 3) iteratively until it returns an argument combination `kwargs` such that `pivot(input_df, **kwargs) == output_df` (Line 5).

To make sure that all the argument combinations returned by `generate_pivot_args(df)` are valid, the expert can implement the function to encode the basic constraints on the arguments required by pandas. Namely, for `pivot` these constraints are:

(1) `arg_col` should be selected from the list of column names of `df`, `df.columns`.
(2) `arg_idx` is either `None`, or selected from the list of column names of `df`, except from the column name used in `arg_col` (`df.columns-{arg_col}`)
(3) Finally, the `arg_val` argument should either be (1) selected from the list of column names except for the ones used in `arg_col` and `arg_idx`, or (2) `None`, in the case where `arg_idx` is `None` and `df` has a multi-level index.

These constraints are universal for the `pivot` function, and an expert can straightforwardly derive them from the documentation.

Figure 3 shows the implementation of `generate_pivot_args(df)`. The calls to **`Select`**`(D,c,i)` return a single element from the domain `D`. To understand Figure 3, assume first that **`Select`**`(D,c,i)` just returns a random element from `D`. We give formal semantics for **`Select`** and explain the arguments `c` and `i` in Section 3.1. Essentially, these calls to **`Select`** allow `generate_pivot_args(df)` to cycle through different argument combinations.

[2]The current number can be determined with the query at https://data.stackexchange.com/stackoverflow/query/edit/1024223.

```
@generator
def generate_pivot_args(input_df: pandas.DataFrame, output_df: pandas.DataFrame):
    context = (input_df, output_df)
    arg_col = Select(df.columns, context, id=1)
    arg_idx = Select({None} | df.columns − {arg_col}, context, id=2)
    if isinstance(df.index, pandas.MultiIndex) and arg_idx is None:
        arg_val = None
    else:
        arg_val = Select(df.columns − {arg_col, arg_idx}, context, id=3)

    return {'columns': arg_col, 'index': arg_idx, 'values': arg_val}
```

Fig. 3. A generator of all valid arguments to the `pivot` function from the pandas API. `Select(D,c,i)` returns a single element from the domain `D`, according to the semantics in Figure 4.

However, there is still a problem. If there are many argument combinations, the basic search in Figure 2 may take some time to terminate. The problem gets worse if the exact function to use is not clear. The novice may be unsure whether to use `pivot`, `pivot_table`, `unstack`, etc., and in that case would have to go through the argument combinations for each of these functions. If the order in which `generate_pivot_args` returns arguments is arbitrary, the correct argument combination is unlikely to show up early enough for the code in Figure 2 to be really practical. The problem is exacerbated if sequences of multiple functions are required to perform the transformation, as the total number of possible argument combinations grows exponentially.

To make `generate_pivot_args` output the correct argument combination more quickly, the API expert *could* replace the calls to `Select(D,c,i)` with a particular enumeration order through `D`. The enumeration order would be based on some additional *heuristics*, for example:

(1) The values in the column from `input` that is used as `arg_col` end up being column names in the `output`. Therefore, the generator should look at the output's column names, and first try to use as `arg_col` any column from the input that shares values with the output's column names.
(2) The values in the column from `input` that is used as the `arg_val` argument end up in the main data of the table. Hence, the generator should look at the output's data, and first try to use as `arg_val` any column whose values are the same as output's data cells. However the values argument also accepts `None` as a valid argument, in which case all the remaining column values are used as the main data of the output. Therefore the generator should take this into account as well.
(3) ... *(more heuristics omitted)*

Designing such heuristics is error-prone. They are not guaranteed to be effective, especially if the I/O example provided by the user cannot actually be solved with a single call to `pivot`. Further, it is much more tedious for the expert to write a generator that uses these heuristics than it is to write a generator that encodes the basic validity constraints, like that in Figure 3. Overall, using heuristics is an *error-prone*, *tedious*, and *non-scalable* way to more quickly find the correct argument combination.

In this paper, we propose a different route. Instead of relying on humans to write more heuristics, we propose a smart backend for operators like `Select(D,c,i)`. This smart backend for `Select` first derives from the context `c` a probability distribution p over `D`. Then, it returns elements $d \in$ `D`

in descending order of their probability $p(d)$. The distribution model can be represented by a neural network, and learned from a training set of inputs, programs, and their outputs, as detailed in Section 3.4. Over a validation set of (`input`, `output`) pairs where `output = pivot(input, **kwargs)` for some arguments `kwargs`, our smart backend has 99% top-1 accuracy in retrieving the correct `kwargs`.

Further, instead of using the smart backends only to generate arguments for `pivot`, we use them to build a prototype synthesis engine for the `pandas` library, AutoPandas. AutoPandas takes in a pair of `(inputs, output)` representing a dataframe transformation and outputs a program `p` in the `pandas` API such that `p(input) == output`. We achieve this by (1) implementing a *program candidate generator* which outputs straight-line `pandas` programs that run without error on `input` and (2) using smart backends for **`Select`** and other operators so that the program candidate generator outputs `p` such that `p(input) == output` early in the search. AutoPandas currently supports 119 `pandas` functions and can form programs with multiple function calls. Given the I/O example in Figure 1, AutoPandas finds the correct program:

```
output_df = input_df.pivot(index='Date', columns='Category', values='Expense')
```

after checking only *one* program candidate.

In the next section, we formalize (1) generators, and the semantics of **`Select`** and other operators, (2) generator-based synthesis, and (3) the smart backend we use to synthesize `pandas` programs.

3 TECHNIQUE

3.1 Generators

We first formally describe *generators*. In our setting, a generator $\mathcal{G}$ is a program that, when invoked, outputs values from a space of possible values. Figure 3 shows an example of such a generator $\mathcal{G}$ for function arguments in the Python `pandas` library [pan 2014] for DataFrame (i.e. table) manipulation. In particular, the generator in Figure 3 takes a `pandas` `DataFrame` as an input, and returns one of the possible argument combinations of the `pandas` API function `pivot`, by selecting various argument values.

Our generators $\mathcal{G}$ can contain arbitrary Python code, along with a set of stateful operators that govern the behavior of $\mathcal{G}$ across runs. An example of such an operator is **`Select`**, which is also used in the generator in Figure 3. Given a collection of values, **`Select`** returns a single value from the collection. For example, the call to **`Select`** in Line 4 selects one of the columns of the input dataframe `df`. The generator then assigns this value to `arg_col`, to be used as the pivot column. Similarly, the call to **`Select`** in Line 5 picks either `None` or one of the columns in `df` *except* the one selected as `arg_col` (i.e., `df.columns − {arg_col}`), to be used as the index. Choosing `arg_val` is more complicated. In particular, if the input dataframe has a multi-level index, and `arg_idx` is `None`, `arg_val` must be `None` for `pivot` to not throw an error. Generators are a natural form in which to specify such contracts. The checks in place in Figure 3 ensure that the generator only generates arguments that follow the contract, and thus, can be given to `pivot` without error.

On different invocations of the generator in Figure 3, the calls to **`Select`** may yield different values. There are a few different ways in which **`Select`** can do this. First, it can simply randomly choose values from `D`. Or, it can assure new values will be different by maintaining internal state which records the values it returned in previous runs. Further, it can use its context argument `c` to determine the order in which it returns these values. We elaborate on this below.

Operators. Apart from **`Select`**, we support three other operators, namely (1) **`Subset`**, (2) **`OrderedSubset`** and (3) **`Sequence`**. An informal description of their behavior is provided in Table 1, while a formal treatment is presented in Figure 4.

Each operator Op is of the form $Op(\mathcal{D}, C, id)$ where $\mathcal{D}$ is the domain passed to the operator; C is the context passed to the operator to control its behavior; and id is the unique static ID of the operator. The static ID of Op simply identifies each call to an operator uniquely based on its static program location. It is provided explicitly in Figure 3 for clarity but may be inserted automatically via a static instrumentation pass of the generator code. The behavior of the generator across runs can be controlled by changing the semantics of these operators, some of which are described below.

Randomized. The simplest case is for the generator to be *randomized*. That is, the generator will follow a random execution path as governed by the values returned by its constituent operator calls. This can be achieved by simply randomizing the underlying operators, the semantics of which are given in Figure 4c. These semantics are rather straightforward — each operator simply returns a random element from the collection of possible values (defined by $\mathcal{W}$, the definition of which is given in Figure 4a). This collection of possible values $\mathcal{W}$ is a function of the operator type (one of { `Select`, `Subset`, `OrderedSubset`, `Sequence` }) and the domain $\mathcal{D}$ passed to the operator call.

Exhaustive (Depth-First). Another option is to have an *exhaustive* generator which systematically explores all possible execution paths as governed by the constituent operator calls. That is, all the operators work in unison by returning a fresh combination of values on each invocation of the generator. The operators also signal *Generator-Exhausted* when all possible values have been explored. Figure 4d presents the operator semantics that achieve this behavior. In particular, the semantics in Figure 4d enforce a *depth-first exhaustive* behavior across runs, where the generator explores all possible values of operator calls occurring later in the execution trace of the generator before exploring the ones occurring before. For example, when using the semantics in Figure 4d, the generator in Figure 3 will first explore all possible values of the `Select` call at Line 9 before moving on to the next possible value for the `Select` call at Line 5.

The operator semantics in Figure 4d uses three internal state variables t, σ and δ. The variable t keeps track of the number of operator calls made in the *current* invocation of the generator. The variable σ is a map from the operator call index to the choice to be made by the operator call in the current invocation of the generator. Note that the operator call index is distinct from the static identifier id as it keeps track of the *dynamic* index of the operator call in the generator call stack. For example, if an operator at a program location is called twice as a result of an enclosing loop, it will have two distinct entries in σ. Finally, δ represents a map from the operator call index to the collection of possible values $\mathcal{W}$ as defined by the operator type and the passed domain $\mathcal{D}$. The variables σ and δ are initialized to empty maps before the first invocation of the generator, but are persisted across the later ones. However t is reset to zero before every fresh generator invocation. We also introduce a special operator called Op_{End} that is implicitly invoked at the end of each invocation in the generator. We now briefly explain the rationale behind all of these variables, Op_{End} and the rules themselves.

(1) Op-Extend - This rule governs the behavior of the operator when it is invoked for the *first time* (as signified by $t \notin dom(\sigma)$). The operator returns the first value from $\mathcal{W}$ and records this choice in σ. It also stores $\mathcal{W}$ in δ for future use.
(2) Op-Replay - The hypothesis $t \in dom(\sigma)$ signifies that this operator call needs to *replay* the choice as dictated by $\sigma(t)$.
(3) Op-End-1 - This rule captures the first behavior of the special operator Op_{End}. It finds the last (deepest) operator call, indexed by k, that has not exhausted all possibilities, and increments its entry in σ. This is key to enforcing depth-first semantics - a later call explores all possibilities before previous calls do the same. Note that it also *pops-off* the later entries ($> k$) from σ and δ. This is required as the generator may take an entirely new path based on the new value returned by this operator and may therefore make an entirely new set of operator calls.

Operator	Description
`Select(domain)`	Returns a single item from `domain`
`Subset(domain)`	Returns an unordered subset, without replacement, of the items in `domain`
`OrderedSubset(domain)`	Returns an ordered subset, without replacement, of the items in `domain`
`Sequence(len)(domain)`	Returns an ordered sequence, with replacement, of the items in `domain` with a maximum length of len

Table 1. List of Available Operators

Together, these two steps maintain the invariant that σ stores the choice to be made by the operators in the current generator run.

(4) Op-End-2 - The final rule covers the case when all operator calls have exhausted all possible values. This makes the special Op_{End} operator signal `Generator-Exhausted` after the last invocation of the generator, indicating that we have explored all possible executions of the generator.

Smart Semantics. Notice that the semantics presented in Figures 4c and 4d do not utilize the context C or the static id passed to the operator. The significance of this is that given the same domain, the behavior of the operator is *fixed*, regardless of the actual input with which the generator is invoked. This is not suitable for tasks such as the one presented in Section 2, where the goal is to quickly find an argument combination to the `pivot` function such that when it is called on `input_df`, it produces the target output `output_df`. In this case, we want to change the behavior of the operators based on the input and output dataframe, and bias it towards the values that have a higher probability of guiding the generator execution in the right direction.

This *smart* behavior of operators is captured in the semantics presented in Figure 4e. The only difference with the semantics in Figure 4d is that the set of possible values $\mathcal{W}_M$ for each operator is now the result of a function $Rank_{(Op,id)}$ that takes in the original collection of possible values, the passed domain $\mathcal{D}$ as well as the context C passed to the operator, and reorders the values in the decreasing order of significance w.r.t to the task at hand. Note that the *Rank* function is sub-scripted by (Op, id) implying that every operator call can have a separate ranking function.

In the generator in Figure 3, the context passed at every operator call is the input and output dataframe. Therefore given suitable ranking functions $Rank_{(Select,1)}$, $Rank_{(Select,2)}$ and $Rank_{(Select,3)}$, the generator can be biased toward producing an argument combination that, when passed to the `pivot` function along with the input dataframe `input_df`, is likely to result in `output_df`.

3.2 Generator-Based Program Synthesis

We now describe how to build an *enumerative* synthesis engine $\mathcal{E}$ using generators. The input to $\mathcal{E}$ is an input-output (I/O) example. The result is a program in the target language that produces the output when run on the input given in the I/O example. Our target language is the python `pandas` API. Figure 5 describes the basic algorithm behind this engine in Python-like pseudo-code. The engine consists of two components — (1) a program candidate generator and (2) a checker that checks if the candidate program produces the correct output. The checker is rather straightforward to implement: we simply execute the program and test the exact match of its output to the target output. The bulk of the work is done by the program candidate generator.

3.2.1 Program Candidate Generator. A program candidate generator $\mathcal{P}$ is a generator that, given an input-output example, generates program candidates. First, assume $\mathcal{P}$ is a generator in *exhaustive* mode (see Section 3.1). That is, on each invocation, $\mathcal{P}$ yields a program candidate that hasn't

$$\mathcal{P}(\mathcal{D}) \stackrel{\text{def}}{=} \text{Power-Set of } \mathcal{D}$$

$$\text{Perms}(x) \stackrel{\text{def}}{=} \text{Set of all permutations of x}$$

$$W(Op,\ \mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{D} \text{ if } Op = \textit{Select} \\ \mathcal{P}(\mathcal{D}) \text{ if } Op = \textit{Subset} \\ \cup\{\text{Perms}(x) \mid x \in \mathcal{P}(\mathcal{D})\} \text{ if } Op = \textit{OrderedSubset} \\ \{(a_1, \cdots, a_k) \mid k \leq l,\ a_i \in \mathcal{D}\} \text{ if } Op = \textit{Sequence}(l) \end{cases}$$

$$\mathcal{R}(\mathcal{W}) \stackrel{\text{def}}{=} \text{Random Element from } \mathcal{W}$$

(a) Common Definitions

$$\mathcal{W} \stackrel{\text{def}}{=} W(Op,\ \mathcal{D})$$

$$\sigma_k \stackrel{\text{def}}{=} \forall t.((t < k) \Rightarrow (\sigma_k(t) = \sigma(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin dom(\sigma_k))$$

$$\delta_k \stackrel{\text{def}}{=} \forall t.((t < k) \Rightarrow (\delta_k(t) = \delta(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin dom(\delta_k))$$

$$\mathcal{W}_M \stackrel{\text{def}}{=} Rank_{(Op,id)}(W(Op,\ \mathcal{D}),\ \mathcal{D},\ C)$$

(b) Common Definitions (continued)

$$\frac{}{Op(\mathcal{D},\ C,\ id) \Downarrow \mathcal{R}(\mathcal{W})}\ \textsc{Op-Random}$$

(c) Operator Semantics - Randomized

$$\frac{t \notin dom(\sigma) \qquad \delta' \equiv \delta[t := \mathcal{W}] \qquad \sigma' \equiv \sigma[t := 0]}{\langle Op(\mathcal{D},\ C,\ id),\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \mathcal{W}[0],\ \sigma',\ \delta',\ t+1\rangle}\ \textsc{Op-Extend}$$

$$\frac{t \notin dom(\sigma) \qquad \delta' \equiv \delta[t := \mathcal{W}_M] \qquad \sigma' \equiv \sigma[t := 0]}{\langle Op(\mathcal{D},\ C,\ id),\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \mathcal{W}_M[0],\ \sigma',\ \delta',\ t+1\rangle}\ \textsc{Op-Extend}$$

$$\frac{t \in dom(\sigma)}{\langle Op(\mathcal{D},\ C,\ id),\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \mathcal{W}[\sigma(t)],\ \sigma,\ \delta,\ t+1\rangle}\ \textsc{Op-Replay}$$

$$\frac{t \in dom(\sigma)}{\langle Op(\mathcal{D},\ C,\ id),\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \mathcal{W}_M[\sigma(t)],\ \sigma,\ \delta,\ t+1\rangle}\ \textsc{Op-Replay}$$

$$\frac{\begin{array}{c}\exists k.\ k \text{ is largest such that} \\ (k \in dom(\sigma) \wedge \sigma(k) < |\delta(k)| - 1)\end{array}}{\langle Op_{\text{End}},\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \sigma_k[k := \sigma(k)+1],\ \delta_k,\ t+1\rangle}\ \textsc{Op-End-1}$$

$$\frac{\begin{array}{c}\exists k.\ k \text{ is largest such that} \\ (k \in dom(\sigma) \wedge \sigma(k) < |\delta(k)| - 1)\end{array}}{\langle Op_{\text{End}},\ \sigma,\ \delta,\ t\rangle \Downarrow \langle \sigma_k[k := \sigma(k)+1],\ \delta_k,\ t+1\rangle}\ \textsc{Op-End-1}$$

$$\frac{\nexists k.\ (k \in dom(\sigma) \wedge \sigma(k) < |\delta(k)| - 1)}{\langle Op_{\text{End}},\ \sigma,\ \delta,\ t\rangle \Downarrow \text{Generator-Exhausted}}\ \textsc{Op-End-2}$$

$$\frac{\nexists k.\ (k \in dom(\sigma) \wedge \sigma(k) < |\delta(k)| - 1)}{\langle Op_{\text{End}},\ \sigma,\ \delta,\ t\rangle \Downarrow \text{Generator-Exhausted}}\ \textsc{Op-End-2}$$

(d) Semantics - Depth-First Exhaustive

(e) Semantics - *Smart* Depth-First Exhaustive

Fig. 4. Operator Semantics for Generators. σ and δ are initialized to empty maps before the first invocation of the generator. t is set to the integer zero before every invocation of the generator. Op_{End} is a special operator that is implicitly called at the end of each invocation of the generator. A detailed explanation is provided in Section 3.1

```
1 def synthesize(input, output, max_len):
2     generator = generate_candidates(input, output, max_len)
3     while (not generator.finished()):
4         candidate = next(generator)
5         if candidate(input) == output:
6             return candidate
```

Fig. 5. Generator-Based Enumerative Synthesis Engine

been produced so far. Figure 6 shows an excerpt of our program candidate generator for pandas programs. This generator produces straight-line programs, each of which is a sequence of up to max_len pandas function calls. The program given at the end of Section 2 is an example of such a candidate.

The generator in Figure 6 generates candidate programs as follows. First, it picks a sequence of functions from a list of supported functions (Lines 3-4). Then, for each function in the sequence, the generator selects the arguments (Lines 8-25), and computes the result by running the function

```
1 @generator
2 def generate_candidates(input, output, max_len):
3     functions = [pivot, drop, merge, ...]
4     function_sequence = Sequence(max_len)(functions, context=[input, output], id=1)
5     intermediates = []
6     for function in function_sequence:
7         c = [input, *intermediates, output]
8         if function == pivot:
9             df = Select(input + intermediates, context=c, id=2)
10            arg_col = Select(df.columns, context=[df, output], id=3)
11            arg_idx = Select(df.columns − {arg_col}, context=[df, output], id=4)
12            if isinstance(df.index, pandas.MultiIndex) and arg_idx is None:
13                arg_val = None
14             else:
15                arg_val = Select(df.columns − {arg_col, arg_idx}, context=[df, output], id=5)
16            args = (df, arg_col, arg_idx, arg_val)
17
18        elif function == merge:
19            df1 = Select(input + intermediates, context=c, id=6)
20            df2 = Select(input + intermediates, context=c, id=7)
21            common_cols = set(df1.columns) & set(df2.columns)
22            arg_on = OrderedSubset(common_cols, context=[df1, df2, output], id=8)
23            args = (df1, df2, arg_on)
24        # Omitted code: case for each function
                                          ⋮
25
26        intermediates.append(function.run(*args))
27
28    return function_sequence
```

Fig. 6. A Simplified Program Candidate Generator for pandas Programs.

with the arguments and stores it as an *intermediate* (e.g. Line 26). Intermediates are the outputs produced by previous functions in the sequence. These are essential to allow the generator to generate meaningful multi-function programs, where a function can operate on the output of a previously applied function.

As shown in Lines 3-4, argument generation is done on a case-by-case basis depending on the given function. For example, for the function `pivot`, the generator follows the argument generation logic of Figure 3, applies the function with the selected arguments to a selected input or intermediate `df`, and stores the output as an intermediate. The program candidate generator can handle `pandas` functions that operate on multiple dataframes, e.g. `merge` on Lines 18-23, by selecting each dataframe from the set of input and intermediates (Lines 19-20).

3.2.2 Building an Exhaustive Depth-First Enumerative Synthesis Engine. Using the exhaustive depth-first semantics for operators presented in Figure 4d for the generator in Figure 6 gives an exhaustive depth-first synthesis engine. This means that the engine explores all possible program candidates and in depth-first order i.e. it explores all possible programs using the same sequence of functions

before exploring another sequence. Also, when enumerating the arguments, it explores all values of a later argument before moving on to the next value for the previous argument.

3.2.3 Building a Smart Enumerative Synthesis Engine. The generator in Figure 6 describes a space of programs that is extremely large for such an enumerative pandas synthesis engine to explore in reasonable time. This generator supports 119 pandas functions, each taking 3 arguments on average. This causes an *enormous* combinatorial explosion in the number of argument combinations and choices made by the generator operators. Hence, we need a *smart* generator that tailors itself to the presented synthesis task. That is, we need to use the smart semantics for operators presented in Figure 4e. For the generator in Figure 6, the context passed to each operator call is explicitly shown. The function sequence selection, as well as the selection of dataframes on which the functions operate (Lines 4, 9, 19, 20) all take the input-output example along with any intermediates as the context. The operator calls used to select values for arguments depends primarily on the dataframe(s) on which the function will be run, so only that dataframe and the output is passed as context.

With this formulation in place, we can now define the $Rank_{(Op,id)}$ function that is at the heart of the semantics in Figure 4e. Given the domain $\mathcal{D}$ and the context C passed to Op, this function reorders the space of possible values $W(Op, \mathcal{D})$ according to a probability distribution over this space. We exploit the recent advances in the area of deep learning and define these *Rank* functions per operator using novel neural network models that we describe in the following section. We call generators that use operators backed by these neural network models *Neural-Backed Generators*.

3.3 Neural-Backed Generators for Pandas

In AutoPandas, we use neural networks to define the *Rank* functions for the operators used in our generators. In short, we design a neural network model for each kind of operator (see Table 1). The first time an operator Op is called with a particular domain $\mathcal{D}$ and context C, a query is constructed using $\mathcal{D}$ and C. This query is passed to the neural network model, which returns a probability distribution over the possible values for the operator (as defined by $W(Op, \mathcal{D})$ in Figure 4a). The *Rank* function then uses this distribution to reorder the elements in $W(Op, \mathcal{D})$ in the decreasing order of probabilities ($\mathcal{W}_M$ in Figure 4b). The operator functions as before, but now returns values in an order conditioned on the context. We now define the query concretely, its encoding as well as the neural network architectures for each operator.

3.3.1 Neural-Network Query. The query Q to each neural network model, regardless of the operator, is of the form $Q = (\mathcal{D}, C)$ where $\mathcal{D}$ and C are the domain and context passed to the operator.

3.3.2 Query Encoding. Encoding this query into a neural-network suitable format poses several challenges. Recall that the context and the domain passed to operators in the pandas program candidate generator (Figure 6) contain complex structures such as dataframes. Dataframes are 2-D structures which can contain arbitrary Python objects as primitive elements. Even restricting ourselves to strings or numbers, the set of possible primitive elements is infinite. This renders all common value-to-value map-based encoding techniques popular in machine learning, such as one-hot encoding, inapplicable. At the same time, the encoding needs to retain enough information about the context to generalize to unseen queries which may occur when the synthesis engine is deployed in practice. Therefore, simply abstracting away the exact values is not viable. In summary, a suitable encoding needs to (1) abstract away only irrelevant information and (2) be suitably structured for neural processing. To this end, we designed a graph-based encoding that possesses all these desirable properties. We describe the encoding below.

Graph-Based Encoding. We now describe how to encode the domain $\mathcal{D}$ and the context C as a graph, consisting of nodes, edges between pairs of nodes, and labels on nodes and edges. The overall rationale is that it is not the concrete values, but rather the *relationships* amongst values, that really encode the transformation at hand. That is, relationship edges should be sufficient for a neural network to learn from. For example, the essence of transformation represented by Figure 1 is that the values of the column 'Category' now become the columns of the pivoted dataframe, with the 'Date' column as an index, and the 'Expense' as values. The concrete names are immaterial.

Recall that the domain and context are essentially collections of elements. Therefore, we first describe how to encode each such element e individually as a graph G_e. Later we describe the procedure to combine these graphs into a single graph G_Q representing the graph-encoding of the full query Q. Figure 7 shows the graph-encoding of the query generated as a result of the `Select` call at line 4 in Figure 3 and will be used as a running example.

Encoding Primitives. If the element e is a primitive value (strings, ints, float, lambda, NaN etc.), its graph encoding G_e contains a single node representing e. This node is assigned a label based on the data-type of the element as well as the *source* of the element. The source of an element indicates whether it is part of the domain, if it is one of the input-outputs in the I/O example, if it is one of the intermediates, or none of these.

Encoding DataFrames. If the element e is a dataframe, each cell element in the dataframe is encoded as a node in the graph G_e. The label of the node includes the type of the element (string, number, float, lambda, NaN, etc.). The label also includes the source of the dataframe, i.e. whether the dataframe is part of the domain, input, output, intermediate, or none of these. We also add nodes to G_e that represent the schema of the dataframe, by creating a node for each row index and column name of the dataframe. Finally, we add a *representor* node to G_e that represents the whole of the dataframe. The label of this node contains the type "dataframe" as well as the source of the parent dataframe. Note that this additional representor node is not created when encoding primitive elements. The node representing the primitive element itself acts as its representor node.

The graph encoding of a dataframe also contains three kinds of edges to retain the structure of the dataframe. The first kind is adjacency edges. These are added between each pair of cell nodes, column name nodes or row index nodes that are adjacent to each other in the dataframe. We only add adjacency edges in the four cardinal directions. The second kind is indexing edges, which are added between each column name node (resp. row index node) and all the cell nodes that belong to that column (resp. row). Finally, the third kind of edge is a representation edge, between the representor node to all the other nodes corresponding to the contents of the dataframe.

Encoding the Query Q. Finally, to encode $Q = (\mathcal{D}, C)$, we construct G_e for each element in $\mathcal{D}$ and C as described above, and create a graph G containing these G_es as sub-graphs. Additionally, to capture relationships amongst these elements, we add a fourth kind of edge - *the equality edge*, between nodes originating in different G_es such that the elements they represent are equal. Formally, we add an equality edge between nodes n_1 and n_2 if $n_1 \in G_{e_i} \wedge n_2 \in G_{e_j} \wedge i \neq j \wedge V(n_1) = V(n_2)$ where V is a function that given n, retrieves the value encoded as n. For representor nodes, V returns the whole element it represents. For example, for a dataframe, V would return the dataframe itself for the representor node.

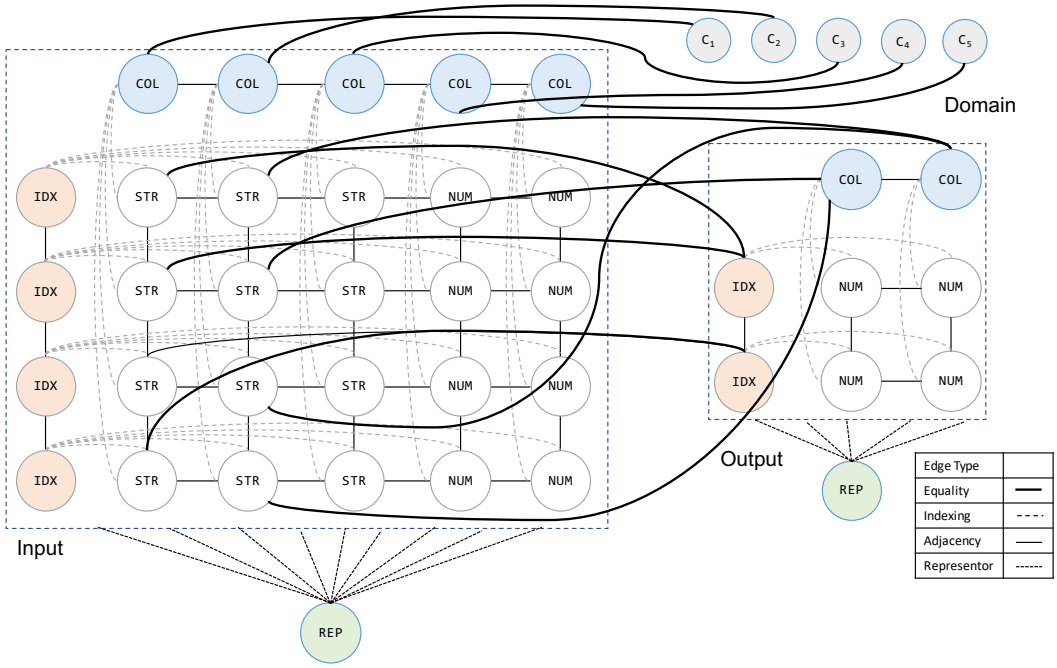


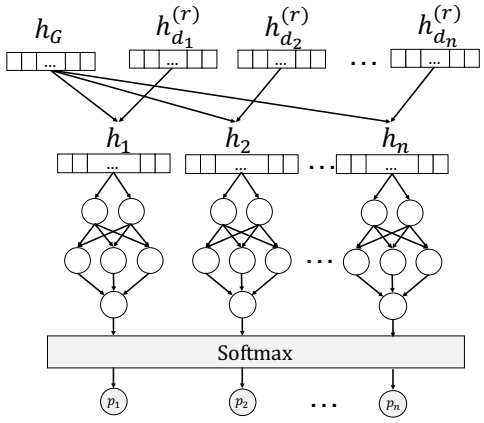
Fig. 7. Graph encoding of the query passed to the `Select` call at Line 4 in Figure 3, on the I/O example from Figure 1.

Equality edges are key to capturing relationships between the inputs and the output in the I/O example, as well as the domain $\mathcal{D}$ and the I/O example. The neural network model can then learn to extract these relationships and use them to infer the required probability distribution.

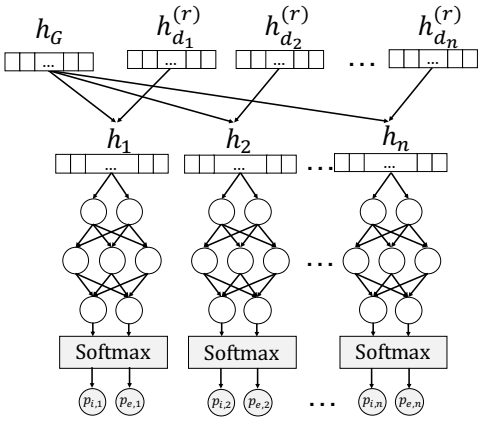
3.3.3 Operator-Specific Graph Neural Network Models. Given the graph-based encoding G_Q of a query Q, we feed it to a graph neural network model. Each operator has a different model. These models are based on the gated graph neural network, introduced by Li et al. [2015]. We base our model on the implementation by Microsoft [Allamanis et al. 2018; Microsoft 2017]. We first describe the common component of all the neural network models. Then, we provide an individual description for the neural network model corresponding to each operator listed in Table 1.

The input to all our network models is a undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X})$. $\mathcal{V}$ and $\mathcal{X}$ characterize the nodes, where $\mathcal{V}$ is the set of nodes and $\mathcal{X}$ is the embedding $\mathcal{X} : \mathcal{V} \to \mathbb{R}^D$. Effectively, $\mathcal{X}$ maps each node to a one-hot encoding of its label of size D, where D is a hyper-parameter. $\mathcal{E}$ contains the edges, where each edge $e \in \mathcal{E}$ is a 3-tuple (v_s, v_t, t_e). The source and target nodes are v_s and v_t, respectively. The type t_e of the edge is one of $\Gamma_e \equiv \{adjacency, indexing, representor, equality\}$ and is also one-hot encoded.

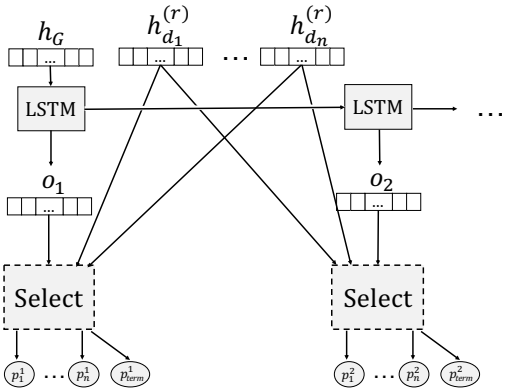
Each node v is assigned a state vector $h_v \in \mathbb{R}^D$. We initialize the vector to the node embedding $h_v^{(0)} = \mathcal{X}(v)$. The network then propagates information via r rounds of *message passing*. During round k ($0 \leq k < r$), messages are sent across edges. In particular, for each edge (v_s, v_t, t_e), v_s sends the message $m_{v_s \to v_t} = f_k(h_{v_s}^{(k)}, t_e)$ to v_t. Our $f_k : \mathbb{R}^{D+|\Gamma_e|} \to \mathbb{R}^D$ is a single linear layer. These are parameterized by a weight matrix and a bias vector, which are learnable parameters. Each node v aggregates its incoming messages into $m_v = g(\{m_{v_s \to v} \mid (v_s, v, t_e) \in \mathcal{E}\})$ using the aggregator g. In our case, we take g to be the element-wise mean of the incoming messages. The new node



(a) Illustration of the **Select** model.



(b) Illustration of the **Subset** model.



(c) Illustration of the **OrderedSubset**/**Sequence** model.
The box label "Select" expands to (a).

Fig. 8. Operator-specific neural network architectures.

state vector $h_v^{(k+1)}$ for the next round is then computed as $h_v^{(k+1)} = GRU(m_v, h_v^{(k)})$ where GRU is the gated recurrent unit [Cho et al. 2014] with start state as $h_v^{(k)}$ and input m_v. We use $r = 3$ rounds of message passing, as we noticed experimentally that further increasing the number of message passing rounds did not increase validation accuracy.

After message passing is completed, we are left with updated state vectors $h_v^{(r)}$ for each node v. Now depending on the operator, these node vectors are further processed in different ways as described below to obtain the corresponding probability distributions over space of values defined by the operator (see Figure 4a). A graphical visualization is also provided in Figure 8

Select : We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G. We now concatenate h_G with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_i for each element in the domain $\mathcal{D}$ in the query Q, to obtain vectors $h_i = h_G \circ h_{d_i}^{(r)}$. We pass the h_is though a multi-layer perceptron with one hidden layer and a one-dimensional output layer, and apply softmax over the output values for all the elements to produce a probability distribution over

the domain elements $(p_1, \cdots, p_n)$. For inference, this distribution is returned as the result, while during training we compute cross-entropy loss w.r.t this distribution and the correct distribution where $p_i = 1$ for the correct choice i and $\forall j \neq i, p_j = 0$. Figure 8a shows an illustration of the model.

Subset : As in **Select**, we perform element-wise sum-pooling of the node state vectors and concatenate it with the state vectors of representor nodes to obtain the vectors $h_i = h_G \circ h_{d_i}^{(r)}$ for each element in the domain. However, we now pass the h_is though a multi-layer perceptron with one hidden layer and apply softmax activation on the output layer to obtain a distribution (p_{i_k}, p_{e_k}) over two label classes "include" and "exclude" for each of the domain element d_k individually. Recall that the space of possible outputs for the **Subset** operator is the power-set of the domain $\mathcal{D}$. The probability of these labels corresponds to the probability with which an element is included and excluded from the output set respectively. To compute the probability distribution, the probability of each possible output set is computed as simply the product of the "include" probabilities for the elements included in the set and the "exclude" probabilities for the elements excluded from the set. Again, this distribution is returned as the result during inference, while during training, loss is computed w.r.t this distribution and the correct individual distribution of the elements where $p_{i_k} = 1 \wedge p_{e_k} = 0$ if element d_k is present in the correct output, else $p_{i_k} = 0 \wedge p_{e_k} = 1$. Figure 8b shows an illustration of the model.

OrderedSubset and Sequence : We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G. We then pass h_G to an LSTM that is unrolled for $T + 1$ time-steps, where $T = |\mathcal{D}|$ for **OrderedSubset** and $T = l$ for **Sequence**(l) where l the max-length parameter passed to **Sequence**. The extra time-step is to accommodate a terminal token which we describe later. For each time-step t, the output o_t is concatenated with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_is for each element in the domain passed to the operator to obtain vectors $h_i^t = o_t \circ h_{d_i}^{(r)}$. At time-step t, in a similar fashion as **Select**, a probability distribution is then computed over the domain elements plus an arbitrary terminal token *term*. The terminal token is used to indicate the end of a sequence/set. Now, to compute the probability distribution, the probability of each set or sequence $(a_0, \cdots, a_k)$ where $(k \leq T)$ is simply the product of probabilities of a_i at time-step i and the probability of the terminal token *term* at time-step $k + 1$. As before, this distribution is directly returned during inference, while during training, loss is aggregated over individual time-steps; the loss for a time-step is computed as described in **Select**. Figure 8c shows an illustration of the model.

All the network models are trained with the ADAM optimizer [Kingma and Ba 2014] using cross-entropy loss.

3.4 Training Neural-Backed Generators for Pandas

A Neural-Backed Generator consists of operators backed by *Rank* functions that influence their behavior. We implement these *Rank* functions using neural networks. as described in Section 3.3.3. Training each of these networks for each call to an operator with static ID *id* requires training data consisting of tuples of the form $\mathcal{T}_{id} = (C, \mathcal{D}, c)$ where c is the correct choice to be made by the operator call with static id *id*. Put another way, the neural network behind the operator call at location *id* is trained to predict the choice c with the highest probability given the context C and domain $\mathcal{D}$.

Unfortunately, such training data is not available externally as it is highly specific to our generators. We therefore aim to synthesize our training data automatically, i.e., synthesize a random tuple containing a context C, domain $\mathcal{D}$, and the target choice c. This is a highly non-trivial problem, as there are two strong constraints that need to be imposed on C, $\mathcal{D}$ and c for this tuple to be a useful training data-point. First, the random context, domain and choice should be *valid*. That is, there should exist *an execution of the generator* for some input such that the operator call in question receives the random context and domain as input, and makes the same choice. Second, this tuple of context, domain and choice should be *meaningful*, i.e., the choice should lead to progress on the task contained in the context. In our synthesis setting, this translates to the property that the generator makes a step towards producing a program that actually produces the output from the input as passed in the context. We rely on two key insights to solve these problems for our pandas program candidate generator.

Suppose we have tuples of the form $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ where $\mathcal{P}$ is a pandas program such that $\mathcal{P}(\mathcal{I}) = \mathcal{O}$ i.e. it produces $\mathcal{O}$ when executed on inputs $\mathcal{I}$. Also, K is the sequence of choices made by the operators in the generator such that the generator produces the program $\mathcal{P}$ when it is fed $\mathcal{I}$ and $\mathcal{O}$ as inputs. Then, it is straight-forward to extract training data tuples $(C, \mathcal{D}, c)$ for each operator call by simply running the generator on $\mathcal{I}$ and $\mathcal{O}$ and recording the concrete context C and domain $\mathcal{D}$ passed to the operator, and forcing the operator to make the choice c. These tuples are also *meaningful* by construction, as the operators make choices that lead to the generation of the program $\mathcal{P}$ which solves the synthesis task described by $\mathcal{I}$ and $\mathcal{O}$.

The second insight is that we can obtain these $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ tuples by using the generator itself. We generate random inputs $\mathcal{I}$ (DataFrames), run the generator on $\mathcal{I}$ using the randomized semantics presented in Figure 4c while simultaneously recording the choices made as K. The program $\mathcal{P}$ returned by the generator is then run on $\mathcal{I}$ to yield $\mathcal{O}$.

The sheer size of APIs such as `pandas` presents another problem in this data generation process. The large number of functions yields a huge number of possible sequences of these functions (Lines 3-4 in Figure 6). Even when considering sequences of length ≤ 3, the total number of sequences possible from the 119 `pandas` functions we support is ~500,000. Generating enough examples for all function sequences to cover a satisfactory portion of all the possible argument combinations is prohibitively expensive and would result in dataset of enormous size that cannot be processed and learned from in reasonable time.

However, not all sequences actually occur in practice. Practitioners of the library come up with sequences that are useful in solving real-world examples. So, we mine Github and StackOverflow to collect the function sequences used in the real-world. We were able to extract ~4300 sequences from both these sources. Then, while generating the tuples $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ using randomized semantics, we tweak the semantics of just the call to `Sequence` at Line 4 in Figure 6 to randomly return sequences from only this mined set of sequences.

4 EVALUATION

We first evaluate the feasibility and effectiveness of our technique by evaluating the end-to-end ability of AutoPandas to synthesize solutions for real-world benchmarks. We then provide deeper insights into the performance of our neural network models and compare it with two baselines to demonstrate the usefulness of the models.

4.1 Implementation

We implement the overall technique described in Section 3 in a tool called AutoPandas. AutoPandas consists of 25k lines of Python code, and uses Tensorflow [Abadi et al. 2015] to implement the neural network models. The code is available at https://github.com/rbavishi/autopandas.

4.2 Training and Setup

We generated 6 million (input, output, program, generator choices) training tuples (as described in Section 3.4) containing 2 million tuples each for programs consisting of one, two, and three function calls. Similarly, we generate 300K validation tuples with 100K tuples each for the three function sequence lengths. From these tuples we extract training and validation data for the 320 operator calls in our program candidate generator for `pandas`, and train their respective models for 10 epochs on four Nvidia Titan V GPUs. We finished training all the models in 48 hours. All our synthesis experiments are run on a single 8-core machine containing Intel i7-7700K 4.20GHz CPUs running Ubuntu 16.04.

4.3 Performance on Real-World Benchmarks

We evaluated AutoPandas on 26 benchmarks taken from StackOverflow questions containing the `dataframe` tag. We ran AutoPandas with a time-out of 20 minutes and used smart depth-first enumeration semantics for the program candidate generator. We also impose an upper bound on the number of Replays an operator is allowed to make (see the Op-Replay semantics in Figure 4d). This prevents any operator from limiting the scope of exploration when it can return a very large number of values given a single domain. In our experiments, we set a bound of 1000. For comparison, we also implement a baseline version of AutoPandas called Baseline that follows depth-first exhaustive enumeration semantics (Figure 4d) for all operator calls except the **Sequence** invocation. The rationale is that given the size of the search space, it is more meaningful to compare the performance of the models backing the exploration of function arguments given the same function sequences. Table 2 contains the results.

The column *Depth* contains the length of the function sequence used in the official solution for the benchmark. *Cand. Explored* denotes the number of candidates both approaches had to check for correctness before arriving at one which produces the target output. *Seq. Explored* contains the number of function sequences explored (by the **Sequence** call at Line 4 in Figure 6), while the *Time* column contains the time taken (in seconds) to produce a solution if any.

AutoPandas can solve 17 out of the 26 benchmarks. The Baseline approach solves 14/26. Both approaches tend to miss the 20 minute mark more often on benchmarks with higher depths, which is expected as the space of possible programs grows exponentially with the length of the function sequence being explored. The guided execution of the program candidate generator enabled by neural networks allows AutoPandas to search this enormous space in reasonable time.

Even on the benchmarks that are solved by both approaches, the lower numbers in the *Candidates Explored* column indicate that our neural-backed program candidate generator indeed learns to adapt to the synthesis task at hand, generating the solution faster than the baseline. Finally the number of sequences explored in both approaches is always at most 10, and often 1, suggesting that the sequence prediction component is quite effective. The difference in time between the two approaches is relatively smaller than in candidate numbers, because AutoPandas includes the time taken to query the neural network models and interpret its results. However we believe this is fundamentally an engineering issue. Performance could easily be improved by batching queries, parallelizing exploration and speculative execution of the generator while waiting for results from the models.

Most of the benchmarks on which AutoPandas fails to find a solution involve arithmetic functions. AutoPandas's encoding does not capture arithmetic relationships readily, so its function sequence prediction is not as accurate for these sequences. In future work we plan to explore richer encoding schemes that can tackle these issues.

Benchmark	Depth	Candidates Explored		Sequences Explored		Solved		Time(s)	
		AutoPandas	Baseline	AutoPandas	Baseline	AutoPandas	Baseline	AutoPandas	Baseline
SO_11881165	1	**15**	64	1	1	Y	Y	**0.54**	1.46
SO_11941492	1	783	**441**	8	8	Y	Y	12.55	**2.38**
SO_13647222	1	**5**	15696	1	1	Y	Y	**3.32**	53.07
SO_18172851	1	-	-	-	-	N	N	-	-
SO_49583055	1	-	-	-	-	N	N	-	-
SO_49592930	1	**2**	4	1	1	Y	Y	**1.1**	1.43
SO_49572546	1	**3**	4	1	1	Y	Y	**1.1**	1.44
SO_13261175	1	**39537**	-	18	-	**Y**	N	**300.20**	-
SO_13793321	1	**92**	1456	1	1	Y	Y	**4.16**	5.76
SO_14085517	1	**10**	208	1	1	Y	Y	2.24	**2.01**
SO_11418192	2	158	**80**	1	1	Y	Y	**0.71**	1.46
SO_49567723	2	**1684022**	-	2	-	**Y**	N	**753.10**	-
SO_13261691	2	**65**	612	1	1	Y	Y	**2.96**	3.22
SO_13659881	2	**2**	15	1	1	Y	Y	**1.38**	1.41
SO_13807758	2	711	**263**	2	2	Y	Y	7.21	**1.81**
SO_34365578	2	-	-	-	-	N	N	-	-
SO_10982266	3	-	-	-	-	N	N	-	-
SO_11811392	3	-	-	-	-	N	N	-	-
SO_49581206	3	-	-	-	-	N	N	-	-
SO_12065885	3	**924**	2072	1	1	Y	Y	**0.9**	4.67
SO_13576164	3	**22966**	-	5	-	**Y**	N	**339.25**	-
SO_14023037	3	-	-	-	-	N	N	-	-
SO_53762029	3	**27**	115	1	1	Y	Y	1.90	**1.50**
SO_21982987	3	8385	**8278**	10	10	Y	Y	30.80	**13.91**
SO_39656670	3	-	-	-	-	N	N	-	-
SO_23321300	3	-	-	-	-	N	N	-	-
Total						17/26	14/26		

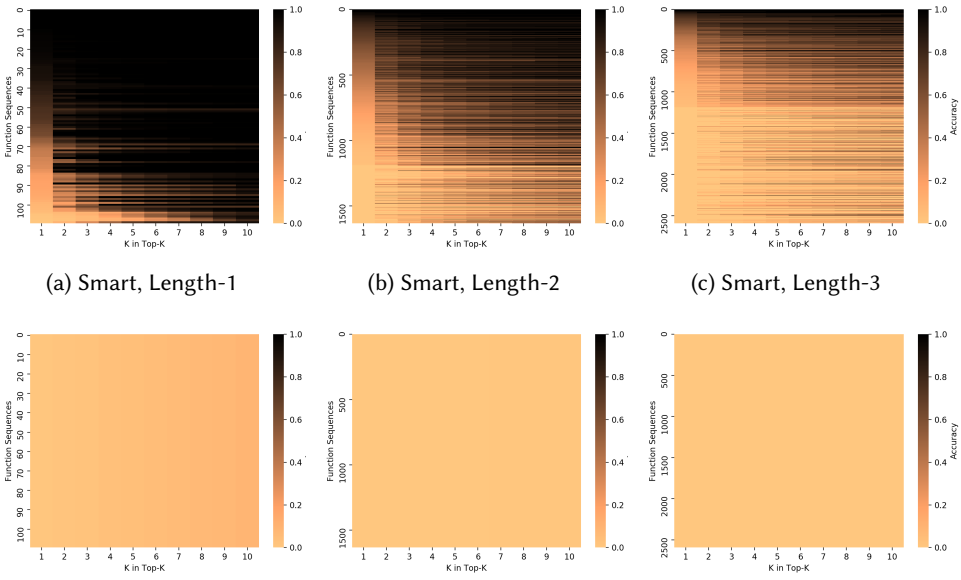
Table 2. Performance on Real-World Benchmarks. Dashes (-) indicate timeouts by the technique.

4.4 Analysis of Neural Network Models

4.4.1 Function Sequence Prediction Performance. We single out the call to **Sequence** in our program candidate generator as it is the component most critical to the performance of the generator, and dissect the performance of the neural network model backing it; on our synthetic validation dataset in Figure 9. In particular, we measure top-1 to top-10 accuracies on a per-sequence basis. Recall that these are the sequences mined from GitHub and StackOverflow. Figures 9a-9c show the performance of the model when predicting sequences of lengths 1, 2 and 3 respectively. As expected, the performance for shorter sequences is better as the logical distance between the input and output is lower, and therefore the encoding can capture sufficient information. Another reason for poorer accuracies at higher lengths is the fact that for large APIs like pandas functions often have overlapping semantics. Therefore multiple sequences may produce viable solutions for a given output example. This is reinforced by the results on real-world benchmarks in Table 2. In particular the numbers in the "Sequences Explored" column for AutoPandas suggest that the model indeed predicts useful sequences, even if they don't match the ground-truth sequence.

Figures 9d-9f present the expected accuracies of a purely random model on the same dataset. As expected, the accuracies are almost zero (there is a slight gradient in Figure 9d). The sheer number of possible sequences makes it improbable for a random model to succeed on this task; even our baseline benefited from the neural model's predictions.

4.4.2 Comparison with Deterministic and Randomized Semantics. We demonstrate the efficacy of the smart semantics for operators by comparing the neural network models with deterministic and randomized baselines. In the deterministic baseline, the order in which operators return values is fixed for a given input domain (see Figure 4d). In the randomized baseline, the operator returns values in a random order (see Figure 4c). We expect the neural network approach (see Figure 4e) to perform better than both these baselines as it is utilizing the context. Figure 10 shows the results.



(a) Smart, Length-1 (b) Smart, Length-2 (c) Smart, Length-3

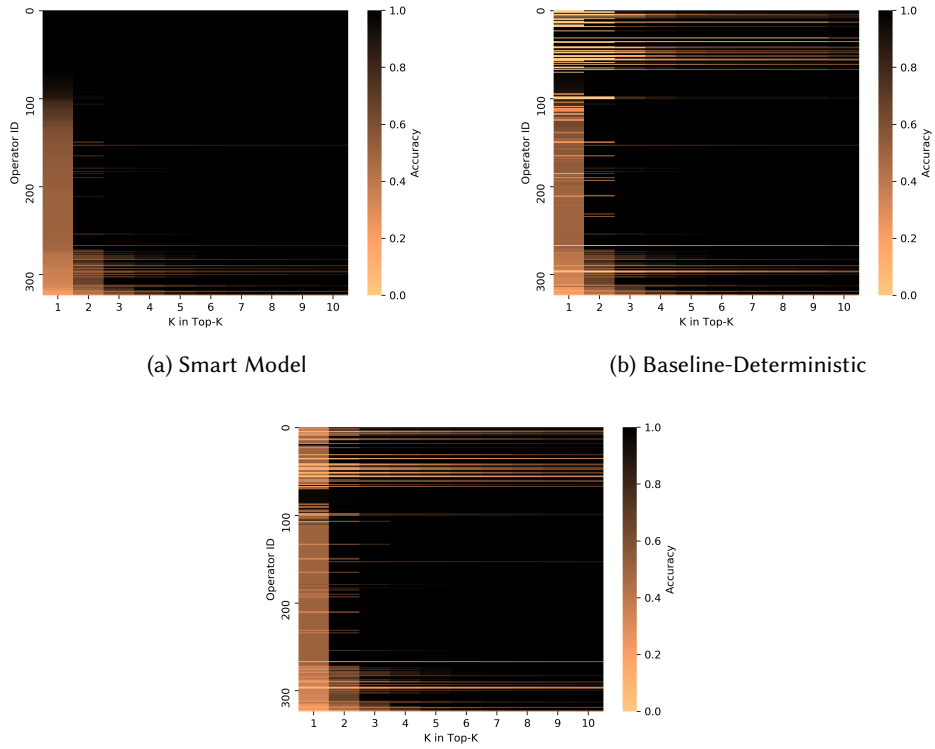
(d) Random, Length-1 (e) Random, Length-2 (f) Random, Length-3

Fig. 9. Smart Model Accuracies on Function Prediction Task, compared to a Random Baseline. Per-sequence Top-k accuracies provided. Color gives accuracy; darker is better. The color point (x, y) gives the top-x accuracy for sequence with ID y. Sequence IDs are sorted based on top-1 accuracy of the smart model.

We see that while a randomized approach smoothens results compared to the deterministic approach (ref. Figure 10c vs. Figure 10b), both still have significant difficulty on certain operator calls (top-left corners of all graphs). The neural network model performs quite well in comparison. There are operator calls where all the three approaches perform poorly or all perform well. The former can be attributed to insufficient information in the context. For example, if a `pandas` function supports two modes of operation which can both lead to a solution, the model may be penalized in terms of accuracies, but may not affect its performance in the actual task. The latter case, where all approaches perform well, can be mostly attributed to small domains. For example, many `pandas` functions take an `axis` argument that can only take the value 0 or 1, which can be modeled as `Select({0,1})` in the generator. Hence the top-2 accuracy of all the approaches will be 100%.

Overall, we see that the neural-backed operators arrive at the correct 'guess' much more quickly than their randomized or deterministic counter-parts, thus helping the generator as a whole to arrive at the solution more efficiently. In fact, the accuracies in Figure 10 are quite high for the neural-backed operators overall. We think this is a very encouraging result, as we are able to learn useful operator-level heuristics.

The contrast between the overall high accuracies in Figure 10a and the accuracies in Figure 9 suggests that the biggest bottleneck is predicting the correct function sequence. This and the previous observation are reinforced by the columns containing the number of candidates and function sequences explored in Table 2.



(a) Smart Model

(b) Baseline-Deterministic

(c) Baseline-Randomized

Fig. 10. Per-operator Top-k accuracies. Color gives accuracy; darker is better. The color point (x, y) gives the top-x accuracy for operator with ID y. Operator IDs are sorted based on top-1 accuracy of the smart model.

5 LIMITATIONS

Our program candidate generator has been hand-written by consulting the Pandas source code and therefore may not be completely faithful to the full internal usage specification of all functions. Due to the complexity of the API pre-conditions for pandas, writing this generator required substantial manual labor. While the writer of such a generator needs only knowledge of the API, and not of the synthesis engine, this still poses a practical barrier to implementing the technique for other APIs. Our technique requires that all programs be expressible in terms of the generator, so the generator restricts the space of programs we can synthesize. The current generator behind AutoPandas only expresses programs consisting of a sequence of API calls of maximum length three.

Another limitation is that our synthetic data may not be representative of the usage of pandas in the real-world. For example, our dataset may contain many transformations that do not "look useful" to a human, and thus our models may be biased away from useful ones. This may be a factor in why we are unable to synthesize 9 of our real-world benchmarks within timeout. Further, our training dataset may not contain enough data-points for effective training for each operator call in our candidate generator. We plan to address this by generating datasets than are larger and have more realistic program distributions for training, by improved sampling of generator executions.

In this paper, we only experimented with graph encodings of DataFrames for our smart graph-neural-network based backends, as described in Section 3.3.2. Graphs could potentially be used to encode many other data structures, including: matrices (similar to dataframes), strings (each node is a character), lists, trees, maps, and objects (each object is a node and each field denotes an outgoing edge). The backends for our smart operators can *ingest* such encodings. However, it is possible the accuracy results for smart operators would not be as high as those presented in Section 4.4 for graphs generated in a different domain.

Also with regards to the encoding, the impact of which relationship edges to add may have a large impact on the results. In this work, we focused on equality edges only. These edges will help distinguish shape transformation methods, but will not distinguish e.g. arithmetic operations. In fact, we observed that our current encoding cannot distinguish *amongst* arithmetic operations such as `min`, `max`, `mean`, etc. However, top-k accuracy remains reasonable because the encoding appears to cluster arithmetic operations together—all these different arithmetic operations result in similar graph encodings. In other domains where shape transformations may be less significant than data transformation, more feature engineering work may need to be done to determine which edges should be added in a graph encoding.

In this paper we omit a full evaluation of whether a simpler machine learning model operating on hand-engineered features over the I/O example could perform as well as our graph-neural-network backends. However, we experimented with manual feature engineering for generator-based synthesis where we trained an SVM over features such as, "a row name becomes a column name", "a data value becomes a column name", etc. We found that results for the overall function sequence prediction task were poor, and reasonable results were obtained only for the functions we used as exemplars in the feature engineering process. Although it is possible the results could be improved with more careful feature engineering, we believe this may be more tedious and harder to scale than specifying relationship edges in our proposed graph encoding.

6 DISCUSSION

Although the results suggest that our current AutoPandas system works pretty well, the neural-backed program candidate generator we use is only one of the many possible generators in a large design space. At a high-level, our generator works by first predicting an entire sequence of functions, and then exploring the resulting space of argument combinations. However, predicting entire sequences is prone to error, especially at higher lengths, since the logical distance between the input and the target output may not allow our graph-based encoding to capture complete information about the transformation. Another possible approach is to predict only one function (along with its arguments) at a time, and make the next decision based on the output of running this function. This approach is closer to reinforcement learning as feedback is solicited at the end of every decision, but with the additional option of backtracking. We intend to explore this direction in future work.

One of the key elements which allows neural-network backed execution of generators is our graph-based encoding of the domain and context that are passed as queries to the network, where relationships between elements are captured using edges. These edges can be thought of as dependency relationships. When considering DataFrames as in our case, these edges (especially equality edges) capture the elements of the output that are dependent on certain elements of the input. This presents an opportunity for user interaction — the user, along with the input-output example, can provide additional help by pointing out the relationships between the cells of the input and output dataframe, which can be directly captured as edges in our encoding. This has the potential of greatly speeding up synthesis and we plan to investigate it further.

Finally, we strongly believe that generators have applications in various other fields apart from program synthesis such as program testing and automated documentation. For example, the generator in Figure 3 for the `pivot` function additionally serves as a precise specification of its intended usage, and can serve as a medium of API usage communication between developers and users. It can also serve as an API contract that implementers of `pivot` can adhere to. It also presents a potential testing application as exhaustively enumerating this generator effectively acts as a stress test for the pivot function and may be extremely useful in regression tests. Finally, for automated testing techniques such as fuzzing, our neural backed generators may be useful as program feedback can be used to train model son-the-fly to bias the generator towards generating inputs that are more likely to exercise interesting parts of the program under test. This is similar to work done in neural fuzzing in Böttinger et al. [2018].

7 RELATED WORK

7.1 Program Synthesis

A large body of work has been dedicated to solving program synthesis problems. Numerous systems have been developed targeting a variety of domains such as string processing [Gulwani 2011; Parisotto et al. 2017], data wrangling [Feng et al. 2018, 2017; Le and Gulwani 2014], data processing [Smith and Albarghouthi 2016; Yaghmazadeh et al. 2018], syntax transformations [Rolim et al. 2017], database queries [Yaghmazadeh et al. 2017] and bit-vector manipulations [Jha et al. 2010]. We attempt to categorise these works at a coarse level according to the high-level synthesis strategy used in their respective systems. We then summarise how our strategy of using neural-backed generators compares and relates to these strategies.

CEGIS [Solar-Lezama 2008; Solar-Lezama et al. 2006] is a general framework for program synthesis that synthesizes programs satisfying a specification Φ. The basic algorithm involves two components — a synthesizer and verifier, where the synthesizer generates candidate programs and the verifier confirms whether the candidate is correct. The synthesizer also takes the space of possible candidates as an input, either explicitly or implicitly. This space may be defined in multiple ways, for example using syntactic definitions of valid programs [Alur et al. 2013]. A large fraction of techniques, including ours, fall under this general CEGIS framework, with differing strategies for producing program candidates, which are broadly classified below.

7.1.1 Synthesis using Logical Reasoning. At a high-level, approaches using logical reasoning either encode the synthesis problem as a constraint-solving problem and use SAT/SMT solvers to generate valid solutions to the constraints [Jha et al. 2010], or use logical specifications to prune the search space. These specifications can be manually specified [Feng et al. 2017; Polikarpova et al. 2016] or learnt as lemmas during synthesis [Feng et al. 2018; Wang et al. 2017]. Regardless, these approaches target domains that are amenable to logical specification, such as string processing and numerical domains. Although Feng et al. [2018, 2017] target the same space as our work—DataFrame transformations—, they consider only 10 R functions and support a fraction of the possible argument combinations. These functions are accompanied by 10 specifications over the structural constraints imposed by these functions. In contrast, our generators are general-purpose and constraints can be written in the target language itself. The choice points introduced by various operators at different locations in the generator are backed by neural network models that effectively guide the search. This allows us to target domains such as `pandas` which is not amenable to logical specification and constraints.

7.1.2 Domain-Specific Inductive Synthesis. These class of approaches involve the design of a special-purpose DSL tailored towards the target domain [Gulwani 2011; Polozov and Gulwani 2015] such

as string processing or table data extraction. Each DSL operator is backed by an algorithm called a *witness function* that prunes away impossible candidates for the given I/O example. Such approaches are highly efficient and can solve a large number of tasks provided their solutions can be expressed in the DSL.

However, targeting a new domain entails the tedious process of designing a DSL for that domain, along with custom algorithms that are fundamental to the synthesis algorithm itself. In contrast, our generators allow us to encode these algorithms using operators backed by neural networks. Our neural network models are akin to the witness functions used in these approaches.

7.1.3 Synthesis using Machine Learning. One class of machine learning approaches predict programs directly using neural networks [Devlin et al. 2017; Parisotto et al. 2017] while another employs neural networks to guide the symbolic techniques above [Balog et al. 2016; Bunel et al. 2018; Kalyan et al. 2018]. In both cases, the models involved take the input-output example directly and make predictions accordingly. However the domains tackled so far are simpler; Balog et al. [2016]; Devlin et al. [2017]; Kalyan et al. [2018] target string-processing and lists of bounded integers where machine learning models such as cross-correlational networks, LSTMs with attention are applicable. In contrast, these models cannot target DataFrames due to the issues we detail in Section 3.3.2. Additionally, since DataFrames are not of a fixed size, the CNNs used to encode fixed-size grid-based examples in Bunel et al. [2018] are also not applicable.

Another class of approaches use probabilistic models to rank program candidates generated by the synthesizer [Feng et al. 2018; Lee et al. 2018; Raychev et al. 2014]. These models are trained on data extracted from large open repositories of code such as Github and StackOverflow. We follow suit in using existing pandas programs to generate training data for our generators. With respect to the target domain, the closest related work is the *TDE* system introduced by [He et al. 2018] which also targets generic APIs in Java. It mines a large corpus of API methods along with the raw arguments taken by those methods. Then, given a new I/O example, it searches through this corpus using a mix of statistical and static analysis techniques. However, the mining of this corpus relies on the availability of rich type information about the API as well as the I/O example, something which is not available for popular Python libraries such as Tensorflow and Pandas. Moreover, *TDE* cannot generate unseen argument combinations, unlike our generator-based approach.

7.2 Graph Neural Networks

Dai et al. [2017]; Kool et al. [2019] use graph-neural networks to solve difficult combinatorial optimization problems such as the Travelling Salesman Problem (TSP). Although their use-case is different, the adopted approach is similar in spirit to the use of graph neural networks in AutoPandas. Their network iteratively constructs the solution by *selecting* and adding a node to the solution in every iteration. This updated graph is then used as input in the next iteration. Similarly in our neural-backed generators, every operator makes a decision based on a graph-encoding of the input query, which may influence the graph-encoding for a query to a subsequent operator call in the generator.

7.3 Generator-based Random Testing

Our generator-based approach to synthesis bears, at a high level, many similarities to generator-based testing. Generator-based testing, pioneered by QuickCheck [Claessen and Hughes 2000], allows users to test their programs by (1) writing a parameterized property test, usually containing an assertion, and (2) repeatedly running this property test with inputs produced with by a type-specific input generator. These input generators usually have randomized semantics to generate a variety of inputs of the given type, such that running the property test with many inputs produced

by the generator quickly "checks" it. Overall, QuickCheck generates inputs to try to find an assertion violation, while our approach generates programs to try and find ones that satisfy the input-output example. Thus, using the randomized semantics for our operators is analogous to a QuickCheck-style testing approach.

However, using purely randomized semantics for generators is unlikely to find a program satisfying the input-output example in reasonable time (see 9, Figures 10). In some cases the same issue arises in testing. There have been recent works in software testing focusing on "guiding" QuickCheck generators towards a testing objective [Löscher and Sagonas 2017; Padhye et al. 2019]. These rely on evolutionary approaches that improve towards the testing objective in a large number of trials trials. Our goal in synthesis is to get the objective (satisfying the I/O example) âĂIJrightâĂİ on the first try. The key advantage of the programming-by-example setting over the testing one is the presence of the I/O example, over which the randomness in the generator can be conditioned. This conditioning allows the neural-backed operators return the correct domain element in the first few tries, instead of relying on evolutionary approaches.

7.4 Probabilistic Programming Languages

Readers familiar with Probabilistic Programming Languages (PPLs) [Gordon et al. 2014] may notice the similarity of our generators to models in PPLs. In particular, the interleaving of operators and arbitrary Python code in our neural-backed generators bears significant resemblance to the interplay between sampling from latent distributions and deterministic code in programs written in PPLs. One possible encoding of our synthesis problem using PPLs would take the input-output example as the input (y) to the model with the observed variable (x) or output as the Pandas program, and the latent random variable (z) as the the choices to be made by the operators. Due to the purely discrete nature of our generators, the model ($p(x|z)p(z)$) is *fixed* in the sense that the choices made by the operators uniquely determine the generated pandas program. This is in contrast to the usual probability distributions that are computed by models written in a PPL. Hence only the *guide* ($p(z|y)$) needs to be trained for the problem. Therefore, our formulation is a specific instance of the general PPL inference problem where the full generality of PPL inference algorithms is not required and is possibly inefficient.

8 CONCLUSION

In this paper we introduced a neural-backed, generator-based approach to synthesize programs matching an input-output example *directly* in a large API. We hand-wrote a candidate program generator for the subset of the Python `pandas` API dealing with dataframe transformations. The candidate program generator includes argument generators for each of these 119 functions, each of which captures the key argument constraints for the given function. We introduce smart operators to make the arbitrary decisions in these generators. We find that these smart operators are much more accurate than deterministic or random operators on validation data. In addition, their use improves the *efficiency* of the program candidate generator, allowing 3 extra test benchmarks to be synthesized before timeout compared to our baseline. Although there remain possible engineering improvements to improve speedups, the results suggest our technique is a promising one for this broad-language synthesis problem.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1409872 and Grant No. 1817122. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

2014. The pandas project. https://pandas.pydata.org. Accessed October 11th, 2018.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. *CoRR* abs/1801.04589 (2018). arXiv:1801.04589 http://arxiv.org/abs/1801.04589

Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. *CoRR* abs/1805.04276 (2018). arXiv:1805.04276 http://arxiv.org/abs/1805.04276

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 1724–1734. https://doi.org/10.3115/v1/D14-1179

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning Combinatorial Optimization Algorithms over Graphs. *CoRR* abs/1704.01665 (2017). arXiv:1704.01665 http://arxiv.org/abs/1704.01665

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *ICML 2017*. https://www.microsoft.com/en-us/research/publication/robustfill-neural-program-learning-noisy-io/

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.* 52, 6 (June 2017), 422–436. https://doi.org/10.1145/3140587.3062351

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. ACM, New York, NY, USA, 167–181. https://doi.org/10.1145/2593882.2593900

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (June 2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *ArXiv e-prints* (April 2018). arXiv:cs.AI/1804.01186

D. P. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. *ArXiv e-prints* (Dec. 2014). arXiv:1412.6980

Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, Learn to Solve Routing Problems!. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ByxBFsRqYm

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA,

542–553. https://doi.org/10.1145/2594291.2594333

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2015. Gated Graph Sequence Neural Networks. *CoRR* abs/1511.05493 (2015). arXiv:1511.05493 http://arxiv.org/abs/1511.05493

Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

Microsoft. 2017. Gated Graph Neural Network Samples. https://github.com/Microsoft/gated-graph-neural-network-samples. Accessed October 17th, 2018.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. https://doi.org/10.1145/3293882.3330576

Emilio Parisotto, Abdelrahman Mohamed, Rishabh Singh, Lihong Li, Denny Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *ICLR 2017*. https://www.microsoft.com/en-us/research/publication/neuro-symbolic-program-synthesis-2/

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (June 2016), 522–538. https://doi.org/10.1145/2980983.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 326–340. https://doi.org/10.1145/2908080.2908102

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3187009.3177735

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887