Data Extraction via Semantic Regular Expression Synthesis

Data Extraction via Semantic Regular Expression Synthesis

QIAOCHU CHEN*, University of Texas at Austin, USA
ARKO BANERJEE, University of Texas at Austin, USA
ÇAĞATAY DEMIRALP*, MIT CSAIL, USA
GREG DURRETT, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA

Many data extraction tasks of practical relevance require not only syntactic pattern matching but also semantic reasoning about the content of the underlying text. While regular expressions are very well suited for tasks that require only syntactic pattern matching, they fall short for data extraction tasks that involve both a syntactic and semantic component. To address this issue, we introduce *semantic regexes*, a generalization of regular expressions that facilitates combined syntactic and semantic reasoning about textual data. We also propose a novel learning algorithm that can synthesize semantic regexes from a small number of positive and negative examples. Our proposed learning algorithm uses a combination of neural sketch generation and compositional type-directed synthesis for fast and effective generalization from a small number of examples. We have implemented these ideas in a new tool called Smore and evaluated it on representative data extraction tasks involving several textual datasets. Our evaluation shows that semantic regexes can better support complex data extraction tasks than standard regular expressions and that our learning algorithm significantly outperforms existing tools, including state-of-the-art neural networks and program synthesis tools.

CCS Concepts: • **Software and its engineering** → *Domain specific languages*; **Programming by example**.

Additional Key Words and Phrases: Program Synthesis, Regular Expression

ACM Reference Format:
Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 287 (October 2023), 30 pages. https://doi.org/10.1145/3622863

1 INTRODUCTION

Regular expressions (or *regexes*) are a convenient and versatile mechanism for extracting information from textual data. Because of their wide applicability, many programming languages provide built-in support for regular expressions, allowing developers to perform textual pattern matching. Further, because regular expressions have numerous applications in user-facing applications like spreadsheets, recent years have seen an explosion in the number of new techniques for learning regular expressions from examples and/or natural language [Chen et al. 2020; Lee et al. 2016].

Despite their general practicality, regexes are mainly applicable in settings where the desired data extraction task is *purely syntactic* in nature. For example, regexes are very well-suited to tasks

*Work started and partially completed at Sigma Computing.

Authors' addresses: Qiaochu Chen, University of Texas at Austin, Austin, Texas, USA, qchen@cs.utexas.edu; Arko Banerjee, University of Texas at Austin, Austin, Texas, USA, arko.banerjee@utexas.edu; Çağatay Demiralp, MIT CSAIL, Cambridge, Massachusetts, USA, cagatay@csail.mit.edu; Greg Durrett, University of Texas at Austin, Austin, Texas, USA, gdurrett@cs.utexas.edu; Işıl Dillig, University of Texas at Austin, Austin, Texas, USA, isil@cs.utexas.edu.

2475-1421/2023/10-ART287
https://doi.org/10.1145/3622863

like describing phone numbers and dates because such concepts can be described in terms of a specific syntactic format (e.g., +D-DDD-DDD-DDDD or DD/DD/DDDD). However, many data extraction tasks of practical relevance are *not* so easy to describe using a *purely* syntactic pattern. As a simple example, consider the task of extracting *business* emails from a text file. Any email address must follow a certain syntactic format, but this task also involves a *semantic* component in that it requires determining whether some text in the email describes a business entity. As another example, consider the problem of extracting zip codes that fall within a certain range. In addition to checking whether a string syntactically matches a zip code pattern (DDDDD or DDDDD-DDDD), it requires interpreting part of the string as a number and then performing a semantic range check, which is difficult to do using regexes.

Based on this observation, this paper proposes the concept of a *semantic regex* as a mechanism for combining the strengths of syntactic pattern matching with semantic reasoning. Our proposed semantic regexes generalize standard regular expressions in that they provide a *semantic pattern matching construct* which accepts strings that (a) belong to a category τ (e.g., business, location, person) and (b) satisfy a predicate ϕ when interpreted as an instance of type τ. For example, this construct can be used to match strings that (a) correspond to a City (type τ), and (b) further satisfy some additional criterion, such as being in the United States or in the state of California (predicate ϕ). Under the hood, semantic pattern matching employs large language models like GPT-3 [Brown et al. 2020; Chowdhery et al. 2022] to test membership in some category τ but further allows refining the query result using a logical predicate ϕ. In this sense, one can view our semantic regexes as deciding membership in a refinement type and then combining the matching strings using standard regex operators.

Beyond proposing the notion of semantic regexes, another key contribution of this paper is a new synthesis algorithm for learning semantic regexes from positive and negative examples. The learning problem in this context is more challenging than traditional regex synthesis because semantic regexes are much more expressive than standard regexes. As a result, the hypothesis space in this setting is very large, which has two important consequences:

- First, the semantic regex learning problem cannot be solved using a purely search-based approach due to the sheer size of the search space. In fact, the search space is theoretically not even bounded because our semantic regex language does not restrict the types τ to a pre-defined vocabulary.
- Second, due to the extremely large hypothesis space, there are typically many semantic regexes consistent with a small number of examples. Hence, to find the *intended* semantic regex, our learning algorithm must have a strong inductive bias towards user intent.

The synthesis technique proposed in this paper surmounts these challenges using a novel combination of three key ideas:

(1) **Neural sketch generation:** Our learning algorithm uses a large language model (GPT-3) to generate a sketch of the desired semantic regex. Our key observation is that LLMs are well-suited to this task because they are effective at identifying semantic commonalities between the positive examples and inferring appropriate types to be used within the semantic pattern matching constructs.
(2) **Compositional synthesis:** Our learning algorithm decomposes the synthesis task into multiple simpler sub-problems. Because the holes (i.e., unknowns) in the generated sketches are *typed*, the synthesis technique lends itself to a compositional solution, where we can synthesize each hole largely (though not entirely) independently.
(3) **Type-directed search:** The presence of type information in the sketches makes it possible to fill each hole in a type-directed way. Specifically, we utilize a type system with subtype polymorphism to infer the space of *valid* completions of a hole.

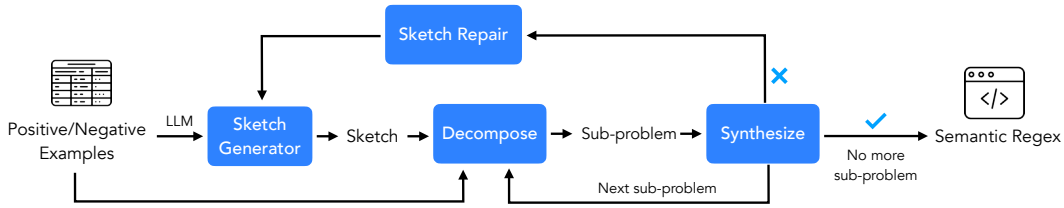

Fig. 1. Schematic overview of our approach.

Object ID	Department	Artist Bio
1	Decorative Art	Heinrich Reinhold, Germany, 1740-1789
2	Contemporary Art	Cindy Sherman, United States, 1954-present
3	Medieval Art	Sandro Botticelli, Italy, 1470-1561
4	Modern Art	Max Ernst, Germany, 1891-1976
5	Medieval Art	Niclaus Gerhaert von Leyden, North Netherlands, 1462-1473
...	...	...

Fig. 2. Dataset about pieces of art exhibited in a museum.

Figure 1 shows the workflow of our proposed learning approach, which first utilizes the provided examples to generate a semantic regex *sketch* using GPT-3. In the next step, our approach searches for completions of the sketch by (a) decomposing the overall problem into several subproblems and (b) using type-directed synthesis to solve each subproblem. If the sketch has a valid completion, the resulting semantic regex is returned to the user. Otherwise, our approach analyzes the root cause of failure and uses this information to query the language model for a more accurate sketch.

We have implemented the proposed technique in a tool called Smore and evaluated it on information extraction tasks involving several different datasets. Our evaluation shows that these data extraction tasks can be successfully automated using our proposed semantic regexes and that our learning algorithm is quite effective for automating the desired data extraction task. In particular, our approach achieves an average F_1 score of 0.87 on the test data, while prior data extraction techniques achieve a maximum F_1 score of 0.65.

To summarize, this paper makes the following contributions:

- We propose *semantic regular expressions* to combine the flexibility of syntactic pattern matching with semantic queries involving types and logical predicates.
- We describe a new learning technique for synthesizing semantic regexes from positive and negative examples. Our approach combines the power of large language models with type-directed synthesis for effective automation of data extraction tasks.
- We evaluate our tool, Smore, on representative data extraction tasks and show that semantic regexes are useful for these tasks and that our learning approach outperforms other data extraction techniques in terms of average F_1 score.

2 OVERVIEW

In this section, we illustrate our technique using the motivating example shown in Figure 2, which contains information about artworks exhibited at a museum. Given this dataset, suppose that a user wants to extract all European artists who were born before the 20th century and whose name contains Thomas. This data extraction task is challenging because it requires both syntactic and semantic reasoning:

- **Syntax:** In order to retrieve the desired information from this dataset, we first need to perform pattern matching over the syntax of the "Artist Bio" column. In particular, because this column contains information of the form "Name, Country, Birth Year - Death Year", we first need to *syntactically* parse the input string into its four constituent fields and check whether the first field (corresponding to the artist name) contains "Thomas".
- **Semantics:** After performing syntactic pattern matching, we then need to perform semantic reasoning about the contents of each row to understand whether (a) the first field describes a name, (b) the artist's nationality is European and (c) they were born before the 20th century.

2.1 Semantic Regexes

Our proposed *semantic regex* concept is a natural fit for the data extraction task illustrated in this example. Semantic regexes combine the convenience of regexes for syntactic pattern matching with the power of semantic reasoning about data types. In addition to supporting the standard regex operators (concatenation, disjunction, Kleene star), semantic regexes provide the following *semantic* pattern matching construct, written using a refinement-type-like notation:

$$\{v : \tau \mid \phi\}$$

This construct matches any string that is semantically of type τ and that further satisfies the (optional) logical qualifier ϕ. For instance, going back to our example, recall that we need to pattern match strings that correspond to a European country. This can be expressed using the semantic regex $\{v : \mathsf{Country} \mid v \in \mathsf{Europe}\}$, which, for example, matches the strings "France", "Britain" and "North Netherlands", but fails on the strings "United States", "Korea" etc. Similarly, we can express the desired constraint on the artists' birth year using the following semantic regex:

$$\{v : \mathsf{Year} \mid v < 1900\}$$

which matches strings that (a) correspond to a year and (b) whose value is less than or equal to 1899. Putting all of this together, our desired data extraction task can be accomplished using the following overall semantic regex:

$$\begin{aligned} & r_1 \cdot \text{“, ”} \cdot r_2 \cdot \text{“, ”} \cdot r_3 \cdot \text{“ – ”} \cdot r_4 \\ \texttt{where}\ & r_1 = \{v : \mathsf{Name}\} \cap \mathtt{Contain}(\text{“Thomas”}) \\ & r_2 = \{v : \mathsf{Country} \mid v \in \mathsf{Europe}\} \\ & r_3 = \{v : \mathsf{Year} \mid v < 1900\} \\ & r_4 = \{v : \mathsf{Year}\} \end{aligned}$$

In other words, this semantic regex matches all strings of the form "X, Y, Z-W" where X is a name containing Thomas, Y is a European country, Z is a year before 1900, and W is any year.

2.2 Synthesizing Semantic Regexes

While semantic regexes provide a useful mechanism for information extraction, they can nonetheless be non-trivial for end-users to construct. Motivated by this problem, another key contribution of this paper is a new technique for synthesizing semantic regexes from a small number of positive and negative examples. We now illustrate how our technique can be used to automate the data extraction task for our running example. Suppose that the user describes the target data extraction task using the following positive and negative examples:

Positive Examples	Negative Examples
John Thomas Young Gilroy, Britain, 1898-1985	Alma Thomas, United States, 1891-1978
Thomas Hudson, Britain, 1701-1779	Sandro Botticelli, Italy, 1470-1561
Thomas Couture, France, 1815-1879	Thomas Nölle, Germany, 1948-2020

Here, the positive examples correspond to the artist biographies that should be extracted, while the negative examples are those that should be ignored. In particular, the first negative example does not conform to the "European country" restriction; the second negative example does not contain "Thomas" in the artist's name; and the third one fails the criteria "born before the 20th century". We will now describe how our approach synthesizes the target semantic regex given only these examples.

At the heart of our learning approach lies the notion of a *typed sketch*, which captures the general syntactic structure of the target semantic regex. In addition, the holes (i.e., unknowns) in the sketch are annotated with types capturing commonalities in the positive examples. Returning to our running example, our synthesis approach generates an initial candidate sketch by querying a large language model (GPT-3) with user-provided positive examples. Suppose that GPT-3 returns the following sketch:

$$\{\square : \mathsf{Name}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Country}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Year}\}$$

Here, the symbol $\{\square : \}$ denotes an unknown expression, and the notation $\{\square : \mathsf{t}\}$ indicates that any string matched by $\{\square : \}$ should be a subtype of t.

Starting with the GPT-3-synthesized sketch, our method decomposes the synthesis problem into multiple sub-problems, one for each hole in the sketch, and performs a type-directed search to complete each hole. For this example, our synthesis method infers the following positive examples for each hole:

$\{\square : \mathsf{Name}\}$	$\{\square : \mathsf{Country}\}$	$\{\square : \mathsf{Year}\}$
John Thomas Young Gilroy	Britain	1898-1985
Thomas Hudson	Britain	1701-1779
Thomas Couture	France	1815-1879

Note that it is not possible to propagate negative examples for individual holes, as it suffices for the synthesized regex for *one* hole to reject its corresponding string, but we do not a priori know which one. In particular, for this example, it would *not* be accurate to deduce that "Alma Thomas", "Sandro Botticelli", and "Thomas Nölle" as negative examples for the first hole.

Given this decomposition, our approach tries to synthesize a regex r_i for each hole $\{\square : \tau_i\}_i$ such that (a) the type of r_i is a subtype of τ_i and (b) r_i matches all of its corresponding positive examples. For this example, our synthesis algorithm can immediately deduce that the sketch is incorrect since no subtype of Year can match the corresponding positive examples for the third hole.

To repair the sketch, our learning algorithm localizes parts of the sketch for which synthesis failed (in this case, Year) and synthesizes a different sketch for the failing part. In the next iteration, suppose that we consider the following correct sketch:

$$\{\square : \mathsf{Name}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Country}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Year}\} \cdot \text{“ – ”} \cdot \{\square : \mathsf{Year}\}$$

Our synthesis algorithm tries to independently find the completion of each hole with the appropriate type and satisfy the corresponding decomposed positive examples. As before, the positive examples are used to prune the search space: for example, since the second hole must match the strings "Britain" and "France", the synthesizer can rule out completions such as $\{v : \mathsf{Country} \mid v \in \mathsf{Asia}\}$ and $\{v : \mathsf{Country} \mid v \in \mathsf{Asia} \ \wedge \ldots\}$. Similarly, type information in the sketch is critical, enabling the synthesizer to avoid enumerating useless sub-programs. For instance, when synthesizing the last hole in the sketch, the synthesizer would not enumerate programs such as $\{v : \mathsf{Month} \mid \ldots\} \cup \{v : \mathsf{Date} \mid \ldots\}$, since this regex can match strings that are not of type Year. It would, however, consider regexes of the form $\{v : \mathsf{Year} \mid v \leq \ldots\}$, as the strings that are matched by this regex would be a subtype of year. After independently synthesizing each hole, the algorithm checks whether the resulting regex r rejects all negative examples and, if so, returns r as a solution. Otherwise, it generates a different regex by looking for a different completion for at least one of the holes.

$$
\begin{array}{rl}
\rho ::= & \lambda s.\, \texttt{match}(s, r) \\
r ::= & c \mid cc \mid \emptyset \\
& \mid \{v : f(\tau_q)\} \mid \{v : f(\tau_b) \mid \phi\} \\
& \mid \neg r \mid r? \mid r* \mid r+ \mid r\{k_1\} \mid r\{k_1, k_2\} \\
& \mid r \cdot r \mid r \cup r \mid r \cap r \\
f ::= & \texttt{id} \mid \texttt{toUpper} \mid \texttt{toLower} \mid \texttt{abbreviate}[c] \\
\phi ::= & \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \\
& \mid t \oplus_{\tau_b} t \;\; \texttt{where} \oplus \in \{\leq, \geq, =, \in\} \\
t ::= & v \mid v.a \mid c \mid n \\
\tau_b ::= & \text{Person} \mid \text{Organization} \mid \text{Product} \mid \text{Event} \mid \text{Work of Art} \\
& \mid \text{Number} \mid \text{Integer} \mid \text{Float} \\
& \mid \text{Date} \mid \text{Year} \mid \text{Month} \mid \text{Day} \\
& \mid \text{Time} \mid \text{Hour} \mid \text{Minute} \mid \text{Second} \\
& \mid \text{Place} \mid \text{Location} \mid \text{Nationality} \mid \text{Country} \mid \text{City}
\end{array}
$$

Fig. 3. Semantic string matching language. c is a constant string, cc is a character class (e.g. letters). τ_b is a built-in base type, and τ_q is an arbitrary base type in our type system. Also, $k \in \mathbb{Z}$, $n \in \mathbb{R}$, and $a \in \texttt{Attributes}$, where `Attribute` is type-dependent.

3 SEMANTIC REGULAR EXPRESSIONS

In this section, we describe the syntax and semantics of our proposed semantic regular expression language. At a high level, semantic regexes combine standard regular expression operators with pre-trained neural networks that identify semantic types and provide knowledge about the world.

DSL Syntax. The syntax of our semantic string matching language is presented in Figure 3. A semantic regex ρ takes as input a string s and returns a boolean indicating whether there is a match. Semantic regexes include all the standard regular expression constructs, including constant strings c, character classes like letters and numbers (denoted cc), concatenation ($\cdot$), complement ($\neg$), union ($\cup$), intersection ($\cap$), and Kleene star ($*$). Additionally, the notation $r\{k_1\}$ denotes repetition of r k_1 times and $r\{k_1, k_2\}$ denotes r repeated between k_1 to k_2 times. As standard, $r?$ indicates an optional occurrence of r, and $r+$ denotes one or more occurrences of r.

In addition to these standard regex constructs, Figure 3 includes two *semantic pattern matching constructs*, denoted as $\{v : f(\tau_q)\}$ and $\{v : f(\tau_b) \mid \phi\}$, where f is an (optional) built-in function, τ_b is a built-in type (Integer, Month, etc) and τ_q is an *arbitrary* (user-defined) type. Note that the DSL does not place any restrictions on τ_q, so the user can provide any arbitrary string to define their own type. However, we only allow a logical qualifier ϕ to be used for built-in types.

In the most basic form, the construct $\{v : \tau\}$ matches strings that are semantically of type τ, where τ can either be a built-in or user-defined type. For example, $\{v : \text{Place}\}$ matches any string that corresponds to a geographical location. The optional function f used in this construct allows refining the query result by performing additional semantic-preserving string processing. For example, $\{v : \texttt{toUpper}(\text{Place})\}$ matches any string that corresponds to a location name in upper case letters (e.g., "NEW YORK"). More generally, $\{v : f(\tau)\}$ matches a string s if s is equal to $f(s')$ where s' is a string of type τ. As another example, $\{v : \texttt{abbreviate}[.](\text{Place})\}$ matches the strings "N.Y.", "S.F." etc. because the function $\texttt{abbreviate}[c]$ abbreviates a string through initialism, using the character c as a separator.

When performing semantic pattern matching using built-in types τ_b, one can additionally use a logical qualifier ϕ. In particular, $\{v : \tau_b \mid \phi\}$ matches those strings that are of type τ_b and additionally satisfy predicate ϕ. To check whether a string s satisfies ϕ, s is first parsed as an instance o of type τ_b and then checked for conformance against ϕ. Note that these semantics justify why logical

$$
\begin{aligned}
[\![\lambda s.\, \texttt{match}(s, r)]\!]s &= s \in [\![r]\!] \\
[\![c]\!] &= \{c\} \\
[\![\neg r]\!] &= \{s \mid s \notin [\![r]\!]\} \\
[\![r^0]\!] &= \{\epsilon\} \\
[\![r^i]\!] &= \{s_1 \cdot s_2 \mid s_1 \in [\![r^{i-1}]\!], s_2 \in [\![r]\!]\} \\
[\![r*]\!] &= \bigcup_{n \in \{0..\infty\}} [\![r^n]\!] \\
[\![r_1 \cdot r_2]\!] &= \{s_1 \cdot s_2 \mid s_1 \in [\![r_1]\!], s_2 \in [\![r_2]\!]\} \\
[\![r_1 \cup r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r_1 \cap r_2]\!] &= [\![r_1]\!] \cap [\![r_2]\!] \\
[\![\{v : f(\tau_q)\}]\!] &= \{[\![f]\!]s \mid \texttt{SemanticType}(s) = \tau_q\} \\
[\![\{v : f(\tau_b) \mid \phi\}]\!] &= \{[\![f]\!]s \mid \texttt{SemanticType}(s) = \tau_b \wedge \texttt{Cast<}\tau_b\texttt{>}(s) = o \wedge \phi(o)\}
\end{aligned}
$$

Fig. 4. Semantics of matching part of the DSL. Here, SemanticType is an oracle that determines the semantic type of string s, Cast<τ> casts string s to object o of type τ.

qualifiers are only allowed with built-in types: because we need to parse the string as an instance of τ_b, there must be some built-in mechanism for deserializing the string, which only makes sense for pre-defined types. As an example, the semantic regex $\{v : \text{Float} \mid v < 0.1\}$ matches strings that can be interpreted as a floating point number whose value is less than 0.1 (e.g., 0.0051). As another example, $\{v : \texttt{toUpper}(\text{City}) \mid v \in \text{Europe}\}$ matches strings, such as "ROME" that (a) correspond to European cities and (b) are in upper case letters.

DSL Semantics. Figure 4 presents the formal semantics of our DSL for semantic string matching, where $[\![r]\!]$ denotes the set of all strings that r matches.[1] Observe that the semantics of the DSL is parametrized by a helper function called SemanticType, which is implemented by a pre-trained neural network and which is used to check whether the type of a string s is τ. Hence, the construct $\{v : f(\tau_q)\}$ matches all strings s such that (a) $s = f(s')$ for some string s', and (b) where $\texttt{SemanticType}(s') = \tau_q$. Similarly, $\{v : f(\tau_b) \mid \phi\}$ matches all strings s such that (a) $s = f(s')$ for some string s', (b) s' is an instance of built-in type τ_b, and (c) when s' is parsed into an object o of type τ_b, o satisfies predicate ϕ.

Example 3.1. The semantic regex $\{v : \text{Date} \mid v.\text{month} = 5\}$ matches all strings that represent dates in May. In particular, any string matching a Date is first parsed into a datetime object and its month field is checked for being equal to 5. Examples of strings matched by this regex include "May 2023" and "2023-05-01".

4 OVERVIEW OF THE TYPE SYSTEM

While our semantic regex DSL is not *explicitly* typed, our approach utilizes a type system to facilitate effective synthesis. In this section, we give an overview of the type system.

4.1 Type Syntax

The syntax of our type system is shown in Figure 5, where Any corresponds to the top element in the type system and CharSeq indicates any string without semantic meaning, such as "1a2b3c", ",.3d,." etc. The type $\texttt{Semantic}(\tau_s)$ indicates strings that can interpreted as instance of τ_s (e.g., Date). In addition, the type $\texttt{Optional}(\tau)$ includes both ϵ (empty string) as well as any string of type τ. Semantic types τ_s include both built-in types τ_b (e.g., Integer, Float, Date) as well as user-defined types τ_q. Hence, the type syntax is *not* fixed a priori and is parametrized over any user-defined types that occur in the program.

[1]Semantics of functions are provided in the appendix.

$$
\begin{array}{rcl}
\tau & := & \texttt{Any} \mid \texttt{Optional}(\tau') \mid \tau' \\
\tau' & := & \texttt{Semantic}(\tau_s) \mid \texttt{CharSeq} \\
\tau_s & := & \texttt{Person} \mid \texttt{Organization} \mid \texttt{Product} \mid \texttt{Event} \mid \texttt{Work of Art} \\
 & & \mid \texttt{Number} \mid \texttt{Integer} \mid \texttt{Float} \\
 & & \mid \texttt{Date} \mid \texttt{Year} \mid \texttt{Month} \mid \texttt{Day} \\
 & & \mid \texttt{Time} \mid \texttt{Hour} \mid \texttt{Minute} \mid \texttt{Second} \\
 & & \mid \texttt{Place} \mid \texttt{Location} \mid \texttt{Nationality} \mid \texttt{Country} \mid \texttt{City}
\end{array}
$$

Fig. 5. Type syntax.

$$
\begin{array}{lll}
\vdash \texttt{CharSeq} <: \texttt{Any} & \vdash \texttt{Semantic}(\tau_s) <: \texttt{Any} & \vdash \texttt{Optional}(\tau) <: \texttt{Any} \\
\vdash \texttt{Year} <: \texttt{Date} & \vdash \texttt{Month} <: \texttt{Date} & \vdash \texttt{Day} <: \texttt{Date} \\
\vdash \texttt{Hour} <: \texttt{Time} & \vdash \texttt{Minute} <: \texttt{Time} & \vdash \texttt{Second} <: \texttt{Time} \\
\vdash \texttt{Country} <: \texttt{Place} & \vdash \texttt{City} <: \texttt{Place} & \\
\vdash \texttt{Institution} <: \texttt{Organization} & \vdash \texttt{Company} <: \texttt{Organization} &
\end{array}
$$

$$
\textsc{Trans}\ \frac{\vdash \tau'' <: \tau' \quad \vdash \tau' <: \tau}{\vdash \tau'' <: \tau} \qquad \textsc{Semantic}\ \frac{\vdash \tau' <: \tau}{\vdash \texttt{Semantic}(\tau') <: \texttt{Semantic}(\tau)}
$$

$$
\textsc{Optional-Width}\ \frac{}{\vdash \tau <: \texttt{Optional}(\tau)}
$$

$$
\textsc{Optional-Congruence}\ \frac{\vdash \tau' <: \tau}{\vdash \texttt{Optional}(\tau') <: \texttt{Optional}(\tau)} \qquad \textsc{User-defined}\ \frac{\gamma(\tau') \subseteq \gamma(\tau)}{\vdash \tau' <: \tau}
$$

Fig. 6. Subtyping relations. $\gamma(\tau)$ is the concretization function denoting the set of objects represented by τ.

4.2 Subtyping

Our type system supports subtype polymorphism because there is a natural subtyping relation between many entities of interest. We formalize the subtyping relation in Figure 6 using the standard judgment $\vdash \tau_1 <: \tau_2$, indicating that τ_1 is a subtype of τ_2. In Figure 6, the first three rules are straightforward and establish Any as the top element of the type system. The following rules (until Trans) show the subtyping relation involving built-in semantic types. For example, according to these rules, Year, Month, and Day are all subtypes of the more generic Date type. The Trans rule states the transitivity of the subtyping relation and the Semantic rule lifts the subtyping relation to $\texttt{Semantic}(\tau)$. The last two rules for Optional are also standard: Optional-Width states that any type τ is a subtype of $\texttt{Optional}(\tau)$ and the last rule lifts the subtyping relation to optional types. Finally, the last rule handles subtyping between user-defined types. If the set of objects represented by τ_1 is a subset of those represented by τ_2, we have $\tau_1 <: \tau_2$. In practice, we perform this check by querying a semantic ontology (specifically, DBPedia [Bizer et al. 2009] in our implementation).

4.3 Typing Rules

We present the typing rules for assigning types to DSL terms in Figure 7. These rules derive judgments of the form $\vdash t : \tau$ indicating that term t has type τ. Note that Figure 7 only shows a representative subset of the typing judgments; the full set is presented in the Appendix under supplementary materials.

Constant and characters. The first four rules show how to assign types to string constants and character classes. For constants, we determine their type by querying a semantic oracle (GPT-3 in

$$\textsc{Const-Semantic}\ \frac{\mathsf{SemanticType}(c) = \tau \quad \tau \neq \mathsf{CharSeq}}{\vdash c : \mathsf{Semantic}(\tau)} \qquad \textsc{Const-CharSeq}\ \frac{\mathsf{SemanticType}(c) = \mathsf{CharSeq}}{\vdash c : \mathsf{CharSeq}}$$

$$\textsc{CC}\ \frac{cc \neq <\mathsf{Num}>}{\vdash cc : \mathsf{CharSeq}} \qquad \textsc{CC-Num}\ \frac{cc = <\mathsf{Num}>}{\vdash cc : \mathsf{Semantic}(\mathsf{Number})}$$

$$\textsc{matchSem}\ \frac{}{\vdash \{v : f(\tau_b) \mid \phi\} : \mathsf{Semantic}(\tau_b)} \qquad \textsc{matchSem}\ \frac{}{\vdash \{v : f(\tau_q)\} : \mathsf{Semantic}(\tau_q)}$$

$$\textsc{Lifting}\ \frac{\vdash r : \tau \quad \tau <: \tau'}{\vdash r : \tau'} \qquad \textsc{Optional}\ \frac{\vdash r : \tau}{\vdash r? : \mathsf{Optional}(\tau)}$$

$$\textsc{Union}\ \frac{\vdash r_1 : \tau_1 \quad \vdash r_2 : \tau_2}{\vdash r_1 \cup r_2 : \tau_1 \vee \tau_2} \qquad \textsc{And}\ \frac{\vdash r_1 : \tau_1 \quad \vdash r_2 : \tau_2}{\vdash r_1 \cap r_2 : \tau_1 \wedge \tau_2}$$

$$\textsc{Not}\ \frac{\vdash r : \tau}{\vdash \neg r : \mathsf{Any}} \qquad \textsc{Concat}\ \frac{\vdash r_1 : \tau_1 \quad \vdash r_2 : \tau_2}{\vdash r_1 \cdot r_2 : \mathsf{Any}}$$

Fig. 7. Typing rules.

our implementation) and assign CharSeq if the oracle does not return a semantic type.[2] Character classes only have semantic meaning for numbers, so we assign the Semantic(Number) type if the character is a number, and CharSeq otherwise.

Semantic matching. The MatchSem rules present the typing rules for the semantic matching construct. The type of the expression is identical to the type specified as part of the program syntax.

Union and intersection. The typing rules for union and intersection presented in the Union and And rules, respectively. These rules utilize the $\vee$ and $\wedge$ operators, which are defined in Figure 8. At a high level, the meet and join of two types are determined as the least upper bound ($\sqcup$) and the greatest lower bound ($\sqcap$), respectively, in the corresponding type lattice. However, there is a special case for the CharSeq type: Intuitively, taking the intersection of a semantic type τ and CharSeq further refines the objects of type τ by placing an *additional* syntactic restriction; hence, $\mathsf{Semantic}(\tau) \wedge \mathsf{CharSeq}$ is defined as $\mathsf{Semantic}(\tau)$. In contrast, the join of $\mathsf{Semantic}(\tau)$ and CharSeq is the top element Any, as expected.

Not and concatenation. The Not and Concat are two cases where specific types cannot be inferred. Even though the type of their arguments is known, the resulting type cannot be determined, resulting in an output type of Any.

5 LEARNING SEMANTIC REGEXES FROM EXAMPLES

In this section, we describe our synthesis algorithm for solving the semantic string matching problem from examples. Our method involves two main steps: generating a *typed sketch* from the positive examples and completing the sketch using an enumerative search-based synthesizer. If sketch completion fails, our method refines the sketch and performs synthesis using the new sketch. In the rest of this section, we first provide some preliminary information, then present our top-level learning algorithm, and then describe each of its key components.

[2]The semantic oracle returns CharSeq if the string has no semantic meaning.

$$
\begin{aligned}
\tau \wedge \texttt{Any} &= \tau \\
\tau_1 \wedge \texttt{Optional}(\tau_2) &= \tau_1 \wedge \tau_2 \\
\texttt{Optional}(\tau_1) \wedge \texttt{Optional}(\tau_2) &= \texttt{Optional}(\tau_1 \wedge \tau_2) \\
\texttt{Semantic}(\tau_1) \wedge \texttt{Semantic}(\tau_2) &= \texttt{Semantic}(\tau_1 \sqcap \tau_2) \\
\texttt{Semantic}(\tau) \wedge \texttt{CharSeq} &= \texttt{Semantic}(\tau)
\end{aligned}
$$

$$
\begin{aligned}
\tau \vee \texttt{Any} &= \texttt{Any} \\
\tau_1 \vee \texttt{Optional}(\tau_2) &= \texttt{Optional}(\tau_1 \vee \tau_2) \\
\texttt{Optional}(\tau_1) \vee \texttt{Optional}(\tau_2) &= \texttt{Optional}(\tau_1 \vee \tau_2) \\
\texttt{Semantic}(\tau_1) \vee \texttt{Semantic}(\tau_2) &= \texttt{Semantic}(\tau_1 \sqcup \tau_2) \\
\texttt{Semantic}(\tau) \vee \texttt{CharSeq} &= \texttt{Any}
\end{aligned}
$$

Fig. 8. Type intersection and union.

$$
\begin{array}{lll}
S := & r & \text{(regex)} \\
 & \mid \mathsf{f}(\overline{S}) & \text{(operator in the language)} \\
 & \mid \{\square : \tau\} & \text{(typed hole)}
\end{array}
\qquad
\begin{aligned}
[\![r]\!] &= \{r\} \\
[\![\mathsf{f}(\overline{S})]\!] &= \{\mathsf{f}(\overline{r} \mid \forall_{i \in |\overline{r}|} r_i \in [\![S_i]\!])\} \\
[\![\{\square : \tau\}]\!] &= \{r \mid \vdash r : \tau\}
\end{aligned}
$$

Fig. 9. Sketch syntax and its semantics. Here f refers to any construct in the DSL defined in Figure 3.

5.1 Sketch Language

Our learning algorithm crucially relies on the notion of a *typed sketch* whose syntax is shown in Figure 9. At a high level, the sketch language extends our semantic regex DSL by allowing a "typed hole" (denoted $\{\square : \tau\}$) which represents an arbitrary expression of type τ. Given a sketch S, we use the notation $[\![S]\!]$ to denote the set of all semantic regexes that can be obtained by completing holes in S by valid expressions of the corresponding type. Figure 9 also defines sketch semantics in terms of the space of all programs they represent.

Example 5.1. Consider the sketch $\{\square : \text{Organization}\} \cdot$ "*.com*", which represents the space of semantic regexes that match strings consisting of an organization name followed by the string constant ".com". Possible completions of this sketch include, but are not limited, to the following semantic regexes: (1) $\{v : \text{Company}\} \cdot$ "*.com*", (2) $\{v : \text{Institution}\} \cdot$ "*.com*", and (3) $(\{v : \text{Institution}\} \cup \{v : \text{Company}\}) \cdot$ "*.com*".

5.2 Top-level algorithm

Our top-level algorithm is outlined in Figure 10. Given a set of positive examples $\mathcal{E}^+$ and a set of negative examples $\mathcal{E}^-$, Synthesize returns a semantic regex that accepts all positive examples and rejects all negative examples. At a high level, the algorithm repeatedly generates a new sketch using a large language model, then attempts to find a valid instantiation of that sketch, and continues this process until it finds a regex that is consistent with all user-provided examples. Intuitively, each candidate sketch serves as a possible generalization of the positive examples, and the goal of the synthesizer is to determine whether that sketch is a suitable generalization.

In more detail, the Synthesize procedure first calls GetNextSketch, which queries GPT-3 to produce a sketch S that is likely to satisfy the positive examples. Then, for a given sketch S, GetNextDecomp infers a *decomposition* Ψ, which is a mapping from each hole in S to a set of positive examples for that hole. Then, for a given decomposition Ψ, the algorithm calls SynthesizeFromDecomp to perform compositional synthesis based on the inferred specification Ψ.

If the call to SynthesizeFromDecomp returns a non-empty mapping M, which maps each hole in S to a concrete regex r, we find a solution that is consistent with the specification and

```
1:  procedure Synthesize(E+, E−)
    input: A set of positive E+ and negative examples E−.
    output: A program that is consistent with the examples.
2:      S_f ← ⊥;
3:      while HasMoreSketch(E+) do
4:          S ← GetNextSketch(E+, S_f);
5:          while HasDecomp(S, E+) do
6:              Ψ ← GetNextDecomp(S, E+);
7:              M ← SynthesizeFromDecomp(S, Ψ, E−);
8:              if M ≠ ⊥ then return S[M];
9:          S_f ← S;
10:     return ⊥;
```

Fig. 10. Top-level synthesis algorithm. Here, $S[M]$ means replacing each hole $h \in S$ with $M[h]$.

returns the synthesized regex by replacing the holes in S with the corresponding solution in M. Otherwise, if the call to SynthesizeFromDecomp yields $\perp$, there are two possibilities: Either the decomposition Ψ is incorrect (recall from Section 2 that there is ambiguity in how to assign positive examples to holes), or the sketch S itself is incorrect. In the former case, the algorithm considers a different decomposition, which maps at least one of the holes in the sketch to a different set of examples. If the algorithm exhausts all possible decompositions, this means that the sketch must be incorrect and the algorithm repairs the current sketch by performing fault localization and querying GPT-3 to produce a different generalization of the positive examples. This process continues until the algorithm finds a globally consistent regex with all (positive *and* negative) examples or runs out of possible sketches. In the following discussion, we explain each of the three components (decomposition, type-directed synthesis, and sketch repair) in more detail.

5.3 Decomposing the Specification

To perform compositional synthesis, our learning algorithm decomposes the global specification into a *set* of specifications, one for each hole in the sketch. In this section, we describe the GetNextDecomp procedure for specification decomposition using the inference rules in Figure 11, which derive judgments of the following shape:

$$\mathcal{E}^+ \vdash S \rightsquigarrow \Psi$$

The meaning of this judgment is that, given positive examples $\mathcal{E}^+$, Ψ is a *possible* decomposition that maps each hole in the sketch to its corresponding positive examples. As mentioned earlier, the decomposition is, in general, *not* unique, so there can be multiple decompositions $\Psi_1, \ldots, \Psi_n$ for a given sketch S.

We now explain the decomposition rules from Figure 11 in more detail. The first rule, labeled Sketch-Match, considers a program sketch with top-level operator f (e.g., concatenation or intersection) and sub-sketches $S_1, \ldots, S_n$. To infer a specification for each hole in S, we first generate a regex $r^\star$ that over-approximates S (via the call to `OverApprox`). Intuitively, `OverApprox` generates a regex $r^\star$ such that for *any* $r \in [\![S]\!]$, $r^\star$ accepts every string that is accepted by r. Because our over-approximation approach is exactly the same as used in prior work [Chen et al. 2020; Lee et al. 2016], we do not formally present it, but the basic idea is to replace each hole that appears under an even (resp. odd) number of negation symbols by the regex $.*$ (resp. $\emptyset$). This method guarantees that the resulting regex $r^\star$ will accept every string that is accepted by any instantiation of S.

$$\textsc{Sketch-Match}\ \frac{S = f(S_1,\ldots,S_n) \quad r^\star = \texttt{OverApprox}(S) \quad (\mathcal{E}_1^+,\ldots,\mathcal{E}_n^+) \in \texttt{Match}(r^\star,\mathcal{E}^+) \quad \mathcal{E}_i^+ \vdash S_i \rightsquigarrow \Psi_i \quad i \in \{1,\ldots,n\}}{\mathcal{E}^+ \vdash S \rightsquigarrow \texttt{Merge}(\Psi_1,\ldots,\Psi_n)}$$

$$\textsc{Sketch-NoPosMatch}\ \frac{S = f(S_1,\ldots,S_n) \quad r^\star = \texttt{OverApprox}(S) \quad \texttt{Match}(r^\star,\mathcal{E}^+) \equiv \emptyset}{\mathcal{E}^+ \vdash S \rightsquigarrow \bot}$$

$$\textsc{Concrete-Feasible}\ \frac{\texttt{Match}(r,\mathcal{E}^+) \neq \emptyset}{\mathcal{E}^+ \vdash r \rightsquigarrow \emptyset} \qquad \textsc{Concrete-Infeasible}\ \frac{\texttt{Match}(r,\mathcal{E}^+) \equiv \emptyset}{\mathcal{E}^+ \vdash r \rightsquigarrow \bot}$$

$$\textsc{Hole-Feasible}\ \frac{\forall e \in \mathcal{E}^+.\ \ \texttt{SemanticType}(e) <: \tau}{\mathcal{E}^+ \vdash \{\square : \tau\} \rightsquigarrow [\{\square : \tau\} \mapsto \mathcal{E}^+]} \qquad \textsc{Hole-Infeasible}\ \frac{\exists e \in \mathcal{E}^+.\ \ \texttt{SemanticType}(e) \not<: \tau}{\mathcal{E}^+ \vdash \{\square : \tau\} \rightsquigarrow \bot}$$

Fig. 11. Procedure for $\textsc{GetNextDecomp}(S,\mathcal{E}^+)$. $\texttt{OverApprox}(S)$ returns a concrete regex that over-approximates S. Merge returns $\bot$ if one of its argument is $\bot$, otherwise it disjointly unions all its arguments.

Furthermore, note that $r^\star$ is a standard regex without any semantic pattern matching constructs, as all holes have been replaced by either the universal or the empty set.

Next, once we generate the over-approximation $r^\star$, we infer positive examples for each sub-sketch $S_1,\ldots,S_n$ used in S. To do so, for each positive example e, we use a standard regex matching tool to find a parse of e into the format $f(S_1,\ldots,S_n)$ with corresponding sub-strings e_i for each sub-sketch S_i. After propagating each example e_i to nested sketch S_i and recursively applying the inference rules, we obtain the decomposed specifications $\Psi_1,\ldots,\Psi_n$ for each of the sub-sketches in S. These mappings are finally combined via the call to the Merge function, defined as follows:

$$\texttt{Merge}(\Psi_1,\ldots,\Psi_n) = \begin{cases} \bot & \text{if } \exists i \in [1,n].\ \Psi_i = \bot \\ \uplus_{i=1}^{n} \Psi_i & \text{otherwise} \end{cases}$$

where the notation $\uplus$ indicates disjoint union.

The next rule, labeled Sketch-NoMatch, corresponds to an infeasible sketch or decomposition. Because every string accepted by $r \in [\![S]\!]$ must also be accepted by the over-approximation $r^\star$, the algorithm yields $\bot$ to indicate a failure when $r^\star$ doesn't match at least one of the positive examples.

The remaining rules correspond to the base cases of the recursive decomposition algorithm. Specifically, the rules prefixed with Concrete consider the case where the sketch is a concrete regex r without a hole. Specifically, we check the feasibility of r by testing whether it matches all of the positive examples. If so, the sketch is feasible, and the algorithm returns the empty mapping $\emptyset$. Otherwise (the Concrete-Infeasible case), the algorithm returns $\bot$ to indicate failure.

The final two rules correspond to base cases for a hole and utilize the fact that sketches are typed. In particular, given a hole of type τ, if there exists a positive example $e \in \mathcal{E}^+$ whose type is not τ, this indicates a conflict and the algorithm returns $\bot$ in the Hole-Infeasible rule. Otherwise, in the Hole-Feasible rule, the constructed specification maps this hole to the input positive examples $\mathcal{E}^+$.

Example 5.2. Consider the positive examples from Section 2 and the following sketch:

$$\{\square : \mathsf{Name}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Country}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Year}\}_1 \cdot \text{“ – ”} \cdot \{\square : \mathsf{Year}\}_2$$

The over-approximation for this sketch is the following regex:

$$.* \cdot \text{“, ”} \cdot .* \cdot \text{“, ”} \cdot .* \cdot \text{“ – ”} \cdot .*$$

1: **procedure** SynthesizeFromDecomp($S, \Psi, \mathcal{E}^-$)
input: A sketch S, a specification Ψ, a set of negative examples $\mathcal{E}^-$.
output: A sketch completion consistent with all examples.
2: $h \leftarrow$ `ChooseHole`(S)
3: **while** `True` **do**
4: $r \leftarrow$ GetNextCompletion(`TypeOf`(h), $\Psi[h]$, `GetAllSubstr`($\mathcal{E}^-$));
5: **if** $r \equiv \bot$ **then return** $\bot$;
6: **while** `True` **do**
7: $M \leftarrow$ SynthesizeFromDecomp($S[r/h], \Psi, \mathcal{E}^-$)
8: **if** $M \equiv \bot$ **then break**;
9: $M' \leftarrow M \cup [h \mapsto r]$;
10: **if** `Reject`($S[M'], \mathcal{E}^-$) **then yield** M';
11: **return** $\bot$;

Fig. 12. Sketch completion algorithm for a given decomposition.

Using our decomposition technique, we infer the following positive examples for each hole:

{□ : Name}	{□ : Country}	{□ : Year}$_1$	{□ : Year}$_2$
John Thomas Young Gilroy	Britain	1898	1985
Thomas Hudson	Britain	1701	1779
Thomas Couture	France	1815	1879

We conclude this subsection by stating the theorem about the soundness of decomposition:

Theorem 1. *Consider the synthesis problem with positive examples $\mathcal{E}^+$. Let S be a candidate sketch and let r be a completion of S mapping each hole h_i in S to a semantic regex r_i. If r satisfies all positive examples $\mathcal{E}^+$, then there exists some $\Psi \in$ GetNextDecomp$(S, \mathcal{E}^+)$ such that every r_i satisfies $\Psi[h_i]$.*

5.4 Compositional Type-Directed Synthesis

Next, we explain our compositional learning technique for synthesizing a semantic regex for a given sketch and decomposed specification. This algorithm, called SynthesizeFromDecomp, is shown in Figure 12. Given a sketch S, specification Ψ, and negative examples $\mathcal{E}^-$, the recursive SynthesizeFromDecomp procedure lazily generates possible sketch completions until it finds a regex that is globally consistent with the top-level specification.

To perform synthesis for a given specification, the algorithm starts by choosing one of the holes h in the sketch (line 2) and synthesizes a completion r for that hole *only* by calling GetNextCompletion at line 4. Then, the loop in lines 6–10 tries to find a completion for the remaining holes. In particular, in each iteration of the nested loop, the algorithm recursively calls SynthesizeFromDecomp to fill all remaining holes, assuming that h is replaced by r. If synthesis fails (i.e., $M \equiv \bot$ at line 8), the algorithm moves on to a different completion of h. Otherwise, it checks if the current solution (which is obtained by instantiating S with $M \cup [h \mapsto r]$) rejects all negative examples, and if so, returns this solution.

The final missing piece for our sketch instantiation algorithm is the GetNextCompletion procedure shown in Figure 13 which performs synthesis for a *single* hole. At a high level, this algorithm performs top-down enumerative search and uses a combination of types [Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016] and observational equivalence [Morris 1968] to prune the search space. As standard in top-down search, this algorithm utilizes the notion of *partial programs* [Feng et al. 2018, 2017], which can be thought of as an abstract-syntax tree where some of the nodes are labeled with non-terminals to be expanded later.

1: **procedure** GetNextCompletion($\tau_h, \mathcal{E}^+, \mathcal{E}_\star$)
input: Goal type τ_h, positive examples $\mathcal{E}^+$, and a set of strings $\mathcal{E}_\star$ for checking observational equivalence.
output: A program of type τ_h that matches $\mathcal{E}^+$
2: $P_0 \leftarrow ((s_\mathcal{G}, \tau_h), \emptyset)$;
3: $\mathcal{W} \leftarrow \{P_0\}$; $R \leftarrow \{\}$;
4: **while** $\mathcal{W} \neq \emptyset$ **do**
5: $P \leftarrow \mathcal{W}.remove()$;
6: **if** IsComplete(P) **then**
7: **if** $\vdash P : \tau_h \wedge \bigwedge_{e \in \mathcal{E}^+} \texttt{match}(e, P)$ **then**
8: $\mathcal{E} \leftarrow \{e \mid e \in \mathcal{E}_\star \wedge \neg\texttt{match}(e, P)\}$;
9: **if** $\mathcal{E} \notin R$ **then** $R \leftarrow R \cup \{\mathcal{E}\}$; **yield** P;
10: **else**
11: **for all** $P' \in$ Expand(P) **do**
12: **if** $\exists n \in \texttt{Nodes}(P').\ \texttt{IsComplete}(P'(n)) \wedge \vdash \texttt{TypeOf}(P'(n)) \not<: \texttt{GoalType}(n)$ **then**
13: **continue**;
14: **else if** $\exists e \in \mathcal{E}^+.\neg\texttt{match}(e, \texttt{OverApprox}(P'))$ **then**
15: **continue**;
16: $\mathcal{W} \leftarrow \mathcal{W} \cup \{P'\}$;
17: **return** $\perp$;

Fig. 13. Hole synthesis algorithm. OverApprox follows the procedure as described in Regel [Chen et al. 2020].

In more detail, the hole synthesis algorithm utilizes a worklist $\mathcal{W}$, which is initialized to a partial program P_0 with a single node (lines 2–3). Each node in the partial program is annotated with a grammar symbol (in this case, the start symbol $s_\mathcal{G}$) and its corresponding type (in this case, τ_h). Then, in each iteration of the loop in lines 4–16, the algorithm dequeues one of the partial programs P in the worklist and processes it. If the partial program is complete (meaning that all nodes are labeled with terminal symbols), the algorithm performs the following checks:

(1) **Type consistency:** If the type of P is *not* τ_h, P clearly does not have the intended type and is rejected (line 7).
(2) **Consistency with examples:** If P does not satisfy all positive examples $\mathcal{E}^+$, it does not satisfy the specification and is also rejected at line 7.
(3) **Observational equivalence:** If P rejects the *exact same set* of strings as a program the algorithm has previously encountered, it is redundant to consider P, as it is observationally equivalent to another solution P' that has been rejected. Hence, the algorithm only yields P as a solution if it is observationally different from a previously encountered solution (lines 8–9).

On the other hand, if the current partial program P is *incomplete* (meaning it has at least one "open" node labeled with a non-terminal), the algorithm chooses one of the open nodes and expands it using the available productions in the grammar (line 11). In particular, given an open node n labeled with a non-terminal N, the Expand procedure considers each production of the form $N \rightarrow \alpha$ and adds new nodes where each new node with a grammar symbol and its corresponding (inferred) type. However, because a resulting expansion P' may not necessarily be feasible, the algorithm performs two additional checks before adding P' to the worklist at line 16:

- **Type-directed feasibility check:** For each complete subprogram P_i of P', the algorithm checks if the actual type of P_i is a subtype of its annotated goal type (line 12). If this type feasibility

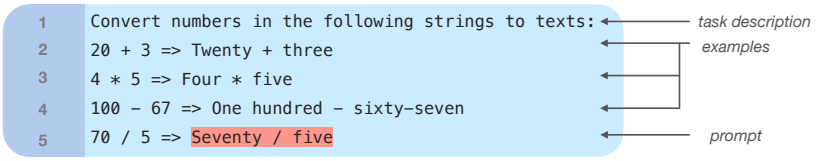

Fig. 14. Sample input for a few-shot string transformation to GPT-3 and its output is highlighted in red.

check fails for *any* node n, then program P' is pruned from the search space, and none of its expansions are considered.

- **Feasibility check using over-approximation:** Additionally, the algorithm constructs an over-approximating regular expression $r^\star$ that accepts every string that is accepted by any $r \in [\![P']\!]$ using the same `OverApprox` procedure from Section 5.3. If this over-approximation $r^\star$ fails to match one of the positive examples, P' is infeasible and therefore pruned away at lines 14–15.

Otherwise, P' is added to the worklist, and the search process continues until a solution is found.

Theorem 2. *Let R be the set of solutions returned by* GetNextCompletion$(\tau_h, \mathcal{E}^+, \mathcal{E}_\star)$. *We have:*

- **Soundness:** *Every $r \in R$ is a solution to the hole synthesis problem, meaning (1) r has type τ_h and (2) satisfies examples $\mathcal{E}^+$*
- **Completeness:** *If $r \notin R$, then r is either not a solution or is observationally equivalent to some $r' \in R$ for strings $\mathcal{E}_\star$.*

5.5 Sketch Generation

In the final part of this section, we describe our technique for generating typed sketches from examples. In particular, we employ few-shot prompting and build our sketch generator on top of GPT-3 [Brown et al. 2020].

5.5.1 Background on Few-Shot Prompting with LLMs. In recent years, large language models (LLMs) [Brown et al. 2020; Chowdhery et al. 2022] have made major breakthroughs in natural language understanding. These are models $P(\mathbf{x}) = P(x_1)P(x_2 \mid x_1) \ldots P(x_n \mid x_1, \ldots, x_{n-1})$ modeling a sequence as a product of distributions over each next word via the chain rule.

By showing LLMs a few examples of a task to perform and then giving them a test example, LLMs can perform that task on the test example via *in-context learning*, without retraining or fine-tuning the model's parameters. The user only needs to provide a few examples and invoke the model's next-word prediction capabilities (repeatedly taking the most likely next token under the model). To give a concrete example, consider the task of transforming numbers in strings to texts, a task that GPT-3 has not specifically been trained on. Figure 14 shows a typical usage scenario of GPT-3 when performing such a task: here, line 1 provides the task description, lines 2-4 provides a few examples, line 5 is the query, and the output of the model is highlighted in red.

5.5.2 Querying LLM for Sketches. To obtain typed sketches, our approach prompts GPT-3 with suitable queries.[3] As shown in Figure 15, the GetNextSketch procedure takes as input positive examples $\mathcal{E}^+$ and an optional infeasible sketch S_f, which is used in later iterations of the algorithm for sketch repair. Initially, the algorithm starts by querying GPT-3 for a sketch using the `GetSketch` procedure, as illustrated in Figure 16. The prompt to GPT-3 contains a task description, a manually-curated set of representative examples (in the form of a query and its desired output), and, finally, the prompt itself (lines 12–17 in Figure 16). The `GetSketch` procedure then attempts to parse the

[3]We consider sketches generated from models `text-davinci-003`, `code-davinci-002` and `gpt-3.5-turbo`.

1: **procedure** GetNextSketch($\mathcal{E}^+, S_f$)
input: A set of positive examples $\mathcal{E}^+$, and an optional infeasible sketch S_f.
output: A new sketch that has not been generated so far.
2: $S_{All} \leftarrow \emptyset$
3: **while** True **do**
4: **if** $S_f \equiv \bot$ **then**
5: $S \leftarrow$ GetSketch($\mathcal{E}^+$);
6: **if** $S \notin S_{All}$ **then** $S_{All} \leftarrow S_{All} \cup \{S\}$; **yield** S;
7: **else**
8: **while** HasRepair($S_f, \mathcal{E}^+$) **do**
9: $\mathcal{S}, \Psi \leftarrow$ LocateError($S_f, \mathcal{E}^+$);
10: $S \leftarrow \mathcal{S}[h_i \mapsto \text{GetSketch}(\Psi[h_i]) \mid h_i \in \text{MetaHoles}(\mathcal{S})]$
11: **if** $S \notin S_{All}$ **then** $S_{All} \leftarrow S_{All} \cup \{S\}$; **yield** S;
12: **return** $\bot$;

Fig. 15. Sketch generation procedure. GetSketch($\mathcal{E}^+$) prompts the neural model for a new sketch, as illustrated in Figure 16.

```
Summarize the structure of the following positive examples in the form of a regular expression sketch.
Use {??: <semantic type>} to represent the unknown part of the sketch.
Positive examples:
– (David J. Alexander), Marc Henri Sempere and Jocelyn Bulow
– (Connie Wong), Sai Wong
– (Amin Abughosh) and Joseph Abugosh and Abeer Elafifi
Sketch:
– \({??: Person}\) ((&|and|,) {??: Person})+

…… (7 more examples)

Summarize the structure of the following positive examples in the form of a regular expression sketch.
Use {??: <semantic type>} to represent the unknown part of the sketch.
Positive examples:
– …
Sketch:
–
```

task description + examples

prompt

Fig. 16. GPT-3 input structure for generating a sketch for the semantic string matching task.

model's output into a typed sketch; however, there is no guarantee that the GPT-3 output will belong to our sketch grammar. Hence, if parsing fails, the GetSketch procedure keeps prompting GPT-3 for a new sketch until the model's output is parseable.[4]

In future invocations of GetNextSketch, this procedure may be invoked with an infeasible sketch S_f that needs to be repaired. Lines 8–11 of Figure 15 deal with this sketch repair aspect of the algorithm. Specifically, given the infeasible sketch S_f and positive examples $\mathcal{E}^+$, LocateError produces a *repair specification*, which consists of a so-called *meta-sketch* $\mathcal{S}$ and a specification Ψ. A meta-sketch is like a sketch except that it contains *untyped* "meta-holes" that need to be instantiated with a *typed sketch*. The specification Ψ maps each meta-hole in $\mathcal{S}$ to a set of positive examples. Such a meta-sketch is instantiated into a regular sketch by querying GPT-3 via the GetSketch procedure for each of the meta-holes h_i in $\mathcal{S}$ and its corresponding examples $\Psi[h_i]$.

[4]Past work has explored few-shot semantic parsing from natural language into DSLs using structured natural language as an intermediate representation [Shin and Van Durme 2022]; however, more recent work has shown that LLMs can do well at this task without such guidance, even in the presence of adversarial perturbations [Zhuo et al. 2023].

$$\textsc{Sketch-Single-Fail}\ \frac{\begin{array}{c}\texttt{Match}(\texttt{OverApprox}(S), \mathcal{E}^+) \equiv \emptyset \\ S = f(S_1, \ldots, S_n) \\ \exists i \in \{1, \ldots, n\} \quad r_i^\star = \texttt{OverApprox}(S[\square/S_i]) \\ (\mathcal{E}_1^+, \ldots, \mathcal{E}_i^+, \ldots, \mathcal{E}_n^+) \in \texttt{Match}(r_i^\star, \mathcal{E}^+) \\ \mathcal{E}_i^+ \vdash S_i \hookrightarrow \mathcal{S}_i, \Psi_i\end{array}}{\mathcal{E}^+ \vdash S \hookrightarrow S[\mathcal{S}_i/S_i], \Psi_i}$$

$$\textsc{Sketch-Multi-Fail}\ \frac{\begin{array}{c}\texttt{Match}(\texttt{OverApprox}(S), \mathcal{E}^+) \equiv \emptyset \\ S = f(S_1, \ldots, S_n) \\ \forall i \in \{1, \ldots, n\} \quad r_i^\star = \texttt{OverApprox}(S[\square/S_i]) \\ \texttt{Match}(r_i^\star, \mathcal{E}^+) \equiv \emptyset\end{array}}{\mathcal{E}^+ \vdash S \hookrightarrow \square, [\square \mapsto \mathcal{E}^+]}$$

$$\textsc{Sketch-Nested-Fail}\ \frac{\begin{array}{c}S = f(S_1, \ldots, S_n) \\ (\mathcal{E}_1^+, \ldots, \mathcal{E}_n^+) \in \texttt{Match}(\texttt{OverApprox}(S), \mathcal{E}^+) \\ \mathcal{E}_i^+ \vdash S_i \hookrightarrow \mathcal{S}_i, \Psi_i \quad i \in \{1, \ldots, n\}\end{array}}{\mathcal{E}^+ \vdash S \hookrightarrow f(\mathcal{S}_1, \ldots, \mathcal{S}_n), \texttt{Merge}(\Psi_1, \ldots, \Psi_n)}$$

$$\textsc{Hole-Fail}\ \frac{\exists e \in \mathcal{E}^+.\ \texttt{SemanticType}(e) \not<: \tau}{\mathcal{E}^+ \vdash \{\square : \tau\} \hookrightarrow \square, [\square \mapsto \mathcal{E}^+]} \qquad \textsc{Hole-Correct}\ \frac{\forall e \in \mathcal{E}^+.\ \texttt{SemanticType}(e) <: \tau}{\mathcal{E}^+ \vdash \{\square : \tau\} \hookrightarrow \{\square : \tau\}, \emptyset}$$

$$\textsc{Concrete-Fail}\ \frac{\texttt{Match}(r, \mathcal{E}^+) \equiv \emptyset}{\mathcal{E}^+ \vdash r \hookrightarrow \square, [\square \mapsto \mathcal{E}^+]} \qquad \textsc{Concrete-Correct}\ \frac{\texttt{Match}(r, \mathcal{E}^+) \neq \emptyset}{\mathcal{E}^+ \vdash r \hookrightarrow r, \emptyset}$$

Fig. 17. Procedure for LocateError.

Finally, we turn our attention to the LocateError procedure, which is presented as inference rules in Figure 17. These rules derive judgments of the following shape:

$$\mathcal{E}^+ \vdash S \hookrightarrow \mathcal{S}, \Psi$$

meaning that $(\mathcal{S}, \Psi)$ is a repair specification for infeasible sketch S and examples $\mathcal{E}^+$. The fault localization rules in Figure 17 largely resemble GetNextDecomp for performing decomposition in that they use over-approximations. We explain these rules in more detail below.

Sketch-Single-Fail. This rule applies to a sketch S of the form $f(S_1, \ldots, S_n)$ where (1) there is at least one positive example that is not matched by the over-approximation of S (premise on the first line) and (2) where only one of the sub-sketches S_i is faulty. To determine whether condition (2) holds, this rule replaces the entire sub-sketch S_i with a single hole and then checks whether the over-approximation of the resulting sketch can accept all positive examples. If so, it recursively performs fault localization on S_i and returns a meta-sketch by replacing S_i in S with its corresponding meta-sketch $\mathcal{S}_i$.

Sketch-Multi-Fail. This rule is similar to the first one except that it deals with the scenario where there are multiple faulty sub-sketches. That is, even after we replace any individual sub-sketch with a hole, there is *still* at least one positive example that is not matched by the over-approximation. In this case, we generate a meta-sketch that consists of a single hole.

Sketch-Nested-Fail. This rule also applies to a sketch S of the form $f(S_1, \ldots, S_n)$ but considers the case where the over-approximation of S matches all the positive examples. However, as the sketch is infeasible, there must nonetheless be at least one problem inside the next sub-sketches.

Hence, our fault localization technique recursively localizes the error in the sub-sketches and returns the merged result.

Hole-Fail. This rule applies to the case where the type of a hole is incorrect in that its annotated type is inconsistent with at least one of the positive examples. In this case, our algorithm generates a meta-sketch by erasing the type annotation of this hole.

Hole-Correct, Concrete-Correct. Since these rules apply to base cases without any problems, fault localization returns the original sketch.

Concrete-Fail. This rule applies to the case where a concrete regex does not match at least one of the examples. In this case, we simply replace the concrete regex with a meta-hole.

Example 5.3. Consider the positive examples from Section 2 and the following sketch:

$$\{\square : \mathsf{Name}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Country}\} \cdot \text{“, ”} \cdot \{\square : \mathsf{Year}\}$$

Suppose the synthesizer concluded this sketch to be infeasible since the string “1898-1985” cannot be identified as a year and sends this as a failed sketch to the sketch generator. To repair this sketch, we follow the Sketch-Nested-Fail rule to recursively traverse through each part of the sketch until we locate the faulty hole, $\{\square : \mathsf{Year}\}$. We then gather the positive examples that should be matched by this hole, which are “1898-1985”, “1701-1779” and “1815-1879”, and replace the faulty typed hole with a new hole with no type (rule Hole-Fail). With the generated repair specification, we query GPT-3 to generate a new sketch for the faulty hole, and it returns a new sketch $\{\square : \mathsf{Year}\} \cdot \text{“ – ”} \cdot \{\square : \mathsf{Year}\}$.

6 IMPLEMENTATION

We have implemented our synthesis algorithm in a new tool called Smore written in Python. In this section, we provide implementation details about different components of Smore.

Implementation of the semantic matching construct. Our tool heavily relies on the use of GPT-3 to identify the semantic meanings of strings.[5] Our few-shot prompt (following the discussion in Section 5.5) to accomplish this is shown in Figure 18. The input begins with a task description that asks the model to identify *all possible substrings* of a particular semantic type, and we instruct the model to return “none” if it does not find any. Following the task descriptions, we provide 8 examples,[6] each of which shows the structure of a query: the first line provides the string of interest, and the second line specifies the semantic type of interest. Furthermore, we provide sample outputs for each example in the expected output format.

Implementation of checking observational equivalence. In the GetNextCompletion procedure (Figure 13), we use the set $\mathcal{E}_\star$ to prune out programs that are observationally equivalent to previously synthesized programs. In Figure 12, $\mathcal{E}_\star$ corresponds to all substrings of the negative examples $\mathcal{E}^-$, but this set might contain too many strings in practice, leading to considerable overhead in the observational equivalence check. To address this issue, we only obtain the substring of the negative examples that are relevant to the specific hole under consideration. Specifically, we identify the relevant substrings of the negative examples using the overapproximation of the sketch. If a negative example can already be rejected by the overapproximation of the sketch, it is safe to conclude that any instantiation of the sketch can reject this negative example and therefore that this example is irrelavant. For those negative examples that can be matched by the overapproximation, we identify substrings that might be matched by each hole of the sketch and

[5] We use the `text-davinci-003` model.

[6] We provide all the in-context examples we use in the supplementary material.

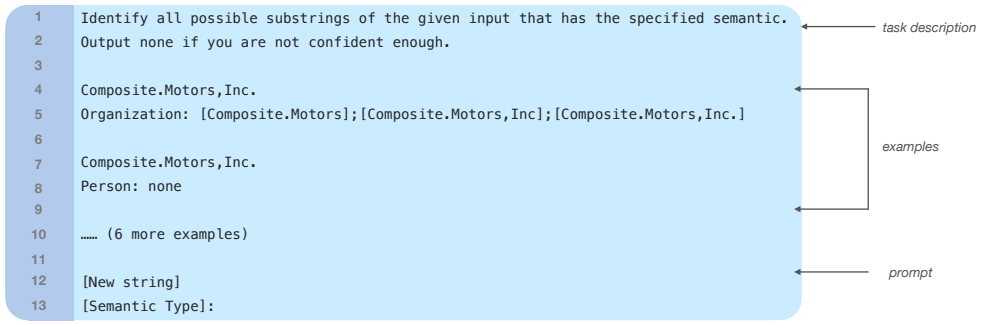

Fig. 18. GPT-3 input structure for identifying substring of specific semantics. [New string] is a placeholder for the string we are querying about, and [Semantic Type] is the semantics we are asking the model to identify.

use those to check observational equivalence. This strategy provides the full benefits of checking observational equivalence but significantly reduces overhead in some cases. We illustrate this discussion through the following example:

Example 6.1. Consider a synthesis task with the following positive and negative examples:

Positive Examples	Negative Examples
14+15	1+18
15+17	2+6
16+13	7-12

Suppose that the generated sketch is $\{\square : \mathsf{Integer}\}[+]\{\square : \mathsf{Integer}\}$. Using the overapproximation $(.*)[+](.*)$, we can already reject the negative example "7-12", so that negative example is not relevant for selecting different instantiations of the sketch. To find the rest of the relevant strings, notice that the overapproximation decomposes the first negative example by sending "1" to the first hole and "18" to the second hole (as the negative example for each of the holes). Following the same procedure, we obtain "1" and "2" as the relevant substring for the first hole. Now, considering the two synthesized programs $\{v : \mathsf{Integer} \mid x > 4\}$ and $\{v : \mathsf{Integer} \mid x > 5\}$ for the first hole, we can safely conclude that these two programs are observationally equivalent with respect to the negative examples since both programs reject the same set of negative examples (specifically, example "1+18" and "2+6").

Ranking heuristic. Because there are often multiple semantic regexes that are consistent with the provided examples, it is important to use a ranking heuristic to choose between possible solutions. To this end, our method prioritizes sketches that maximize the number of type annotations, and it prefers decompositions that minimize the number of holes that are assigned empty strings as positive examples. Finally, when choosing between multiple regexes for a given hole, our algorithm prefers those with smaller ASTs, first ranked by height and then by the number of nodes.

Hyperparameters. The Smore system has a hyperparameter that controls the maximum depth of the synthesized programs for each hole, which is set to 4 by default. For GPT-3 hyperparameters, we set the temperature to 0 (corresponding to greedy inference) and maximum length to 256.[7]

[7]We also define the suitable stop sequences for each prompt to ensure GPT-3 doesn't have to generate 256 tokens.

Table 1. Description of the sample tasks used in the evaluation.

Domain	Task Description
Business	Restaurants that are created before 2000 or after 2010 Businesses located in California
Sales	Products with Intel CPU that have more than 8GB memory TVs of size less than 50' or resolution less than 1080P
Retail	Website titles that start with product names and are followed by a url Product names that contain measurement information
Marketing	Software engineering jobs that have specified working locations Business names with at least 3 words
Account	Email addresses that have a country domain and where the username ends with number Software versions with at least 10 minor updates and more than one patch
Stock	Company names with 3-letter abbreviation Company names with ticker symbols containing special characters
Science	Location description with format State; County; More details Locations that are less than 11 miles from a road
Server	Apache logs with file id >=151000 or in the format of a zip file with id <= 50 Photo files with numbers in their name
Museum	Purchase made by using three different funds Artwork with two artists born in the 14th century
Exhibition	Dimension of item between 10 and 50 inches Item that is associated with at least three categories

7 EVALUATION

In this section, we describe the results of our experimental evaluation, which is designed to answer the following research questions:

- **RQ1.** How does our proposed data extraction approach compare against existing approaches?
- **RQ2.** How does our synthesis algorithm compare to relevant baselines?
- **RQ3.** How important are the different components of our synthesis algorithm for successfully solving these benchmarks?
- **RQ4.** Do semantic regexes help humans more effectively solve data extraction tasks compared to standard regexes?

Benchmarks. To answer these questions, we evaluate Smore on 50 data extraction tasks involving 10 different datasets, which cover a wide range of domains like sales, science, and art. These datasets contain many different string formats and involve a large variety of entities. Out of 50 tasks, 34 of the tasks require at least one built-in semantic type and 33 of the tasks require at least one custom semantic type. We consider an average of 5 data extraction tasks for each dataset and manually label a subset of the strings in each dataset as positive or negative for each task. Specifically, we use 6 of the manually labeled examples for training and the rest for testing. Table 1 describes some example tasks for each domain.

Experimental Setup. All of our experiments are conducted on a machine with an Apple M2 Max CPU and 32GB of physical memory, running the macOS 13.2.1 operating system. We run GPT-3 through the OpenAI API. For each task, we set the timeout to 60 seconds (excluding the time to query OpenAI).

Table 2. Evaluation results for Smore and data extraction baselines. P means precision and R means recall.

Tool	# Finished	P	R	F_1	Synth Time (s)	Matching Engine
ChatGPT-Regex-Synth	23/50	0.60	0.40	0.44	-	Regex
ChatGPT-Exec	-	0.60	0.77	0.65	-	ChatGPT
FlashGPT	15/50	0.45	0.83	0.58	3.16	FlashGPT DSL
Smore	48/50	0.94	0.84	0.87	4.96	Semantic Regex

7.1 Comparison with Other Automated Data Extraction Techniques

There are several techniques that can be used to automate data extraction tasks. To answer our first research question, we compare Smore against the following alternative data extraction approaches:

- ChatGPT-Regex-Synth [OpenAI 2022]: One way to automate data extraction is to synthesize standard regexes from positive and negative examples. To evaluate this approach, we use ChatGPT to synthesize standard regexes. If the synthesized regex rejects the positive examples or accepts the negative examples, we ask ChatGPT to synthesize a different regex for up to ten iterations.[8]
- ChatGPT-Exec [OpenAI 2022]: Another way to automate data extraction is to directly use ChatGPT. To evaluate this approach, we provide ChatGPT with positive and negative examples and then query it about strings in the test set. Hence, this approach does not require synthesizing a program; instead, it invokes ChatGPT on every test example.
- FlashGPT [Verbruggen et al. 2021]: Recent work has proposed an extension of FlashFill, called FlashGPT, that can query GPT-3 in addition to performing syntactic transformations and pattern matching. For our third baseline, we also compare against FlashGPT by giving it positive and negative examples and then using it to synthesize a program in their DSL.

Main results. Our main results are summarized in Table 2. We evaluate each tool in terms of precision, recall, and F1 score on the test set as well as synthesis time and number of benchmarks solved. The **P**, **R**, and $\mathbf{F_1}$ columns represent the precision, recall, and F_1 score on the test set. Smore achieves the highest precision, recall, and F_1 score among all the alternative data extraction approaches. In particular, Smore outperforms the second best approach, namely ChatGPT-Exec, by 22% in terms of F_1 score. While ChatGPT-Exec and FlashGPT have fairly high recall, they have low precision. ChatGPT-Regex-Synth has similar precision to ChatGPT-Exec but has very low recall on the test set. Finally, FlashGPT and Smore are close in terms of recall, but Smore significantly outperforms FlashGPT in terms of precision (for benchmarks that both tools can synthesize within the time limit).

Next, the column labeled "# Finished" in Table 2 shows the number of tasks that each tool is able to solve. For Smore and FlashGPT, solving a benchmark means they were able to find a program consistent with the positive and negative examples within the 60-second time limit. Solving a benchmark for ChatGPT-Regex-Synth means finding a regex consistent with the examples within 10 iterations.[9] Since ChatGPT-Exec does not perform synthesis, this column is not applicable to it. Among all the synthesis-based approaches, Smore terminates for 48 out of 50 tasks, which is around twice as many as ChatGPT-Regex-Synth and around 3 times as many as FlashGPT.

Finally, the column labeled "Synth time" shows the synthesis time in seconds for FlashGPT and Smore. Since we exclude the time to query OpenAI from synthesis time (this only takes at most a few seconds), this column is not applicable to ChatGPT-Regex-Synth. As we can see from this

[8]We set the temperature to 0.7 for sampling.

[9]Recall we keep querying for a different regex for up to 10 times if the synthesized regex does not match the examples.

Table 3. Evaluation results for our tool and synthesis baselines. P means precision and R means recall.

Tool	# Finished	P	R	F_1	Synth Time (s)
ChatGPT-Synth	6/50	0.76	0.67	0.71	-
Smore-NoSketch	12/50	0.79	0.85	0.79	15.27
Smore	48/50	0.94	0.84	0.87	4.96

column, the synthesis time of Smore is around 5 seconds, so it takes slightly longer than FlashGPT (which takes around 3 seconds) for the 14 tasks that both of the tools can solve. However, Smore is able to synthesize a program for three times as many tasks as FlashGPT.

Failure Analysis for the baselines. To provide some insight into the shortcomings of existing approaches, we briefly discuss the failure cases of the baselines. As expected, ChatGPT-Regex-Synth struggles with tasks that are hard to represent as regular expressions, such as matching all businesses that are in California. Although FlashGPT combines neural and symbolic constructs, its neural component processes positive and negative examples rather than semantic types. In other words, the neural constructs directly query GPT with positive and negative examples rather than querying whether a string matches a certain type. As a result, it frequently generates trivial programs that directly invoke GPT with the training examples as input. Hence, it ultimately ends up sharing the same limitations as ChatGPT-Exec.

Failure analysis for the Smore. We examined instances where Smore is unable to complete the synthesis task within the allotted time and found that it encounters difficulties in tasks that demand a higher level of granularity from semantic pattern matching. For example, consider a task that involves finding restaurant names containing a person's name. For the positive example "Alice Chinese Bistro", the entity matcher may fail to recognize "Alice" as a person's name, causing Smore to fail to synthesize a program consistent with all examples.

7.2 Comparison with Other Semantic Regex Synthesis Techniques

To answer our second research question, we compare the neural-guided synthesis algorithm of Smore against the following two purely-neural or purely-symbolic baselines:

- ChatGPT-Synth [OpenAI 2022]: To evaluate whether a purely neural synthesizer can solve these benchmarks, we use ChatGPT to create a synthesizer for semantic regexes. Specifically, our ChatGPT-Synth baseline queries ChatGPT to synthesize a *semantic regex* that matches all positive examples and rejects all negative examples. If the generated semantic regex is inconsistent with the examples, we query it again for a different one. We repeat this process for up to 10 times, as done with our ChatGPT-Regex-Synth baseline in the previous subsection.
- Smore-NoSketch: To evaluate a semantic regex synthesis without neural sketch generation, we create a variant of Smore that does not start with a sketch (i.e., it uses $\{\square : \text{Any}\}$ as the sketch).

The results of this comparison are presented in Table 3. As we can see from the "# Finished" column, ChatGPT-Synth can synthesize a semantic regex consistent with the examples for only 6 of the 50 benchmarks within 10 iterations. On the other hand, Smore-NoSketch times out on the majority of benchmarks and only finds a consistent regex for 12 of the 50 benchmarks. Furthermore, for semantic regexes that both Smore-NoSketch and Smore can synthesize, Smore is significantly faster. Table 3 also shows that Smore outperforms both of these synthesizers in terms of F_1-score when evaluated on the test data. In particular, among all tasks that can be solved by both Smore and ChatGPT-Synth, Smore achieves an F_1 score of 0.94 versus 0.71, and, among tasks that can be solved by both Smore and Smore-NoSketch, Smore achieves an F_1 score of 0.88 versus 0.84.

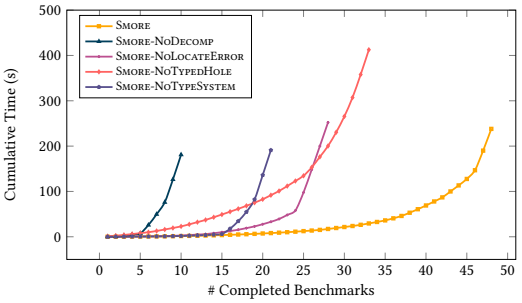

Fig. 19. Solved tasks over time.

Task	Manual-Regex	Manual-SemRegex	Smore
1	0.31	0.93	1.00
2	0.65	0.65	0.89
3	0.55	0.81	0.89
4	0.63	0.71	0.89
Average	0.54	0.78	0.92

Fig. 20. Average F_1 scores achieved by manually-written regexes, semantic regexes, and synthesized semantic regexes.

Table 4. Evaluation results for our tool and the no-sketch variant. P means precision and R means recall.

Tool	# Finished	P	R	F_1	Synth Time (s)
ChatGPT-Synth	6/50	0.76	0.67	0.71	-
ChatGPT-Synth-Repair	12/50	0.76	0.67	0.71	-
Smore	48/50	0.94	0.84	0.87	4.96

7.3 Ablation Study

In this section, we describe two ablation studies to assess the relative impact of different components of Smore: one evaluates the impact of the synthesis techniques proposed in Section 5.2-5.4, and the other one evaluates the impact of generating sketches rather than concrete regexes.

Ablations of components of the synthesis techniques. To evaluate the effectiveness of the proposed synthesis techniques, we consider the following ablations:

- Smore-NoDecomp: A variant of Smore that does not perform compositional sketch completion. In particular, this variant does not infer positive examples for each hole.
- Smore-NoTypedHole: A variant of Smore that does not use typed sketches. That is, each hole in the sketch is annotated with type Any.
- Smore-NoLocateError: A variant of Smore that does not perform error localization for sketch repair. Instead, it queries GPT-3 for a new sketch through sampling.
- Smore-NoTypeSystem: A variant of Smore that does not perform type-directed synthesis.

The results of this ablation study are presented in Figure 19, which shows the number of benchmarks completed (x-axis) within the given time limit (y-axis). As we can see from the gap between the five lines, Smore is significantly faster than all other variants and achieves a speedup of 14× compared to the second-fastest baseline, Smore-NoTypedHole. Hence, this ablation study shows that all algorithmic components proposed in this paper are important for speeding up the synthesis.

Ablations of sketch generations. To understand the significance of generating sketches as opposed to concrete semantic regexes, we introduce a new baseline named ChatGPT-Synth-Repair. This baseline extends the ChatGPT-Synth baseline from Section 7.2 with program repair. Specifically, it first generates a concrete program using ChatGPT (using a similar prompt as the ChatGPT-Synth baseline). If the generated program does not satisfy all the positive and negative examples provided, it then performs the error localization and repair strategies presented in Section 5.5.

The results of this ablation study are presented in Table 4. For clarity, we also include the results of ChatGPT-Synth and Smore from Section 7.2 to show the difference evaluation results. This ablation leads to the two following observations:

- ChatGPT-Synth-Repair solves 6 more benchmarks compared to ChatGPT-Synth. This shows that our sketch repair technique can also be generalized to concrete program repair.
- Although ChatGPT-Synth-Repair exhibits superior performance over ChatGPT-Synth, it is not comparable to the performance of Smore, which leverages sketches for synthesis. This underscores the pivotal role sketches play in enhancing the tool's efficacy. Upon analysis, we find that ChatGPT-Synth-Repair is able to accurately locate the error when it does not generate the desired program. However, ChatGPT struggles to generate a new program that precisely separates positive from negative examples. In contrast, with Smore, since we produce sketches, ChatGPT only needs to generate segments of the program it is confident about, delegating the uncertain parts or those demanding intricate reasoning to the program synthesizer.

7.4 User Study

We conducted a user study to assess the efficacy of semantic regexes in aiding humans with data extraction tasks compared to standard regexes. We recruited 13 participants, consisting of 3 CS undergraduate students, 6 CS graduate students, and 4 professional software engineers who regularly use regexes in their work. We asked each participant to complete 4 data extraction tasks by writing a regex. The participants were given 5 minutes for each task and asked to write standard regexes for two randomly chosen tasks (out of the 4 total tasks) and semantic regexes for the other two. The four tasks used in the study are simplified versions of the benchmarks used in our evaluation — we intentionally simplified the tasks so that they are doable within 5 minutes.

Setup. To conduct this user study, we developed a command-line interface for Smore. For each task, the interface initially displays the prompt for the task (including 3 positive and negative examples) and then asks the user to input their answer. The tool randomly determines whether the answer should be a standard or semantic regex and only accepts user answers in the correct format. Upon entering a regex, the interface evaluates it against the test set and informs the user of their regex's performance, allowing unlimited attempts to enter a new regex within the 5-minute time limit. The details of the user study protocol are provided in the supplementary material.

Results. We evaluate the quality of the regexes in terms of their F_1 score on the test set. For each task, Table 20 presents F_1 scores for (a) manually-written standard regexes ("Manual-Regex"), (b) manually-written *semantic* regexes ("Manual-SemRegex"), and (c) semantic regexes generated automatically by Smore (the "Smore" column). Since some of the manually-written regexes have a precision or recall score of 0, the F_1 score is undefined. In Table 20, we only show average F_1 score across regexes for which the F_1 score is defined.

As we can see from Figure 20, manually-written *semantic* regexes achieve a better overall F_1 score (0.78) compared to standard regexes, for which the F_1 score is 0.54. We ran a two-way ANOVA to find the most significant factor affecting the F_1 score. In particular, we model the F_1 score as the dependent variable and the type of tool and task as independent variables. The ANOVA analysis shows that the "task" variable has a high p-value of 0.57, which indicates it does not have a significant impact on the F_1 score. On the other hand, the "type of tool" variable has a low p-value of 0.003, suggesting that the type of tool used has a significant impact on user performance. The analysis result indicates that participants are more effective at performing these types of data extraction tasks using semantic regexes than with standard regexes. Another interesting aspect of Figure 20 is that the semantic regexes learned by Smore seem to be *even* more effective than

manually-written semantic regexes. In particular, for these four tasks, Smore learns regexes that achieve an overall F_1 score of 0.92 compared to the F_1 score (0.78) of manually-written semantic regexes. This result suggests that our proposed learning technique has the potential to improve productivity even for expert users who are generally comfortable with writing regexes.

8 RELATED WORK

In this section, we survey related work on program synthesis and data extraction.

Learning regexes from examples. There is a large body of prior research on learning regular expressions from positive and negative examples [Alquezar and Sanfeliu 1994; Angluin 1987; Firoiu et al. 1998; Gold 1978; Parekh and Honavar 1996, 2001; Rivest and Schapire 1989]. Our work builds on existing works that prune partial programs by evaluating the examples with respect to over- and under-approximations [Chen et al. 2020; Lee et al. 2016; Ye et al. 2021]. In this work, we not only use the over-approximations for pruning but also for decomposing the synthesis tasks.

Information Extraction from Semi-Structured Data. Past work has investigated similar extraction tasks, particularly for extracting lists from web sources [Chen et al. 2021a; Lin et al. 2020; Pasupat and Liang 2014; Raza and Gulwani 2020], answering questions based on tables [Pasupat and Liang 2015], and general information extraction from tabular data [Le and Gulwani 2014; Wu et al. 2018]. Recent work has specifically employed LLMs to extract information from tables [Cheng et al. 2023] or raw text [Dunn et al. 2022]. Despite the prevalence of neural-based approaches that emphasize data semantics, our work uniquely targets the integration of both semantic and symbolic aspects of the data structure.

Neurosymbolic DSLs. Recent work has considered so-called *neurosymbolic DSLs* with both standard language constructs and neural components [Andreas et al. 2016a,b; Bastani et al. 2022; Chen et al. 2021a; Cheng et al. 2023; Gaunt et al. 2017; Huang et al. 2020b; Jiang et al. 2021; Shah et al. 2020; Valkov et al. 2018; Verbruggen et al. 2021]. Among these, most relevant to our approach are FlashGPT [Verbruggen et al. 2021] and Binder [Cheng et al. 2023]. FlashGPT augments the DSL used in Flashfill [Gulwani 2011] with semantic transformation operators that can be used to reason about the semantic properties of the input. However, FlashGPT relies on in-context examples and does not utilize explicit semantic types, which hinders its ability to reason about combined semantic and symbolic properties. On the other hand, Binder [Cheng et al. 2023] proposes a new program structure that extends programming languages, such as SQL, with a function that allows querying large language models (in particular, Codex). However, the constructs proposed in Binder focus mainly on SQL-related tasks and do not transfer well to the string-matching domain.

Program Synthesis Using LLMs. The growing interest in leveraging LLMs for program synthesis [Austin et al. 2021; Chen et al. 2021b; Cheng et al. 2023; Nijkamp et al. 2023; Zhou et al. 2023] stems from general-purpose models like ChatGPT and Codex demonstrating code generation capabilities from various specifications, including natural language and input-output examples. However, these models often generate code that violates syntactic and semantic rules due to their limited understanding of program syntax and semantics. To address this, several approaches [Jain et al. 2022; Poesia et al. 2022; Rahmani et al. 2021] integrate LLMs with symbolic methods like program analysis to improve code quality. In our work, we use LLMs to generate sketches and introduce a sketch repair technique to handle cases where the LLM fails to generate accurate sketches.

Compositional program synthesis. Various approaches have been proposed for compositional program synthesis [Bansal et al. 2023; Feser et al. 2015; Huang et al. 2020a; Polozov and Gulwani

2015]. Among these works, both λ^2 [Feser et al. 2015] and FlashMeta [Polozov and Gulwani 2015] perform compositional PBE by inferring input-output examples for sub-programs using the inverse semantics. In another example, Raza et al. [Raza et al. 2015] rely on the natural language description to decompose the synthesis problems into smaller sub-problems. Furthermore, Zhang et al. [Zhang et al. 2021] decompose the synthesis task into simpler sub-problems in the domain of UDF-to-SQL translation using a dataflow graph. Our work differs from prior research by presenting a new decomposition strategy on a typed sketch in the context of synthesizing string-matching programs. While our decomposition approach helps reject incorrect programs using inferred positive examples, the full result must still be tested against the negative examples to ensure correctness.

Semantic Checks for String Matching. There has been prior work in combining string matching with semantic matching [Greenberg et al. 2022; Kozen 1997]; for example, Kleene algebra with tests (KATs) [Kozen 1997] combines Kleene and Boolean algebra. While our semantic matching construct can be conceptually viewed as a semantic guard for string matching, one key difference is that the predicate (i.e. the "test") part of the language in KATs is restricted to boolean algebra, whereas our vocabulary of predicates is much richer, including function invocations and machine learning models. Furthermore, the intended application domains are quite different: our proposed semantic regexes are intended for textual data extraction, whereas KATs have traditionally been used in the context of verification.

9 CONCLUSION

We have presented Smore, a new synthesis-powered system for data extraction. The key idea behind Smore is the concept of *semantic regexes*, which augments the syntactic pattern matching capabilities of regexes with a semantic pattern matching construct of the form $\{v : \tau \mid \phi\}$ which matches strings that have entity type τ and that satisfy logical predicate ϕ when interpreted as an instance of τ. As shown in our user study from Section 7.4, semantic regexes allow users to more easily perform data extraction tasks that are hard to do using standard regular expressions.

In addition to proposing semantic regexes, we have also described a learning algorithm that can synthesize semantic regexes from examples. Our synthesis algorithm is neural-guided and uses a LLM to generate a *typed sketch* where unknown parts of the regex have useful type annotations that are used to guide the search. Our synthesis algorithm is compositional and uses type-directed reasoning to find a completion of each hole in the sketch. Our evaluation shows that our proposed approach outperforms alternative data extraction techniques in terms of precision, recall, and F_1 score. Our evaluation also shows the advantages of combining neural-guided sketch generation with type-directed compositional synthesis in terms of synthesis time.

DATA-AVAILABILITY STATEMENT

The software that supports Section 5 and Section 7 is available on Zenodo [Chen et al. 2023].

ACKNOWLEDGMENTS

We thank our anonymous reviewers, Shankara Pailoor, Anders Miltner, Xi Ye, Nathan Taylor, Josh Hoffman, Maxine Xin, Cole Vick, Sammy Thomas for their helpful feedback and support. This material is based upon work supported by the National Science Foundation under Grant No. 1918889 and Grant No. 1762299. 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

R. Alquezar and A. Sanfeliu. 1994. Incremental Grammatical Inference From Positive And Negative Data Using Unbiased Finite State Automata. In *In Proceedings of the ACL'02 Workshop on Unsupervised Lexical Acquisition*. 291–300.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016a. Learning to Compose Neural Networks for Question Answering. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, San Diego, California, 1545–1554. https://doi.org/10.18653/v1/N16-1181

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016b. Neural Module Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 39–48. https://doi.org/10.1109/CVPR.2016.12

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

Suguman Bansal, Giuseppe De Giacomo, Antonio Di Stasio, Yong Li, Moshe Y. Vardi, and Shufang Zhu. 2023. Compositional Safety LTL Synthesis. In *Verified Software. Theories, Tools and Experiments.: 14th International Conference, VSTTE 2022, Trento, Italy, October 17–18, 2022, Revised Selected Papers* (Trento, Italy). Springer-Verlag, Berlin, Heidelberg, 1–19. https://doi.org/10.1007/978-3-031-25803-9_1

Osbert Bastani, Jeevana Priya Inala, and Armando Solar-Lezama. 2022. *Interpretable, Verifiable, and Robust Reinforcement Learning via Program Synthesis*. Springer International Publishing, Cham, 207–228. https://doi.org/10.1007/978-3-031-04083-2_11

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantics* 7, 3 (2009), 154–165. https://doi.org/10.1016/j.websem.2009.07.002 The Web of Data.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Isil Dillig. 2023. *Data Extraction via Semantic Regular Expression Synthesis*. https://doi.org/10.5281/zenodo.8144182

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021a. Web Question Answering with Neurosymbolic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3453483.3454047

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding Language Models in Symbolic Languages. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=lH1PV42cbF

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat,

Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *ArXiv* abs/2204.02311 (2022).

Alexander Dunn, John Dagdelen, Nicholas Walker, Sanghoon Lee, Andrew S. Rosen, Gerbrand Ceder, Kristin Persson, and Anubhav Jain. 2022. Structured information extraction from complex scientific text with fine-tuned large language models. *arXiv* 2212.05238 (2022).

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* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, 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. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Laura Firoiu, Tim Oates, and Paul R. Cohen. 1998. Learning Regular Languages from Positive Evidence. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*. 350–355.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable Programs with Neural Libraries. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 1213–1222.

E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978), 302 – 320.

Michael Greenberg, Ryan Beckett, and Eric Campbell. 2022. Kleene Algebra modulo Theories: A Framework for Concrete KATs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 594–608. https://doi.org/10.1145/3519939.3523722

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* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Jiani Huang, Calvin Smith, Osbert Bastani, Rishabh Singh, Aws Albarghouthi, and Mayur Naik. 2020b. Generating Programmatic Referring Expressions via Program Synthesis. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 4495–4506. https://proceedings.mlr.press/v119/huang20h.html

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020a. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

Chengyue Jiang, Zijian Jin, and Kewei Tu. 2021. Neuralizing Regular Expressions for Slot Filling. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 9481–9498. https://doi.org/10.18653/v1/2021.emnlp-main.747

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (may 1997), 427–443. https://doi.org/10.1145/256167.256195

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

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts*

and Experiences (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

Bill Yuchen Lin, Ying Sheng, Nguyen Vo, and Sandeep Tata. 2020. FreeDOM: A Transferable Neural Architecture for Structured Information Extraction on Web Documents. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1092–1102. https://doi.org/10.1145/3394486.3403153

James Hiram Morris. 1968. *Lambda-calculus models of programming languages*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed on March 16, 2023.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Rajesh Parekh and Vasant Honavar. 1996. An incremental interactive algorithm for regular grammar inference. In *Grammatical Interference: Learning Syntax from Sentences*, Laurent Miclet and Colin de la Higuera (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–249.

Rajesh Parekh and Vasant Honavar. 2001. Learning DFA from Simple Examples. *Machine Learning* 44, 1 (01 Jul 2001), 9–35. https://doi.org/10.1023/A:1010822518073

Panupong Pasupat and Percy Liang. 2014. Zero-shot Entity Extraction from Web Pages. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Baltimore, Maryland, 391–401. https://doi.org/10.3115/v1/P14-1037

Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1470–1480. https://doi.org/10.3115/v1/P15-1142

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=KmtVD97J43e

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 158 (oct 2021), 29 pages. https://doi.org/10.1145/3485535

Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction Using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1967–1978. https://doi.org/10.1145/3318464.3380608

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 792–800.

R. L. Rivest and R. E. Schapire. 1989. Inference of Finite Automata Using Homing Sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing (STOC '89)*. ACM, 411–420.

Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 415, 13 pages.

Richard Shin and Benjamin Van Durme. 2022. Few-Shot Semantic Parsing with Language Models Trained on Code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Seattle, United States, 5417–5425. https://doi.org/10.18653/v1/2022.naacl-main.396

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. HOUDINI: Lifelong Learning as Program Synthesis. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/edc27f139c3b4e4bb29d1cdbc45663f9-Paper.pdf

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

Sen Wu, Luke Hsiao, Xiao Cheng, Braden Hancock, Theodoros Rekatsinas, Philip Levis, and Christopher Ré. 2018. Fonduer: Knowledge Base Construction from Richly Formatted Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1301–1316. https://doi.org/10.1145/3183713.3183729

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, Punta Cana, Dominican Republic, 1691–1704. https://doi.org/10.18653/v1/2021.findings-emnlp.146

Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL Translation through Compositional Lazy Inductive Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 112 (oct 2021), 26 pages. https://doi.org/10.1145/3485489

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating Code by Retrieving the Docs. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ZTCxT2t2Ru

Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. 2023. On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex. *arXiv* 2301.12868 (2023).

Received 2023-04-14; accepted 2023-08-27