Semantic Programming by Example with Pre-trained Models

GUST VERBRUGGEN, KU Leuven, Belgium
VU LE, Microsoft, USA
SUMIT GULWANI, Microsoft, USA

The ability to learn programs from few examples is a powerful technology with disruptive applications in many domains, as it allows users to automate repetitive tasks in an intuitive way. Existing frameworks on inductive synthesis only perform syntactic manipulations, where they rely on the syntactic structure of the given examples and not their meaning. Any semantic manipulations, such as transforming dates, have to be manually encoded by the designer of the inductive programming framework. Recent advances in large language models have shown these models to be very adept at performing semantic transformations of its input by simply providing a few examples of the task at hand. When it comes to syntactic transformations, however, these models are limited in their expressive power. In this paper, we propose a novel framework for integrating inductive synthesis with few-shot learning language models to combine the strength of these two popular technologies. In particular, the inductive synthesis is tasked with breaking down the problem in smaller subproblems, among which those that cannot be solved syntactically are passed to the language model. We formalize three semantic operators that can be integrated with inductive synthesizers. To minimize invoking expensive semantic operators during learning, we introduce a novel deferred query execution algorithm that considers the operators to be oracles during learning. We evaluate our approach in the domain of string transformations: the combination methodology can automate tasks that cannot be handled using either technologies by themselves. Finally, we demonstrate the generality of our approach via a case study in the domain of string profiling.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: program synthesis, programming by example, language models

ACM Reference Format:
Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-trained Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 100 (October 2021), 25 pages. https://doi.org/10.1145/3485477

1 INTRODUCTION

Teaching a machine to write programs that satisfy a given specification is widely regarded as one of the fundamental problems in artificial intelligence. More specifically, the task of *inductive synthesis* or *programming by example*, where the specification is given by (partial) examples of the desired output on given input, allows for the automation of repetitive tasks in a variety of domains. Examples of domains in which robust synthesizers have been rapidly adapted in industrial tools are IntelliCode suggestions for code refactoring in Visual Studio [Gao et al. 2020; Miltner et al. 2019; Rolim et al. 2017], extracting tabular data in PowerQuery [Le and Gulwani 2014] and most famously the FlashFill algorithm for performing string transformations in Excel [Gulwani 2011].

Authors' addresses: Gust Verbruggen, KU Leuven, Department of Computer Science, Leuven, Belgium, gust.verbruggen@kuleuven.be; Vu Le, Microsoft, One Microsoft Way, Redmond, WA, 98052, USA, levu@microsoft.com; Sumit Gulwani, Microsoft, One Microsoft Way, Redmond, WA, 98052, USA, sumitg@microsoft.com.

2475-1421/2021/10-ART100
https://doi.org/10.1145/3485477

dec	december
nov	november
oct	*octember*
sep	*sepember*

(a) Expanding month abbreviations. After two examples, the *emphasized* output is returned.

```
Q: dec
A: december
Q: nov
A: november
Q: oct
A:
```

(b) Example of a question-answer style prompt. GPT-3 returns "october" as most likely answer.

05 jan 2001	05/01/2001
15 mar 2020	15/03/2020
16 apr 1993	
25 dec 1992	

(c) A transformation problem that requires a combination of syntactic and semantic operators.

Fig. 1. Transformations on dates require a combination of syntactic processing and semantic knowledge. Large, autoregressive language models can be taught to extract such knowledge in just a few examples.

Semanti

Current approaches in inductive synthesis are limited to writing programs that perform only syntactic transformations of the input. All information required to perform such a syntactic transformation is either available from the specification or has to be explicitly encoded in the domain specific language used by the synthesizer. A popular scenario that is often used to emphasize this limitation in the context of FlashFill is shown in Figure 1a. Without explicitly encoding information about months, the synthesizer makes a valiant attempt using only syntactic information—concatenating the constant "ember" to the input—but fails miserably. Explicitly encoding such information works for limited domains, such as dates, but quickly becomes infeasible as the number of domains grows, or when support for multiple languages or natural language processing is required for more complicated tasks.

Recent advances in transformer architectures for large, autoregressive language models have shown that these models can perform few-shot learning without fine-tuning [Brown et al. 2020; Radford et al. 2018]. Given a short prompt of text, the autoregressive model returns a distribution of likely continuations of this snippet of text. By structuring the prompts in a specific format, for example, the question answering format in Figure 1b, the model adapts to the given task at inference time and effectively solves the given problem with just a few given examples.

A first key observation is that these models are trained on vast amounts of data and have been shown to contain a lot of information about the world [Petroni et al. 2019] and that querying these models for this information through prompts neatly integrates with the kind of specifications that are used in program synthesis. The prompt describes a specification on the output of the model by providing a few input and output examples in a designated format, just like an inductive specification does the same for inductive synthesis.

A second observation is that language models use subword tokens to keep their vocabulary small [Sennrich et al. 2016] and the output is generated token by token. This allows simple, syntactic string transformation problems to be solved, but more complicated problems either require many examples or are not solved at all. Substring extraction based on regular expressions is hard, while operating on data structures other than strings is even harder. A simple task like extracting a constant number of characters from each word in a list of words is impossible if there is no combination of tokens that corresponds exactly to this substring.

Based on these two observations, we propose a novel integration of pre-trained, autoregressive language models with inductive synthesis. These few-shot learners are used to introduce semantic operators to the underlying domain-specific language, which the synthesizer can then use to solve a new class of mixed syntactic and semantic problems, such as the one in Figure 1c.

More concretely, we introduce three new *learnable* semantic operators that map a string to another string (for semantic lookup), to an integer (for indexing into the input string), or a to boolean (for conditional logics). They are learnable in the sense that their concrete executable semantics depend on each problem instance and are determined during the inductive synthesis process. However, during learning, the inductive synthesizer makes many calls to these operators. Because making queries to the language model is slow, learning becomes unfeasible in practice. To that end, we introduce a deferred querying algorithm, which assumes these operators to be oracles during learning and uses the ranking step of inductive synthesis to pick the correct program.

We have implemented this integration in the task domain of string transformations using the PROSE framework [Microsoft 2015] as FlashGPT3 and evaluate it on a collection of challenging transformation problems. Using the deferred querying algorithm, FlashGPT3 learns most programs in under 1s, with the most difficult problems taking less than 3s. On its own, GPT-3 solves fewer problems and typically requires significantly more examples on the problems that it can solve. Additionally, we demonstrate that semantic operators can be given descriptive names (for better program readability) and how these named semantic operators can be integrated in the task domain of string profiling.

1.1 Contributions

In summary, we make the following contributions.

- We propose a novel framework of integrating pre-trained language models with inductive synthesis by augmenting the language over which programs are synthesized with semantic operators that are powered by the language model.
- We present a deferred execution algorithm for quickly learning programs with these semantic operators under the uncertainty of the language models.
- We implement and evaluate this integration in the domain of string transformation problems.
- We present a case study on integrating semantics with string profiling.

2 MOTIVATING EXAMPLES

We start by illustrating some repetitive task settings that have been disrupted by inductive synthesis and that would further benefit from semantic components. The core idea behind these inductive synthesis systems is to define an appropriate domain-specific language (DSL) that can succinctly represent various tasks in an underlying domain, and to describe an appropriate learning algorithm over the DSL [Gulwani et al. 2012]. In this section, we motivate the significance of extending such DSLs with semantic components. In Sections 4 (language) and 5 (learning), we show how learning can be performed over these semantic components.

2.1 String transformations

Transforming strings by example is one of the most commonly used benchmarks in inductive synthesis. A major breakthrough in this domain was the FlashFill algorithm [Gulwani 2011]. Its ability to quickly and robustly learn string transformation programs from few examples has helped shipping it in Microsoft Excel. FlashFill is widely recognized as one of the first commercial applications of inductive program synthesis. FlashFill turned out to be a very popular feature in Excel, not least because 99% of spreadsheet users do not know programming and struggle with repetitive tasks.

Consider the task of formatting a phone number as shown in Table 2a. From the first four rows, FlashFill is able to learn a program

$$\texttt{"("} \circ \mathsf{SubStr2}(v_1, \texttt{\textbackslash d+}, 1) \circ \texttt{")␣"} \circ \mathsf{SubStr2}(v_1, \texttt{\textbackslash d+}, 1) \circ \texttt{"␣"} \circ \mathsf{SubStr2}(v_1, \texttt{\textbackslash d+}, 2)$$

that performs this transformation, where `"quoted"` strings are constants, $\circ$ denotes concatenation such that $a \circ b \equiv \mathsf{Concat}(a, b)$ and $\mathsf{SubStr2}(s, r, i)$ extracts the ith token that matches regular

Input	Output
323-708-7700	323-708-7700
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
425 235 7654	425-235-7654
425/745.8139	*425-745-8139*

(a) Formatting phone numbers [Gulwani 2011].

Input v_1	Input v_2	Output
235-7654	Taiwan	(886) 235 7654
174.5539	Spain	(34) 174 5539
(254) 9620	South Korea	(82) 254 9620
618 4390	Panama	(507) 618 4390
447/4350	Netherlands	(31) 447 4350

(b) Formatting phone numbers by looking up the country code from its name.

Fig. 2. Examples of repetitive transformation tasks.

Input v_1	Output
Jasmine was a student.	Jasmine .... a student. (past)
We are friends.	We .... friends. (present)
I was hungry.	I .... hungry. (past)

(a) Generating exercises on completing present or past simple forms.

Input v_1	Output
He wanted to eat pizza.	<u>He</u> wanted to eat pizza.
We talked for hours.	<u>We</u> talked for hours.
It is her boat.	It is <u>her</u> boat.

(b) Marking pronouns. Solutions can be generated using markup style syntax.

Fig. 3. Examples of problems and solutions from worksheets on English grammar.

expression r in string s.[1] This program is then applied to automate the intended transformation for the remaining large number of entries, thereby saving users a lot of time and frustration. A very similar problem is shown in Table 2b. This time, however, the required country codes need to be derived from the name of the country. This is a semantic transformation, which FlashFill is unable to do. Before being able to format the number, the user then first has to look up the country code. Our proposed method is able to learn an anonymous, semantic string $\longrightarrow$ string function getCountryCode(x) by exploiting the question answering (QA) capabilities of GPT-3. The full program becomes

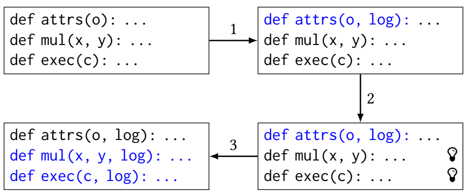
$$\texttt{"("} \circ \mathsf{getCountryCode}(v_2) \circ \texttt{") "} \circ \mathsf{SubStr2}(v_1, \backslash d+, 1) \circ \texttt{" "} \circ \mathsf{SubStr2}(v_1, \backslash d+, 2).$$

Another instance of string transformations that FlashFill struggles with is generating educational material for language learning. Textbooks and worksheets often require students to fill in gaps in sentences, to build sentences from abstract descriptions, to mark parts of sentences or to perform other manipulations of given sentences and words. Some examples of exercises are shown in Figure 3. The process of coming up with sentences, turning them into exercises and generating solution sheets is a repetitive process. Natural language sentences do not contain syntactic clues that FlashFill can use to determine positions for extracting substrings. Determining these positions requires understanding of natural language, and generating exercises in a specific format, or solutions with a specific markup, requires syntactic manipulations. Our method learns anonymous, semantic string $\longrightarrow$ int operators such as positionLeftOfPredicate(x) or positionRightOfSubject(x) that exploit the ability of GPT-3 to parse the grammatical structure of sentences and return positions in the sentence that FlashFill can use. The full program that solves the task in Figure 3b is

$$\mathsf{SubStr}(v_1, 0, \mathsf{p}_{\mathsf{Left}}) \circ \texttt{"<u>"} \circ \mathsf{SubStr}(v_1, \mathsf{p}_{\mathsf{Left}}, \mathsf{p}_{\mathsf{Right}}) \circ \texttt{"</u>"} \circ \mathsf{SubStr}(v_1, \mathsf{p}_{\mathsf{Right}}, -1)$$

where $\mathsf{p}_{\mathsf{Left}}$ = positionLeftOfPronoun(v_1), $\mathsf{p}_{\mathsf{Right}}$ = positionRightOfPronoun(v_1) and integers are absolute positions in the string.

[1]Please refer to [Gulwani 2011] and [Polozov and Gulwani 2015] for a detailed overview of the FlashFill syntax.



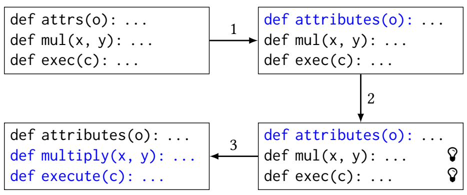
```
def attrs(o): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attrs(o, log): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attrs(o, log): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attrs(o, log): ...
def mul(x, y, log): ...
def exec(c, log): ...
```

(a) Adding a new logger (syntactic only).



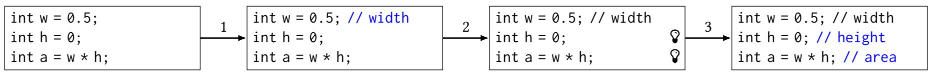
```
def attrs(o): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attributes(o): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attributes(o): ...
def mul(x, y): ...
def exec(c): ...
```

```
def attributes(o): ...
def multiply(x, y): ...
def execute(c): ...
```

(b) Renaming methods to full words (semantic).



```
int w = 0.5;
int h = 0;
int a = w * h;
```

```
int w = 0.5; // width
int h = 0;
int a = w * h;
```

```
int w = 0.5; // width
int h = 0;
int a = w * h;
```

```
int w = 0.5; // width
int h = 0; // height
int a = w * h; // area
```

(c) Adding comments with descriptions (semantic).

Fig. 4. Example scenarios of (1) user making an edit and (2) the system suggesting another location to make the same edit. The user is able to click on the light bulb to see the suggestion and accept it, after which it is applied by the system (3). Today, Blue-Pencil can only do scenario (a). With our proposed integration, it is able to do (b) and (c) as well.

2.2 Refactoring

When changing or refactoring code, developers often find themselves propagating an intended change at multiple places in the codebase. The Blue-Pencil algorithm tackles the problem of repetitive refactoring by making on the fly code suggestions [Miltner et al. 2019]. It looks at what the developer is doing, identifies repetitive edits and makes real time suggestions, rather than requiring the developer to explicitly provide the system with input-output examples. For instance, consider a developer who needs to add a new logger argument to all functions in a project, as shown in Figure 4a. Blue-Pencil sees the user making one edit, recognizes different locations where a function is defined, suggests applying the same transformation at those locations and then automatically applies the transformation if the user accepts the suggestion.

However, Blue-Pencil only supports *syntactic* transformations and fails at tasks that requires semantic knowledge, such as the one in Figure 4b. In this task, a developer wishes to change naming convention from abbreviated forms to full names. Given the ability to learn renaming programs for symbols in the AST, this problem is very similar to learning semantic string transformation programs. Adding semantic operators enables renaming with natural language understanding.

Another repetitive yet crucial task in programming that involves natural language is writing documentation. By extending the transformation language to operate on *full syntax trees*, which keep formatting and comments to guarantee lossless conversion between the syntax tree and source code, these semantic operations become even more powerful. This functionality is illustrated in Figure 4c, where descriptive comments are automatically suggested by synthesizer as it learned a $\text{getDescription}(x)$ function.

2.3 Profiling

The goal of *string profiling* is to learn succinct regular expression patterns that describe a collection of strings. These profiles are useful for a myriad of applications, from checking the quality of data, computing the syntactic similarity between strings, tagging large datasets with column metadata [Song and He 2021] and making string transformation synthesizers more robust by improving the ranking of programs [Ellis and Gulwani 2017] or learning separate programs for examples with different profiles [Padhi et al. 2018].

```
1991/12/31
2005-11-14
October 12, 2005
December 10, 1980
September 25, 1970
1993-06-26
```

(a) Dates

```
iPhone 11 512GB - Red AT&T
iPhone 11 64GB - Red T-Mobile
iPhone 11 128GB - Red Verizon
iPhone 11 128GB - Midnight Green Verizon
iPhone 11 64GB - Silver T-Mobile
iPhone 11 512GB - Space Gray AT&T
```

(b) iPhone 11 data with colors and carrier

Fig. 5. Excerpts of candidate datasets for automatic string profiling.

The recent FlashProfile [Padhi et al. 2018] algorithm uses an inductive synthesis based approach by representing these patterns as programs. Let an atom be a function f : string $\longrightarrow$ int that returns the length of the longest prefix that it matches of a given string. A program is simply a sequence of atoms that matches a string if each atom matches the suffix of matching all preceding atoms. FlashProfile supports syntactic atoms that match constant strings, regular expressions, character classes and arbitrary functions.

As an example, consider the dates in Figure 5a. Two profiles

$$\text{Digit}^4 \circ \text{Punct} \circ \text{Digit}^2 \circ \text{Punct} \circ \text{Digit}^2$$

$$\text{TitleWord} \circ \text{Space} \circ \text{Digit}^2 \circ \texttt{", "} \circ \text{Digit}^4$$

are learned, where $\circ$ denotes the concatenation of atoms, `"quoted"` strings are constants and Digit^n matches n digits. By asking for an output example for each pattern, the number of examples required to transform these dates into a standard format decreases.

Next, consider the strings in Figure 5b. Syntactic patterns struggle to (i) capture symbols in strings and (ii) whether and how to group the words after the `" - "` or not. A semantic pattern can distinguish the colors and carriers without falling victim to irregular characters. An example pattern is

$$\texttt{"iPhone 11 "} \circ \text{Digit+} \circ \texttt{"GB - "} \circ \text{matchColor} \circ \text{matchCarrier}$$

where matchX are anonymous semantic atoms. Our integration allows to learn exactly these kinds of semantic atoms.

3 BACKGROUND

Our proposed integration builds on the idea of decomposing the inductive synthesis problem into smaller subproblems and using the neural model to solve those subproblems that cannot be solved syntactically. The FlashMeta framework performs this kind of synthesis decomposition using *deductive backpropagation* [Polozov and Gulwani 2015]. In this section, we introduce the FlashMeta framework, as well as the specific flavour of neural network that we can use to solve those semantic problems that cannot be further decomposed using FlashMeta. In later sections, we show how to integrate such neural networks in the FlashMeta architecture.

3.1 FlashMeta

All synthesizers that we described in the previous section are instantiations of the FlashMeta framework. More specifically, they are implemented using the publicly available implementation of this framework called PROSE. In the PROSE framework, developers can define a DSL (as a context-free grammar) and provide an executable function for each operator in the DSL. Given a specification of a program, typically as a set of input-output examples, the PROSE framework then provides synthesis strategies to search for a program over this DSL that satisfies the given specification.

The main synthesis strategy is called *deductive backpropagation*, which recursively breaks down a problem into smaller subproblems that, once solved, can be used by a specific operator to solve the bigger problem. The logic of how a problem is to be broken down in subproblems is given by *witness functions* for each argument of each operator in the DSL. Given an operator and a specification, the witness function for a parameter of this operator should return specifications that the parameter should satisfy in order for the operator to satisfy the given specification.

Example 3.1. *Suppose we have an operator* `sum(a, b)` *that sums two integers and a specification that says that the output of this operator on some input σ should be 5, which we write as $\sigma \rightsquigarrow 5$. The witness function for the argument* `a` *of* `sum` *then needs to answer what the value of* `a` *can be, for example, an integer $\in [1, 4]$. It returns a disjunctive specification $\sigma \rightsquigarrow 1 \vee 2 \vee 3 \vee 4$ with all possible values of* `a`*. Following the body of a rule with* `a` *as head in the grammar, the algorithm then continues to look for a way to satisfy this specification—to make* `a` *be one of the allowed values.*

Rather than a single program, PROSE returns a set of programs that satisfy the specification, represented by a version space, and allows operations on these program sets. The intersection between two program sets is an important operation, which allows to compute the set of programs that satisfy multiple specifications. In the original FlashFill setting, this corresponds to learning a program set for each individual row and then taking the intersection over these program sets to find those programs that correctly transform all rows.

Finally, PROSE ranks the programs in the resulting program set, and allows custom scoring functions to be specified for each operator. The final score is computed bottom-up, with scoring functions for operators typically aggregating the scores obtained for their arguments.

Example 3.2. *In the previous example, we can assign higher scores to lower constant numbers and use the scoring function for* `sum(a, b)` *to assign calls in which* `a < b` *a higher score.*

It is exactly this breaking down of a problem in smaller subproblems, intersection over different examples and ranking that makes the PROSE framework an excellent candidate for integrating semantic operators powered by few-shot learning neural networks. In order to do so, we need to (i) define the operators with their semantics, (ii) define witness functions that specify how to learn the arguments to these operators and (iii) describe how programs with semantic operators should be ranked.

3.2 Generative Language Models

In language modelling, a common task is to predict the next token for a given set of input tokens. Repeated application of this process allows large language models to effectively generate text when given an initial sequence of context tokens. Such autoregressive language generation quickly generated popularity with the impressive results obtained by the GPT models, which combine a transformer architecture, unsupervised pre-training, and millions of parameters.

Recently, it was shown that such models are able to learn a task by encoding a few input and output examples of this task in the context [Brown et al. 2020]. An example task was already shown in Figure 1a, where the goal was to map the first three letters of a month to its full name. More general examples of tasks are machine translation, question answering and determining which word a pronoun refers to. This ability to detect the task is called *in-context learning* and doing it from few examples is called *few-shot learning*.

3.2.1 Abilities. We observe three tasks that GPT-3 is able to learn—closed book question answering, natural language understanding and text classification—and that can be used to create useful semantic operators.

Q: What does a manometer measure?
A: pressure
Q: A pickerel is a young what?
A: *pike*

(a) Closed book QA.

Q: What does a manometer measure?
A: measure
Q: A pickerel is a young what?
A: *is*

(b) POS tagging through QA.

Lebron James => True
Lionel Messi => False
Christiano Ronaldo => False
Kobe Bryant => *True*

(c) Text classification.

Fig. 6. Tasks performed using GPT-3. *Emphasized* text is the (expected) response.

Closed book question answering. In closed book question answering, the goal is to answer a question about factual knowledge without access to a document that contains evidence [Roberts et al. 2020]. An example question from the TriviaQA dataset [Joshi et al. 2017] is "What does a manometer measure?" and the expected answer is "pressure". It was shown that large language models are able to store a lot of such knowledge in their parameters, with GPT-3 beating fine-tuned models that have explicit access to Wikipedia [Brown et al. 2020]. Figure 6a shows a query for one-shot QA using GPT-3. We exploit this knowledge to learn semantic mappings, such as mapping countries to their language code in the $\mathsf{getCountryCode}(x)$ function or expanding abbreviated month names in the $\mathsf{getFullName}(x)$ function.

Natural language understanding. We consider natural language understanding to encompass concrete tasks like part-of-speech tagging, role labeling and cloze tests. As GPT-3 is a generative model, these tasks are also framed as a QA prompt, but a different skill is required to solve it. Figure 6b shows an example of such prompt, where the goal is to tag the predicate of a question. We exploit this ability to recognize parts of sentences for extracting semantic locations in the input, for example, in the $\mathsf{getPositionLeftOfPronoun}(x)$ function.

Classification. Finally, GPT-3 is able to perform text classification in a similar fashion. An example of classifying whether an athlete is a basketball player or a soccer player is shown in Figure 6c. We exploit the ability to semantically classify text to learn semantic matching functions, such as the $\mathsf{matchColor}()$ atom.

3.2.2 Prompts as functions. We consider the GPT-3 model as a function $\mathsf{M} : \mathsf{string} \to \mathsf{string}$ that takes a prompt and returns the most likely continuation of the prompt. All tasks that we defined use this function in a very similar way. A few input-output examples are encoded in a question-answer format, an input example is appended as a question and the output is conditioned on the allowed tokens. We can consider this as a function $\mathsf{QA} : \mathsf{string}^2[\,] \times \mathsf{string} \to \mathsf{string}$ that combines its first two arguments in a single string and then calls M to obtain its result.

The template that is used for generating the prompt—a function that takes one input-output example and returns a string—is a hyperparameter. For a fixed set of input-output examples, the only free variable is a new input example and we have obtained a semantic string $\to$ string function.

Example 3.3. *When given some input-output examples* $BG = \{(\texttt{"France"}, \texttt{"€"}), (\texttt{"Japan"}, \texttt{"¥"})\}$, *the unary function* $getCurrencySymbol(x) \equiv QA(BG, x)$ *extracts the currency symbol of a country. Given a new string* `"USA"`, *we get*

$$getCurrencySymbol(\texttt{"USA"}) = M\begin{pmatrix} \texttt{"France => €} \\ \texttt{Japan => ¥} \\ \texttt{USA =>"} \end{pmatrix} \simeq \texttt{"\$"}$$

where $\simeq$ *indicates the expected answer.*

Algorithm 1 Build query from examples

```
Require: list of tuples $Q$
Require: template T : string × string → string
  function BuildPrompt($Q$, T)
      $Q \leftarrow$ Map($\lambda(q, a) \Rightarrow$ T$(q, a)$, $Q$)
      return Join($Q$, "\n")
```

Algorithm 2 Semantic function through QA

```
Require: list of examples $E$ and new input $x$
Require: list of allowed output tokens $L$
  function QA_T($E$, $x$, $L$)
      prompt ← BuildPrompt($E + (x, \epsilon)$, T)
      return M(prompt, $L$)
```

```
language FlashFill;
using FlashgGPT3;

@output string start := e | std.ITE(cond, e, start);

string e := f | Concat(f, e);
string f := ConstStr(w) | SubStr(v_i, pp) | SemMap(v_i, Q);

Tuple<int, int> pp     := std.Pair(pos, pos);
int pos                := AbsPos(x, k) | RegPos(x, rr k) | SemPos(x, Q, m);
Tuple<Regex, Regex> rr := std.Pair(r, r);

bool cond := Match(v_i, r, k) | SemMatch(v_i, P, N);

@input string[] vs;   string w;    int k;    Regex r;
Tuple<string, string>[] Q;   string[] P,N;   string m;
```

Fig. 7. DSL $\mathcal{L}_{SF}$ for FlashFill with semantic operators. Changes with respect to the original $\mathcal{L}_{FF}$ are printed in **bold**. We use F$(v_i, -)$ as shorthand notation for let s = std.Kth(vs, i) in F$(s, -)$.

4 SEMANTIC OPERATORS

We introduce three generic semantic operators that each exploit one of the identified abilities of GPT-3. Each of these operators builds a prompt, performs a query and parses the result. The data used to construct the prompt is made an argument of the operators. Learning a specific operator, such as extracting country codes, then corresponds to learning the argument.

As FlashMeta is designed to support operator reuse, these operators can be easily integrated into another DSL. Throughout this section, we use integration with the FlashFill DSL [Gulwani 2011; Polozov and Gulwani 2015] to illustrate the new operators. The augmented DSL is shown in Figure 7. We use the following syntactic sugar for readability purposes; top-level conditionals are omitted when not applicable, the shorthand notation F$(v_i, -)$ is used over `let s = std.Kth(vs, i) in F(s, -)` and a list of tuples of strings is simply called a `Query`.

4.1 Maps

The semantic map operator SemMap(v, Q) is used to look up semantic properties of the input. Listing 1 show its executable semantics, which simply calls GPT-3 using Q as examples and v as the new question. Semantic map requires no additional logic on top of the QA function.

Example 4.1. *Let us revisit the example of country codes from Figure 1c. The getCountryCode(x) function can be easily represented as a semantic map by building an appropriate set of input-output examples Q. The full program becomes:*

```
string SemMap(string v, Query Q) {
    return QA(Q, v);
}
```

Listing 1. Executable semantics of the semantic mapping function.

```
int SemPos(string x, Tuple<string, string>[] Q, string d) {
    string answer = QA(Q, x);
    MatchCollection ms = x.Matches(new Regex(answer))[0];
    if ms.Count() != 1:
        return null;
    return (d == "L") ? ms[0].Index : ms[0].Index + x.Length;
}
```

Listing 2. Executable semantics of the semantic position logic. The `AllWordsIn` function extracts all words from the given list of strings, which are used to constrain the output to return only words in x.

```
ConstStr("(") ∘ SemMap(v2, Q) ∘ ConstStr(") ") ∘ SubStr2(v1, NumTok, 1)
               ∘ ConstStr(" ") ∘ SubStr2(v1, NumTok, 2)
Q = [("Taiwan", "886"), ("Spain", "34"), ("South Korea", "82")]
```

Example 4.2. *In textbooks and course notes on grammar, examples of irregular forms are often provided. For grammatical constructs with many irregular forms, such as plurals or tenses, generating and formatting these examples is a very repetitive task. We can use the power of language models to easily generate formatted examples. Consider, for example, a table of comparative and superlative adjectives.*

Input v_1	*Output*
good	*good – better – best*
old	*old – older – oldest*
many	*many – more – most*

Using the power of lookup, such tables can be easily generated from just the base adjective with the following transformation program:

```
v1 ∘ ConstStr(" - ") ∘ SemMap(v1, Q1)
   ∘ ConstStr(" - ") ∘ SemMap(v1, Q2)
  Q1 = [("good", "better"), ("old", "older"), ("many", "more")]
  Q2 = [("good", "best"), ("old", "oldest"), ("many", "most")]
```

4.2 Position logic

Position logic is used to determine interesting locations in the input, which is useful in tasks that involve formatting and extraction. Semantic positions are similar to regular expression positions. We use the model to select a substring from the input and return either the left or right position of that substring. The output is constrained to be a substring of the input. Listing 2 shows the executable semantics of position logic.

Example 4.3. *Consider exercises in which a specific part of a sentence has to be underlined or emphasized. Mapping the sentence to the correct word is not sufficient, the position of the word is required to build the correct output string.*

```
bool SemanticMatch(string x, string[] P, string[] N) {
    p = Map(λp => new Tuple<string, string>(p, "True"), P);
    n = Map(λn => new Tuple<string, string>(n, "False"), N);
    string answer = QA(p + n, x);
    return answer == "1";
}
```

Listing 3. Executable semantics of the semantic condition function.

v_1	*Output*
Dogs are great.	*Dogs \emph{are} great.*
I love dogs.	*I \emph{love} dogs.*
The dog barked really loud.	*The dog \emph{barked} really loud.*

The following transformation program uses the semantic positioning logic to build the output from five parts. Four different `SemPos` *invocations are used, but they only require one call to the semantic model by caching the results. In the query, we denote the string in row i and column* v_1 *by* `Ii`.

```
  SubStr(v1, std.Pair(AbsPos(0), SemPos(v1, Q, "L")))
∘ ConstStr(" \emph{")
∘ SubStr(v1, std.Pair(SemPos(v1, Q, "L"), SemPos(v1, Q, "R")))
∘ ConstStr("}")
∘ SubStr(v1, std.Pair(SemPos(v1, Q, "R"), AbsPos(-1)))
Q = [(I1, "are"), (I2, "love"), (I3, "barked")]
```

Example 4.4. *Conversely, we can also start from a sentence and generate exercises. This requires both semantic positions and mapping.*

Input v_1	*Output* o
A bird is smaller than a dog.	*A bird is (smaller/smallest) than a dog.*
He had the worst cold ever.	*He had the (worse/worst) cold ever.*
Jogging is faster than walking.	*Jogging is (faster/fastest) than walking.*

The following program uses semantic position logic on the adjective to allow the output to be composed from parts of the input and interjecting constants, and a semantic map to obtain the comparative and superlative forms of the adjective.

```
  SubStr(v1, std.Pair(AbsPos(0), SemPos(v1, Q1, "L")))
∘ ConstStr("(")
∘ SubStr(v1, std.Pair(SemPos(v1, Q1, "L"), SemPos(v1, Q1, "R")))
∘ ConstStr("/") ∘ SemanticMap(v1, Q2) ∘ ConstStr(")")
∘ SubStr(v1, std.Pair(SemPos(v1, Q1, "R"), AbsPos(-1)))
Q1 = [(I1, "smaller"), (I2, "worse"), (I3, "faster")]
Q2 = [(I1, "smallest"), (I2, "worst"), (I3, "fastest")]
```

4.3 Conditions

The top-level statement decides which expression to use for constructing the output. Classically, this was done by learning a pattern based on regular expressions that some column must satisfy. Given a set of positive and negative examples, we use GPT-3 to learn how to classify them by mapping them to `"True"` or `"False"`. The semantics are shown in Listing 3.

Example 4.5. *Consider a dataset of athletes and game scores, but the sport itself was lost. We need to distinguish whether a player makes goals or points by deciding their sport. Rather than explicitly mapping an athlete to a sport, GPT-3 implicitly learns to make the distinction.*

v_1	v_2	*Output*
Christiano Ronaldo	*1026*	*1026 goals*
Lebron James	*34852*	*34852 points*
Lionel Messi	*711*	*711 goals*

The following transformation program uses the semantic conditional to distinguish whether the `" goals"` *or* `" points"` *constant should be used.*

```
std.ITE(SemanticMatch(v₁, P, Q),
        v₂ ∘ ConstStr(" goals"), v₂ ∘ ConstStr(" points"))
P = ["Christiano Ronaldo", "Lionel Messi"]
N = ["Lebron James"]
```

Example 4.6. *In this classical example, the goal is to extract the month from dates. Depending on localization, however, the month is in a different location. The third format can be distinguished using only syntax, but the first two rows require a semantic condition to decide whether to use the American standard or not.*

v_1	v_2	*Output*
Chicago	*01/02/1990*	*01*
Brussels	*01/02/1991*	*02*
Beijing	*1992-03-02*	*03*

```
std.ITE(SemanticMatch(v₁, P, N),
        SubStr(v₂, std.Pair(AbsPos(0), AbsPos(2))),
        std.ITE(Match(v₂, "-", 1),
                SubStr(v₂, std.Pair(AbsPos(5), AbsPos(7))),
                SubStr(v₂, std.Pair(AbsPos(3), AbsPos(5)))))
P = ["Chicago"]
N = ["Brussels", "Beijing"]
```

5 LEARNING SEMANTIC OPERATORS

Learning semantic operators boils down to selecting the right data to build the prompt, which is done by implementing their witness functions. Recall that witness functions are used to determine a specification on the parameters of an operator, given a specification of the operator. In other words, if an operator $f(x_1, x_2)$ must satisfy a specification φ, the witness functions w_1 and w_2 for the arguments x_1 and x_2 must determine new specifications φ_1 and φ_2 that the arguments must satisfy for this to be true. The deductive backpropagation algorithm uses these witness functions to recursively break down a synthesis problem into smaller synthesis problems.

The intuition behind our integration is using the semantic model to solve these subproblems that cannot be solved in any other way. There are two main challenges; given the specification, the witness function is unable to determine (i) whether syntactic operators are able to solve this specification or not, and (ii) whether GPT-3 is able to solve any given problem without performing a lot of queries, as performing queries is both slow and expensive.

In order to solve both challenges, we consider the model to be an oracle that always gives the correct answer during training. Ranking is used to select programs that perform few different calls to the model and in which the smallest number of output characters is obtained through semantic operators. We refer this technique as *deferred query execution.*

All semantic operators have an argument x that corresponds to the input that should be mapped. For example, in a spreadsheet context, it is one of the input columns. Witness functions are conditional on the selected input, which is taken care of by the DSL in which the operator is integrated. For example, in the spreadsheet context and $\mathcal{L}_{\mathsf{FF}}$, the `let` statement selects a column and the witness function is given only this column. When there are multiple inputs, selecting the (most likely) correct input can be done through ranking.

In the following sections, all specs are a conjunction of $x \rightsquigarrow V$ with x the input string and $V = \{v_1, \ldots, v_n\}$ the atoms in a disjunction, all of the return type of the operator. For example, in an outer specification for SemMap, V is a set of strings. As all semantic operators are terminal, their witness functions return a list of possible values for the argument that they witness, instead of a mapping from states to values—these values should hold for all states.

5.1 Learning maps

In order to learn a semantic map, we need to learn the query given a set of disjunctive specifications over strings. When each disjunction consists of a single element, the witness function is trivial—simply map the input to each of these strings. When multiple options are possible, however, it becomes more challenging. In order to keep the witness function complete, all combinations of queries obtained by picking one option from each disjunction have to be considered.

If the semantic map is required closer to the root of the target program, other operators depend on the exact query to build the rest of the output. For instance, if the semantic map is required in the first argument of a `Concat` statement, for each possible query, a new branch is started, which can again contain semantic maps.

Example 5.1. *Consider this spec obtained from the witness of the first argument of* `Concat` *in* $\mathcal{L}_{SF}$.

$$\texttt{Japan} \rightsquigarrow \texttt{"¥"} \lor \texttt{"¥ "} \lor \texttt{"¥ 10"}$$
$$\texttt{France} \rightsquigarrow \texttt{"€"} \lor \texttt{"€ "} \lor \texttt{"€ 20"}$$

Three out of nine possible queries are

```
[("Japan", "¥"), ("France", "€")]
[("Japan", "¥ 10"), ("France", "€")]
[("Japan", "¥"), ("France", "€ ")]
```

and it is impossible to know which one is correct. If this semantic map is the first argument of a `Concat` *statement and the second query is chosen, the spec sent to the second argument is*

$$\sigma_1 \rightsquigarrow \texttt{""} \land \sigma_2 \rightsquigarrow \texttt{" 20"}$$

and in a similar way, nine branches are started for the second argument.

In order to minimize the number of possible queries and make the learning tractable, we sacrifice completeness for speed and perform greedy clustering over the possibilities from each disjunctive spec. Each cluster contains exactly one string from every disjunct. The spec with highest number k of possibilities is taken as a reference and used to initialise k clusters. From all other specs, k possibilities are greedily assigned to each cluster using a similarity function between two strings. The greedy clustering algorithm and witness function are shown in Algorithm 3.

Example 5.2. *In the running example, using syntactic similarity based on occurrence of tokens, we get three clusters* {{"¥","€"}, {"¥ ","€ "} *and* {"¥ 10","€ 20"}.

Algorithm 3 Learning the semantic mapping query.

```
Require: Similarity function S : string × string → $\mathbb{R}$
 1: function GreedyCluster(Y)
 2:     reference ← longest in Y
 3:     C ← {[e] | e ∈ reference}                               ▷ Initialize clusters with reference
 4:     for $Y \neq$ reference ∈ Y do
 5:         C′ ← C                                               ▷ Make shallow copy of clusters
 6:         while C′ ≠ ∅ do
 7:             $e^*, C^* \leftarrow \arg\max_{e \in Y, C \in C'} S(e, C[0])$   ▷ Unassigned element closest to reference
 8:             append $e^*$ to $C^*$
 9:             remove $C^*$ from C′
10:     return C
11: function WitnessMapQ($\varphi$)
12:     options ← {$Y \mid x \rightsquigarrow Y \in \varphi$}
13:     clusters ← GreedyCluster(Y)
14:     return MakeQueries(clusters, $\varphi$)                   ▷ Map states to elements from clusters
```

5.2 Learning position logics

Learning the query to extract a position starts from a set of disjunctions over the positions to extract. The direction (left or right) is given. Let p be a position and s the string. Depending on direction, we generate strings $s[p : p + j]$ or $s[p - j : p]$ for increasing j as candidates values for the query. Instead of all j, we only select interesting candidates for j by tokenizing the string $s[p :]$ or $s[: p]$ with a tokenizer that extracts interesting positions, for example, on word boundaries.

Example 5.3. *Given the spec* `"He wanted to eat pizza."` $\rightsquigarrow 3$ *and direction left, the string* `"wanted to eat pizza."` *is tokenized into* [`"wanted"`, `"to"`, `"eat"`, `"pizza"`] *and possible j are {6, 9, 13, 20}. Three candidate values for the query are* `"wanted"`, `"wanted to"` *and* `"wanted to eat"`.

Even with low values of j, the number of different queries quickly increases. The same greedy clustering approach used to learn maps is also used to select a subset of promising queries. The witness function for learning a left position is shown in Algorithm 4. Learning a right position is almost identical, with line 2 selecting sides $x[: p]$ and line 3 using a function that extracts interesting right positions.

Algorithm 4 Learning a position query for left direction.

```
Require: tokenizing function LeftPositions : string → int[]
 1: function WitnessPosQueryLeft($\varphi$)
 2:     sides ← {$x \rightarrow \{x[p :] \mid p \in V\} \mid x \rightsquigarrow V \in \varphi$}
 3:     tokens ← {$x \rightarrow \{\text{LeftPositions}(s) \mid s \in S\} \mid (x \rightarrow S) \in$ sides}
 4:     candidates ← {$x \rightarrow \sum t[: j] \mid (x \rightarrow T) \in$ tokens, $t \in T, j \in [1, |t|]$}
 5:     clusters ← GreedyCluster(candidates)
 6:     return MakeQueries(clusters, $\varphi$)
```

5.3 Learning conditions

The witness for learning conditions is given a conjunction of $\sigma \rightsquigarrow \mathbb{B}$ specs. All inputs mapped to true correspond to positive examples and vice versa. In learning conditions, the main challenge is knowing when to learn a different program. The FlashFill paper [Gulwani 2011] describes a procedure based a partitioning of the input with (i) each partition having a program that is consistent with the specification for that partition and (ii) having the fewest number of partitions. Because of deferred querying and SemMap acting like an oracle during learning, it will always yield a program that is consistent with any partitioning of the input. We assume that semantic conditions are only used to distinguish otherwise syntactic programs. Both examples shown in Section 4.3 satisfy this assumption.

5.4 Ranking

After learning programs, we rank them with respect to the following criteria; (i) rely as much as possible on syntactic operators and (ii) having as few distinct queries to the model as possible. Because semantic operators are considered oracles during learning, the synthesizer can use them to *solve* any subproblems with the appropriate type, but we only want to use them for subproblems that cannot be solved syntactically.

Example 5.4. *Consider a problem* `"Dogs are great."` → `"Dogs \emph{are} great."` *that requires extracting the predicate. We need to learn two semantic positions (5 and 8) that are used in three substring operators. When learning the first semantic position, given the left direction, two possible outputs for queries are* `"are"` *and* `"are great"`*. Similarly, for the second position and given right direction, two options are* `"Dogs are"` *and* `"are"`*. For all combinations of queries, a valid program will be learned, but only by selecting* `"are"` *will the query perform exactly the desired task (first criterion). Note that the constant* `"\emph{"` *and other parts of the output can also be the result of a semantic map, hence the goal to rely as much as possible on syntactic operators (second criterion).*

To support easily integrating semantic operators, we want the ranking to be as independent as possible from the ranking of other operators. This independence is achieved by assigning semantic operators a score of 1 during the hierarchical ranking, with the goal of having a minimal influence in both additive and multiplicative aggregation of ranks. After ranking, programs are re-ranked based on semantic operators.

Map queries are punished based on the number of characters that they are expected to output. Let Q be the list of input-output examples and $\#c = \sum_{(i,o) \in Q} |o|$ the number of characters obtained through this query. We add a bonus score for not requiring characters through semantics maps $S_m/\#c$ to the original score.

Position and condition queries are punished based on the number of distinct position queries. For example, extracting the left and right side of the same word only requires a single query. A second bonus score $S_q/\#q$ is added to the final score, with $\#q$ the number of distinct queries required by semantic position and condition operators.

6 EVALUATION

We perform experiments to answer the following questions.

Q1 Is a combination of syntactic and semantic parsing required?
Q2 Do we need deferred execution of queries and greedy clustering to quickly learn programs?
Q3 Case study: can we generate descriptive names for semantic operators?
Q4 Case study: can we easily use semantic operators in other domains?

6.1 Environment

6.1.1 Implementation. Starting from a string transformation DSL on the PROSE website, we added the position and map operators in a prototype implementation called FlashGPT3. A syntactic similarity measure was used in the greedy clustering for possible queries. We compute the number of occurrences of tokens to generate a feature vector and compute a similarity as the cosine similarity between these vectors. Tokens are lowercase words, uppercase words, camel case words, numbers and a list of specific symbols.[2]

6.1.2 Benchmark Suite. Three types of benchmark problems are collected. The first are *head cases* used to evaluate basic capabilities that we believe a semantic transformation synthesizer should offer. This type of head cases was also used to evaluate TDE [He et al. 2018]. The second are examples related to the language learning domain and take inspiration from course notes on English grammar and websites with worksheets.[3][4][5][6] Together, these make up a collection of 30 diverse and challenging problems that require a combination of syntactic and semantic parsing. Finally, we use a subset of 30 of the internal FlashFill benchmark to evaluate the syntactic capabilities of GPT-3.

6.1.3 Model and hyperparameters. We use the largest *davinci* model with 175 billion parameters and set the temperature parameter, which roughly determines the level of creativity of the generated output, to 0.[7] All experiments were performed on a laptop.

6.1.4 Inference time. The run-time of FlashGPT3 programs on new inputs is heavily dominated by calls to the GPT-3 API. Across all experiments an average, we have reported an average query time of 462 ± 271 ms. Asynchronous execution speeds this process up for multiple rows, as the overhead is largely caused by network overhead. We have found that performing more than four concurrent invocations is met with rate limiting. This limitation stems from the fact that a single endpoint is responsible for serving all GPT-3 calls in the world. Commercially shipping FlashGPT3 is then possible through a dedicated endpoint that does not limit concurrent requests.

6.2 Combining syntactic and semantic operators

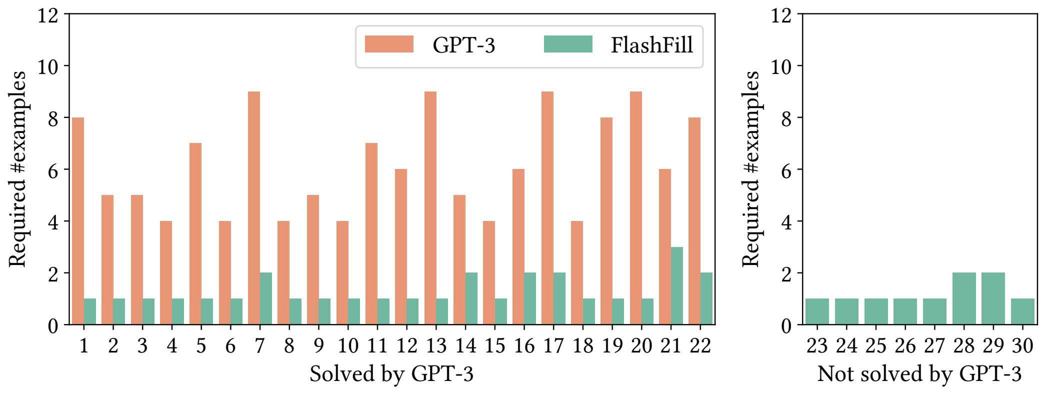
As GPT-3 is able to perform syntactic manipulations, we start by evaluating whether it is required to combine it with syntactic inductive programming or not, and argue why such an integration is relevant regardless of GPT-3 being able to solve some problems on its own.

6.2.1 Experimental Setup. Our evaluation takes the first n examples to learn programs, and uses the top-ranked program to try solving all remaining examples. Once a program is obtained that solves all remaining cases, execution is stopped. Experiments with only GPT-3 are performed by a DSL that only contains the SemMap operator. We use the version of FlashFill shipped with the PROSE SDK [Microsoft 2015].

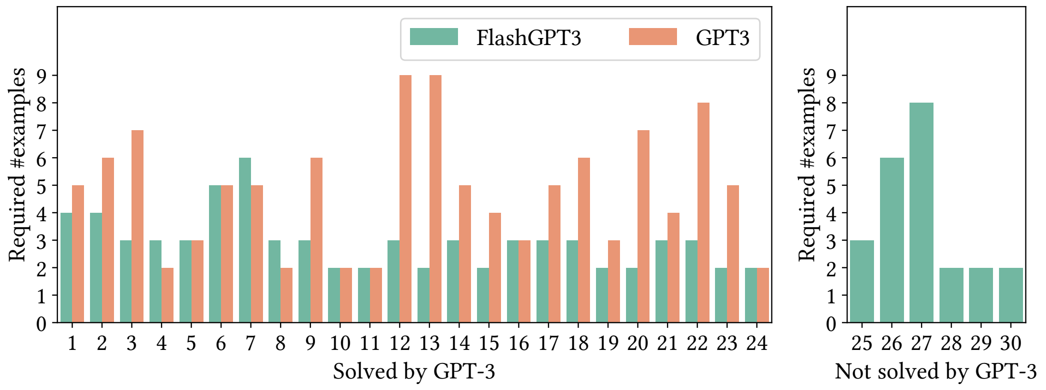
6.2.2 Results. Figure 8a shows the number of examples that FlashFill and GPT-3 require on the syntactic benchmarks. When the number of examples equals the total number of examples, GPT-3 fails. The result indicates that GPT-3 is able to solve some problems, but requires significantly more examples. Its main weaknesses are tokenization and complex substring extraction logic. A problem as simple as extracting the first 4 letters of a word (`"Alakazam"` → `"Alak"`) is not solved after 7 examples. Similarly, extracting the penultimate word from a path specification (`"path/to/file"` → `"to"`) proves hard for the language model.

Figure 8b shows the number of examples that FlashGPT-3 and GPT-3 require on the semantic benchmarks. FlashGPT3 consistently outperforms GPT-3 on problems that require more complex

[2] `,.;'+*_!()"/\‘ ?%@#$[]{}<>=`
[3] www.perfect-english-grammar.com/grammar-exercises.html
[4] www.agendaweb.org/grammar-exercises.html
[5] www.englisch-hilfen.de/en/exercises_list/alle_grammar.htm
[6] www.english-4u.de/grammar_exercises.htm
[7] As used in the demonstration on QA on the OpenAI website (https://beta.openai.com/examples/default-qa).



(a) Solving FlashFill benchmarks with GPT-3. Eight problems (26%) were not solved by GPT-3. Those problems that are solved by GPT-3 require significantly more examples than FlashFill. It is clear that a strong, syntactic synthesizer is required to automate repetitive transformation tasks.

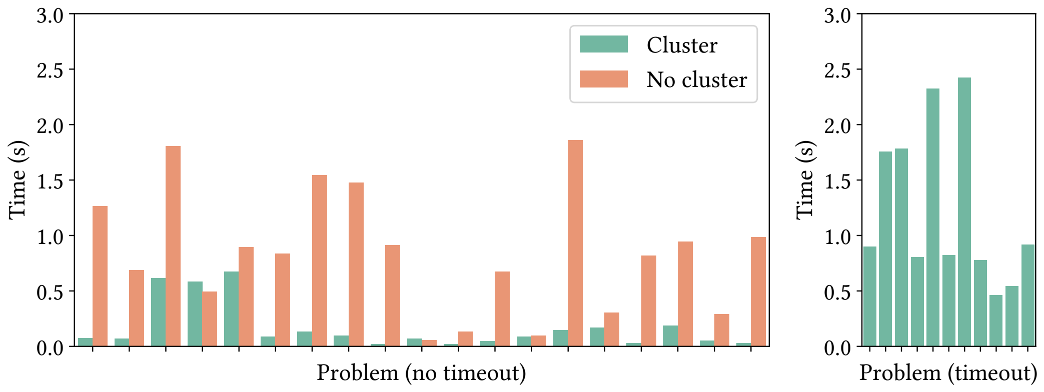


(b) Results on the mixed syntactic and semantic benchmark problems. Six problems (20%) were not solved by GPT-3. On half of those, FlashGPT3 requires barely two examples. On others, the bottleneck is GPT-3 requiring more examples to solve the semantic subproblems.

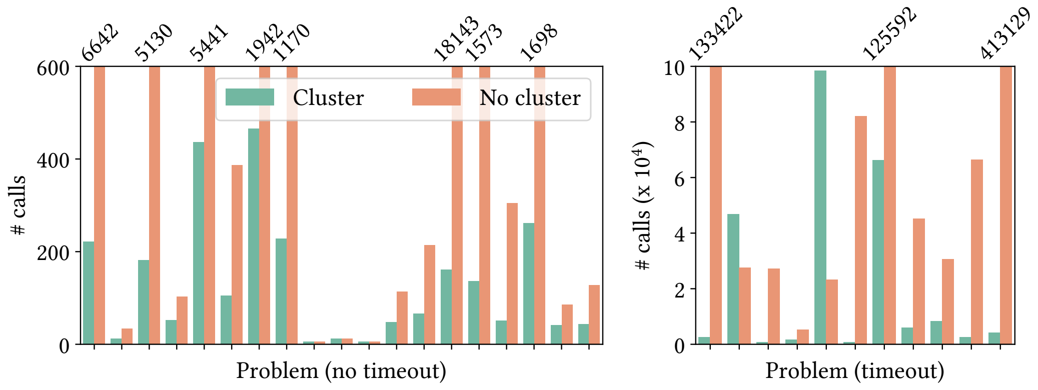
Fig. 8. Results on syntactic and mixed benchmarks.

syntactic parsing. Aside from taking care of the syntactic part, deductive backpropagation has the advantage of generating smaller, more targeted problems for GPT-3. For instance, explicitly obtaining the infinitive of a verb (`"were"` → `"be"`) is easier than requiring this transformation as part of a larger problem (`"were"` → `"data/be.mp3"`).

Whereas generally a blessing, in some cases, these smaller subproblems sometimes lack enough context for GPT-3 to learn the task. For instance, consider the problem of converting 24-hour to 12-hour notation (`"22:00"` → `"10:00 PM"`). FlashGPT3 breaks this down into two semantic subproblems `"22:00"` → `"10"` and `"22:00"` → `"PM"` as the space is considered a constant. The context of this task is important, as it takes FlashGPT3 one more example to solve the first of these problems as opposed to the whole problem at once (6 versus 5). Note that the program using only a single map is also discovered during synthesis. Cross-validating programs during ranking allows to trade off performing more queries for requiring fewer examples.



(a) Total time taken to learn a program (in seconds). Evaluation time is not counted. Problems in the right figure timed out without clustering. Clustering is clearly required for learning problems in reasonable time.



(b) Semantic operator calls during learning, spread out over two plots for clarity. Problems on the left side did not time out before finding the correct program without clustering, those on the right did.

Fig. 9. The effect of clustering on the number of calls (not) made and the time taken to learn a program. Each data point is the sum over invocations with increasing number of examples until a solution is found. Even with clustering, the number of calls is far too high for practical purposes. Deferred querying brings the number of calls down to zero.

During evaluation, the syntactic guarantees of learning a program with FlashGPT3 allowed us to correct syntactic mistakes in the benchmark, such as trailing or missing spaces. Despite showing decent performance on some syntactic problems, these kind of syntactic guarantees are unavailable when relying only on GPT-3.

6.3 Deferred execution and clustering

After recursively solving the disjunctive specs from the witness functions of an operator, deductive backpropagation performs a *soundness check* on these arguments by executing the operator on these arguments, before witness functions dependent on the result of this operator are invoked. We evaluate whether deferring the execution of queries until after ranking and clustering queries is required to quickly learn programs or not.

```
Describe the relation between the following items.

Belgium | Brussels => capital
Pizza is delicious. | Pizza => subject
Lionel messi | football => sport
China | Asia => continent
Like a Prayer | Madonna => artist
x | y =>
```

Listing 4. Prompt to extract the name of a relation, where x and y are placeholders to be substituted with the input and output of an example from a query.

6.3.1 Experimental setup. We run all experiments with greedy clustering replaced by the Cartesian product over all possible queries. Learning is timed out after five minutes. During learning, we count how often a query would have been made by the synthesizer.

6.3.2 Results. Figure 9a shows the total time taken to learn the correct program. Bars that do not fit on the plot are instances where learning timed out. Without clustering, that happens for 11 problems. With clustering, most programs are learned in less than a second. Only a few problems, involving long sentences and requiring more examples to be learned, take slightly longer, but are still learned in less than three seconds. Using cheap language models to improve the clustering step on these instances can still improve performance.

Figure 9b shows the number of times the semantic operators were invoked during learning, both with and without clustering. In other words, this plot shows the number of calls *not* made to the model by having these operators act as oracles during learning. They are divided over two plots for clarity, based on whether learning without clustering timed out or not.

Without clustering, the number of calls is prohibitively high. Even with clustering, however, the number of calls quickly grows to tens of thousands for complex programs that require more examples. Such calls are both slow and expensive, and learning will still be slow for all but the smallest problems. Using deferred query execution, the number of calls drops to zero and programs are learned quickly.

6.4 Case study: renaming semantic operators

Running examples in this paper use descriptive names for semantic operators, but the actual operators are anonymous and represented by a query. In this case study, we explore using GPT-3 to rename semantic operators with descriptive names based on the examples in these queries.

Example 6.1. *A transformation* `SemMap(x, [("UK", "£"), ("Japan", "¥")])` *is not very readable. Using GPT-3, we can rename this to* `getCurrency(x)`.

Listing 4 shows a prompt in which each examples describe the name of a relation between two concepts. If the two placeholders x and y are replaced with one of the input-output example in a query, it hopefully returns a descriptive name for the operator of that query. Rather than only the best completion, we ask for the top-k completions and rank them by how often they occur. The temperature is set to 0.8 to obtain a greater variety of possible names.

Example 6.2. *Setting* x = `"UK"` *and* y = `"£"`, *the top-10 results are currency, currency, currency, currency, currency, currency, currency, currency, currency and country.*

Table 1 shows the names obtained using this method for some of the examples in our evaluation, for queries performed by FlashGPT3 and GPT-3. Names on the more specific FlashGPT3 queries are more accurate when compared to using GPT-3, which attempts to solve the whole problem at once.

Table 1. Using GPT-3 to generate names for semantic operators. We write $\cdots$ for parts of sentences that are omitted for brevity. Because it generates concrete subproblems, names for FlashGPT3 are more accurate.

Input	FlashGPT3		GPT-3	
	Output	Name	Output	Name
were	be	infinitive	data/be.mp3	to be
May 2, 1953	mai	month	29 mai 1953	date
How many mice does your cat catch?	mouse	subject	$\cdots$ ____ $\cdots$ (mouse)	subject-verb
Guernica	Picasso	painter	Picasso's Guernica	artwork
1984	Orwell	writer	Orwell's 1984	book
PRG	Prague	airport	Departure from Prague (PRG)	airport code

6.5 Case study: String profiling

This section presents a case study where we use our semantic operators to perform semantic string profiling. Recall that for the profiling task in Figure 5b (Section 2.3), we wanted to learn the following semantic profile

$$\texttt{"iPhone 11 "} \circ \text{Digit+} \circ \texttt{"GB - "} \circ \text{matchColor} \circ \text{matchCarrier}$$

that represents a concatenation of atoms. We extend FlashProfile with a SemPos atom that finds the next ending position of a semantic concept.

Example 6.3. *The matchColor atom can be represented by* $\texttt{SemPos}(x, Q, \texttt{"R"})$ *with*

$$Q = \texttt{[("Red AT\&T", "Red"), ("Space Gray AT\&T", "Space Gray")]}.$$

To add an atom to FlashProfile, we need a function that takes a set of strings $\mathcal{S}$ and returns a set of compatible atoms with the prefixes of those strings. This is achieved by creating a disjunctive spec that maps each string to all possible locations and then uses the witness for SemPos with a semantic similarity measure, for example, cosine similarity between embeddings [Mikolov et al. 2013]. Finally, we select only the cluster with the highest inter-cluster similarity.

Example 6.4. *For the leading example on profiling, we generate the following specs.*

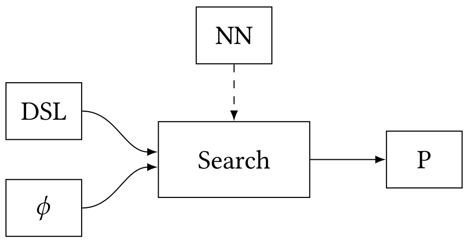
$$\begin{aligned} \texttt{"Red AT\&T"} &\rightsquigarrow 3 \vee 6 \vee 8 \\ \texttt{"Midnight Green Verizon"} &\rightsquigarrow 8 \vee 14 \vee 22 \\ \texttt{"Space Gray Unlocked"} &\rightsquigarrow 5 \vee 10 \vee 19 \end{aligned}$$

If we compute the similarity between strings as the cosine similarity between their average word embedding, the following clusters are obtained with the GreedyCluster *algorithm.*

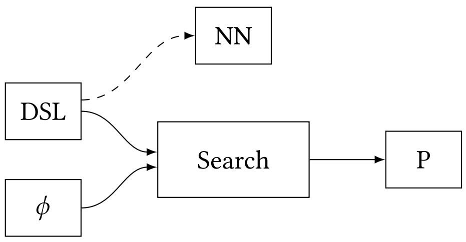
$$\begin{gathered} \{\texttt{"Red"}, \texttt{"Midnight Green"}, \texttt{"Space Gray"}\} \\ \{\texttt{"Red AT"}, \texttt{"Midnight Green"}, \texttt{"Space Gray"}\} \\ \{\texttt{"Red AT\&T"}, \texttt{"Midnight Green"}, \texttt{"Space Gray"}\} \end{gathered}$$

The first cluster achieves the highest inter-cluster similarity and is selected to build the atom.

After we find the pattern for the colors, we can perform a similar step for the carriers. Notice that these SemPos atoms are generic, however, and in order to be useful to users, they are ideally given a descriptive name. Using the query from Section 6.4, our system is able to do exactly that.



(a) Classical integration of neural networks with program synthesis. The NN is used to bias either the search or the DSL.



(b) Our proposed integration of neural networks with program synthesis. The DSL uses a neural model that allows few-shot learning to perform semantic operations.

Fig. 10. Integrating neural networks with inductive synthesis where the goal is to learn a program P over a given DSL that satisfies a specification ϕ.

7 RELATED WORK

Inductive program synthesis. Learning to write programs from demonstrations has been a popular research area for a long time [Cypher and Halbert 1993]. After the success of FlashFill, learning string transformation programs has become one of the most popular domains in this area [Gulwani 2011]. Later, partial examples were shown sufficient to learn extraction programs by FlashExtract [Le and Gulwani 2014]. The FlashMeta framework generalizes the deductive backpropagation algorithm behind FlashFill and FlashExtract to a unified framework that significantly reduces the effort required to develop industrial synthesizers [Polozov and Gulwani 2015]. Other successful applications of this technology are *predictive synthesis*, in which no output is given at all, for text splitting [Raza and Gulwani 2017] and *modeless synthesis*, in which the system watches a user and generates its own examples, for suggesting refactoring operations [Miltner et al. 2019].

Neural program synthesis. In earlier approaches to neuro-symbolic program synthesis, neural networks were used to guide [Balog et al. 2019; Ellis et al. 2021] or replace [Devlin et al. 2017; Parisotto et al. 2016] the search over a given DSL. There, the goal is to allow longer programs to be learned over possibly noisy inputs, but the scope of problems that can be solved remains limited to purely syntactic ones. A limited level of semantic capabilities was achieved by leveraging APIs to transform data and using a neural guided search for navigating the large branching factor caused by this integration [Bhupatiraju et al. 2017]. Our integration, on the other hand, extends the DSL with neural operators that are able to learn a required task from few examples, which allows for a fast, enumerative search using deductive backpropagation and is more flexible in the scope of semantic tasks that it performs. Figure 10 compares both ways of integrating neural networks with inductive synthesis.

Language modelling. The ability to learn vector representations of words without supervision [Mikolov et al. 2013] did not only drastically improve the downstream performance of a plethora of natural language processing (NLP) tasks, it also significantly lowered the bar for adding *semantics* to different applications. The challenging task of estimating the semantic similarity between words was reduced to computing a similarity between their vector representation, pre-trained versions of which are readily available to download. Ever since, language modelling has shifted towards training a general model on large amounts of unlabelled data and fine-tuning this model towards a specific task on smaller amounts of labeled data [Devlin et al. 2019]. One way of training such a general model, called *generative* or *autoregressive* pre-training, involves predicting the next token when given a short piece of text. With the ever increasing size of these models, from 117M parameters in the original GPT model [Radford et al. 2018], to 1.5B parameters in GPT-2 [Radford et al. 2019], to 175B parameters in GPT-3 [Brown et al. 2020], the question has arisen of how much knowledge is stored in these parameters [Petroni et al. 2019].

Prompt engineering. It has been shown that the prompt format used to extract information from GPT-3 has a significant influence on the performance [Zhao et al. 2021]. This task of constructing good prompts is called *prompt engineering*. Recent research has focused on determining what constitutes good examples for question answering [Liu et al. 2021a] and how to rewrite prompts to be better for natural language understanding [Liu et al. 2021b].

Semantics in program synthesis. With the increasing availability of large code bases and corpora of web tables, it was only a matter of time until these would be integrated with inductive synthesis. InfoGather [Yakout et al. 2012] and the first DataXFormer [Abedjan et al. 2016] extract and match information contained in web tables for data augmentation and transformation. Later versions of DataXFormer complement web table data with information from knowledge graphs and web forms. The data transformations are limited to lookup in tables, without PBE component, and it therefore requires both input and output to be explicitly present in the tables. Transform-data-by-example (TDE) uses functions from code bases and web forms to allow semantic operations in inductive synthesis [He et al. 2018]. As opposed to our framework, the synthesis algorithm has to be highly tailored towards using these external sources and is limited to string $\rightarrow$ string transformations.

Correctness in program synthesis. Examples are an under-specified format of user intent in program synthesis [Gulwani et al. 2017] and PBE systems are typically not able to guarantee correctness. As opposed to neural networks, which also rarely provide guarantees on their output, synthesized programs can still be validated by users. In this regard, our approach is slightly better than raw neural networks, as the output program conforms to a DSL, but worse than traditional PBE systems, because the program may contain *black-box* neural operators. This may not matter in practise, however, as real-world synthesizers such as FlashFill [Gulwani 2011] and Blue-Pencil [Miltner et al. 2019] do not expose learned programs to the users, as they can be complex and written in a DSL that a user might not be familiar with. Instead, the output is presented to the user for validation. If the number of non-exemplar rows is too large to be validated manually, we can use the technique proposed in [Mayer et al. 2015], where users only need to focus on rows where the outputs of top-rank programs are different.

8 CONCLUDING REMARKS

This paper introduces a novel integration two popular technologies: inductive program synthesis and autoregressive language models with few-shot learning capabilities. We formalize three semantic operators, powered by the language model, that enable tasks involving language understanding and general knowledge, and describe procedures for learning them using deductive backpropagation. These operators can be easily integrated in DSLs for different tasks, such as string transformations and profiling. We show that a combination of syntactic string processing and semantic operators allows the automation of repetitive tasks that involve lookup and natural language understanding from a few examples. In this evaluation, we show that having these operators act as oracles during learning and pruning the set of candidate operators is required to learn these programs quickly. Additionally, we show that the operator semantics and learning can be easily integrated in existing DSLs with a case study on string profiling.

The ideas introduced in this paper suggest several interesting directions for future work. Cheaper language models may be used to improve witness functions and ranking. Specifically, models that allow semantic similarity computations may remove syntactic limitations that stem from clustering. PBE systems are generally sensitive to noise, as they have to learn an exact program from very few examples. In the presence of noise, when a syntactic program is not found, FlashGPT3 will default to semantic operators, which might be resilient to some levels of noise. Finally, we plan to extend this integration to different domains. Most notably, advances in the domains of semantic refactoring and data extraction may quickly lead to commercial adaptation.

ACKNOWLEDGEMENTS

We would like to thank Luc De Raedt for his valuable feedback during the early stages of this research, as well as our anonymous reviewers for their insightful remarks that helped us improve the quality of this work.

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models). This research received funding from the Flemish Government (AI Research Program).

REFERENCES

Ziawasch Abedjan, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. DataXFormer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1134–1145. https://doi.org/10.1109/ICDE.2016.7498319

M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. 2019. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017*.

Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and P. Kohli. 2017. Deep API Programmer: Learning to Program with APIs. *ArXiv* abs/1704.04327 (2017).

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration.* MIT press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

J. Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and P. Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*.

Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *IJCAI*. 1638–1645. https://doi.org/10.24963/ijcai.2017/227

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428287

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

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

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. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. 2017. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1601–1611. https://doi.org/10.18653/v1/P17-1147

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*. 542–553. https://doi.org/10.1145/2666356.2594333

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021a. What Makes Good In-Context Examples for GPT-3? *arXiv preprint arXiv:2101.06804* (2021).

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021b. GPT Understands, Too. *arXiv preprint arXiv:2103.10385* (2021).

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 291–301. https://doi.org/10.1145/2807442.2807459

Microsoft. 2015. Program synthesis from input-output examples (PROSE). https://microsoft.github.io/prose.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13, Vol. 26)*. 3111–3119.

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

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28. https://doi.org/10.1145/3276520

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

Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 2463–2473. https://doi.org/10.18653/v1/D19-1250

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)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

Mohammad Raza and Sumit Gulwani. 2017. Automated data extraction using predictive program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

Adam Roberts, Colin Raffel, and Noam Shazeer. 2020. How Much Knowledge Can You Pack into the Parameters of a Language Model?. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 5418–5426. https://doi.org/10.18653/v1/2020.emnlp-main.437

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

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

Jie Song and Yeye He. 2021. Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes. *Proceedings of the 2021 International Conference on Management of Data* (2021). https://doi.org/10.1145/3448016.3457250

Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2213836.2213848

Tony Z Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. *arXiv preprint arXiv:2102.09690* (2021).