Linguistic Tools for Managing Grammatical Domains (Work in Progress)

Anders Miltner, Computer Science Dept., UT Austin
Devon Loehr, Computer Science Dept., Princeton University
Arnold Mong, Computer Science Dept., Princeton University
Kathleen Fisher, Computer Science Dept., Tufts University
David Walker, Computer Science Dept., Princeton University

***Abstract*—Many common data types, such as dates, phone numbers, and addresses, have multiple textual representations. The possibility of varied representations creates ambiguities when parsing this data and can easily lead to bugs in what would otherwise be straightforward data processing applications. In this work-in-progress paper, we explore this problem and consider the design of linguistic tools to help manage it. More specifically, we introduce the idea of a *grammatical domain*—a set of related grammars—which we use to characterize the many possible representations of a date or phone number or other object of this sort. We also propose the design of a YACC-like language, which is capable of defining individual grammatical domains, such as dates, or larger document formats that contain such ambiguous elements. We illustrate our ideas via example, and sketch a continuing research agenda aimed at producing tools to help programmers process these ambiguous formats.**

I. Introduction

Data transformation, cleaning, and processing is a tedious and difficult task that stands in the way of getting important *information* out of the raw text all around us. A single data set may contain files encoding the same kinds of information in different formats; if data is drawn from multiple sources, the chances of formatting inconsistencies skyrocket. This lack of standardization can have real costs—if an ambiguous file is incorrectly parsed, it may lead to processing system failures, silent data corruption that pollutes critical data bases, or even security vulnerabilities [1].

As an example, consider the myriad possible representations of a date. Assuming a DD/MM/YY format when a MM/DD/YY format is used may lead to catastrophic errors. Phone numbers, addresses, times, names, postal codes, time ranges, abbreviations (such as for states or provinces), and other common data types suffer these problems.

More complex formats can suffer similar problems. For instance, one might expect a "comma-separated-value" (CSV) file to be formatted as a series of fields separated by commas, but it is often not. Tabs, vertical bars, semi-colons, spaces, or carats are all sometimes used instead of commas to separate fields. Some files use quotes to delimit strings and related symbols, while others use tildes or single quotes. And of course, some CSV files are "typed," with one column expected to contain integers, and perhaps strings in another.

As a result, the appropriate way to parse a CSV file can be ambiguous [2]; to avoid incorrectly interpreting it, one must determine the correct CSV grammar (or "dialect") to use for a given data set. In the case of CSVs, there are some tools to help users – for instance, Microsoft Excel provides a wizard that allows a user to select the kind of delimiter to use when importing a CSV file as a spreadsheet. Unfortunately, the process is a manual one. Python has libraries that implement "sniffers" to detect CSV dialects, and van den Burg *et al.* [2] built their own tool to analyze CSV files and infer the grammar best suited to parse it.

Another widely-used example domain suffering such problems is PDF. While PDF has been standardized, different tools implement different subsets or (or even supersets!) of the standard. Moreover, some of these PDF variants, and the tools used to process them, contain vulnerabilities [3], [4]. Hence, ambiguities in how to process PDF lead not just to bugs, but possibly to security vulnerabilities. The DARPA SafeDocs program [1] is currently exploring ways to define safe subsets of PDF to limit tool vulnerabilities.

In this paper, we introduce the idea of a *grammatical domain*, that is, a *set* of formal grammars that describe the many possible representations of a data type. For instance, the date grammatical domain would contain a grammar describing the DD/MM/YY format, another grammar describing the MM/DD/YY format, and many other grammars describing other formats (e.g., MM-DD-YY).

We propose the design of a new language for defining such domains, called Saggittarius. Such a language is a first step towards developing more robust data processing tools. Roughly speaking, SAGGITTARIUS may be viewed as an extension of a standard YACC-based parser generator. However, whereas YACC defines a single grammar, SAGGITTARIUS defines a *set* of possible grammars—*i.e.,* a grammatical domain. To do so, SAGGITTARIUS allows grammar engineers to specify certain grammatical productions as optional. Grammars within the domain are defined by the subset of optional productions that they include. SAGGITTARIUS also has features that allow grammar engineers to declare constraints that force certain combinations of productions to appear, or not appear, and hence provides fine control over the grammars in the grammatical domain in question.

Once a grammatical domain is defined in Saggittarius, it may be *applied* to example data. We note that the engineers *defining* the domain and those *using* the domain may (and frequently will) be different. We call the engineer who defines the domain the *grammar engineer* and the engineer who uses the grammar

the *instance engineer*. These two roles are separate because we expect heavy reuse of grammatical domains. For instance, the date domain need only be defined once by an expert. It can then be used countless times by instance engineers developing specific data processing tools that contain dates.

When an instance engineer uses the grammatical domain, a *grammar induction algorithm* will analyze example data and pick out the specific grammar from the domain to be applied in this instance. In other words, it will select the productions that allow the resulting grammar to parse all positive examples and none of the negative ones. Because more than one grammar from the domain may satisfy the provided examples, SAGGITTARIUS allows grammar engineers to provide functions to rank the generated grammars. While grammar engineers require sophisticated knowledge of a grammatical domain and the SAGGITTARIUS tool, instance engineers need only supply appropriate examples from the domain in question and then use the generated parser.

To summarize, in this short paper, we describe the idea of grammatical domains and propose a language, SAGGITTARIUS, for describing them. The following section illustrates the features of SAGGITTARIUS, by applying it to the description of two canonical domains: the domain of dates and the domain of CSV formats. Section III describes related work in the area and Section IV concludes.

II. MOTIVATING EXAMPLES

Grammatical domains appear in many different contexts. In this section, we show how to use SAGGITTARIUS to define two useful domains: the domain of calendar dates and the domain of comma-separated-value (CSV) formats.

A. Example 1: Calendar Dates

Recall that dates are formatted in many different ways, and so date parsers must be specialized to a particular data set. The many date formats form a natural grammatical domain, and different data sets adhere to different grammars within the domain.

SAGGITTARIUS programs specify grammatical domains through the use of *metagrammars* ($\mathcal{M}$), where a metagrammar is a set of *candidate productions* (a.k.a *candidate rules*) together with (a) constraints that limit which combinations of productions may appear, and (b) preferences that rank the grammars, for breaking ties when multiple grammars are applicable.

The simplest SAGGITTARIUS components specify productions using a YACC-like syntax with the form `N -> RHS`. Here, `N` is a non-terminal and `RHS` is a regular expression over terminals and non-terminals. For instance, to begin construction of our date grammatical domain, we can specify `Digit` and `Year` non-terminals as follows.

```
Digit -> ["0"-"9"].
Year -> Digit Digit
      | Digit Digit Digit Digit.
```

This first definition looks like a definition one might find in an ordinary grammar. It states that `Year` can have either two or four digits. The denotation of such a definition is a grammatical domain—in this case, a grammatical domain (a set) containing exactly one grammar.

SAGGITTARIUS is more interesting when one defines metagrammars that include optional productions. Optional productions are preceded by a "?" symbol. For instance, consider the following definition.

```
Digit -> ["0"-"9"].
Year -> ? Digit Digit
        ? Digit Digit Digit Digit.
```

The metagrammar above denotes a grammatical domain that includes four grammars:

1) one grammar in which Year has no productions,
2) two grammars in which Year has one production, and
3) one grammar in which Year has two productions.

To extract a single grammar from this set of four grammars, one supplies the SAGGITTARIUS grammar induction engine with positive and negative example data. If no grammar parses all the data as required, the grammar induction algorithm will return "no viable grammar."

Continuing, consider the following specification for days.

```
Day -> ? ["1" - "9"]
       ? "0" ["1" - "9"]
       | ["1" - "2"] Digit | "30" | "31".
```

This metagrammar includes grammars for days ranging from 1 to 31. It allows single digit days to be prefixed with a 0. However, it is natural to require grammars that parse either single-digit days or 0-prefixed-days, but not both. One way to specify such a constraint is as follows.

```
Day -> ? ["1" - "9"]
       ? "0" ["1" - "9"]
       | ["1" - "2"] Digit | "30" | "31".
constraint(|productions(Day)| = 4).
```

Here, the constraint specifies that the number of production rules for `Day` must be exactly 4. Since the last row is always included, exactly one of the ? production candidates can be in the solution grammar. Another option is to *name* productions and to use the names in constraints.

```
Day -> ? ["1" - "9"] as SDDays
       ? "0" ["1" - "9"] as TDDays
       | ["1" - "2"] Digit | "30" | "31".
constraint(SDDays + TDDays = 1).
```

Here, we have given names to each of the rule candidates which represent indicator variables in the constraint expression; they evaluate to 1 if the production is included in a given grammar, and 0 otherwise. In addition to defining constraints using arithmetic, one may use logical connectives. For instance, a constraint of the form `R1` $\Rightarrow$ `R2` will require `R2` to be included whenever `R1` is.

Figure 1 includes the rest of the (simplified) definition of the date format, adding definitions for separators, months, and

```
Sep -> ? ","  ? "/"  ? "-" .
constraint(|Production(Sep)| = 1)

Digit -> ["0"-"9"].

Year -> ? Digit Digit
        ? Digit Digit Digit Digit.
constraint(|Production(Year)| = 1)

Month ->  ? Digit
          ? "0" Digit
          | "10" | "11" | "12".
constraint(|Production(Month)| = 2)

Day -> ? ["1" - "9"]
       ? "0" ["1" - "9"]
       | ["1" - "2"] Digit | "30" | "31".
constraint(|Productions(Day)| = 2).

Date -> ? Day   Sep Month Sep Year
        ? Month Sep Day   Sep Year
        ? Year  Sep Month Sep Day
        ? Year  Sep Day   Sep Month.
constraint(|Productions(Date)| = 1).

S -> Date
```

Fig. 1. Calendar Dates Metagrammar

dates as a whole. Non-terminal `S` denotes the grammar start symbol. The use of constraints is common. For instance, notice the grammar engineer who designed this particular format allowed for the possibility of several different separators, but required a single separator to be used consistently throughout a format. Hence, while a date format may use "–" or "/" as a separator, it never uses both.

To extract a particular grammar from the domain, an instance engineer will supply positive and/or negative example data. For example, one could supply U.S.-style dates `12/11/72` and `01/10/72`, marking them as positive examples. Having done so, the SAGGITTARIUS grammar induction algorithm might generate the example grammar presented in Figure 2. However, there are other grammars in the domain that are also valid for this set of examples.

While one might worry that a naive instance engineer could supply insufficient data and thereby underconstrain the set of possible solution grammars, such problems could likely be mitigated through a well-designed user interface that informs a user when multiple solutions are possible and presents example data to the user, asking them to choose valid and invalid instances of the format.

B. Example 2: Mini-CSV

One challenge in specifying a CSV domain is that if we want the columns of the CSV format to be "typed" — one column must be integers, another strings or dates, for instance — we need to consider many, many potential grammar productions. To facilitate construction of such metagrammars succinctly, we allow grammar engineers to define indexed collections of productions. For instance, suppose we would like to specify

```
Sep -> "/".

Digit -> ["0"-"9"].

Year -> Digit Digit.

Month ->  "0" Digit | "10" | "11" | "12".

Day -> ["1" - "9"]
     | ["1" - "2"] Digit
     | "30" | "31".

Date -> Month Sep Day Sep Year.

S -> Date
```

Fig. 2. Date Solution Grammar

a spreadsheet with three columns (numbered 0-2) where each column can contain either a number or a string value. We might define the `i`th Cell in each row as follows.

```
Cell{i in [0,2]} -> ? Number ? String.
for (i in [0,2])
  constraint(|Productions(Cell[i])| = 1)
```

This declaration defines three nonterminals simultaneously: `Cell{0}`, `Cell{1}`, and `Cell{2}` and provides the same definition for each of them. However, since each of `Cell{0}`, `Cell{1}`, and `Cell{2}` are separate non-terminals, the underlying inference engine can specialize them independently based on the supplied data. For instance, `Cell{0}` could be a string and `Cell{1}` and `Cell{2}` might both be numbers.[1]. Constraints can refer to specific indexed non-terminals as shown.

While each `Cell` has the same definition, it is also possible to define collections of nonterminals with varying definitions through the use of conditionals. Below, we define `Row{i}`, a non-terminal for a row containing cells `Cell{0}` through `Cell{i}`. The use of normal context-free definitions allows `Row{i}` to refer to `Row{i-1}`. Notice that separators (`Sep`) are not indexed. We want one separator definition that is used consistently throughout the format, though we do not know which separator it will be.

```
Row{i in [0,2]} ->
  ? if (0 = i) then Cell{i}
  ? if (0 < i) then Row{i-1} Sep Cell{i}.

Sep -> ? "," ? "|" ? ";".
  constraint(|Productions(Sep)| = 1).
```

`Row{i}` represents a single row with `i` `Cells`. To create a table with many rows, we might write the following definition.

[1]Observant readers may worry that the characters "12" could be interpreted as either a number or a string if the definition of strings includes numbers. We will explain how to create preferences to disambiguate shortly.

```
S     -> Table.
Table -> MyRow ("\n" MyRow)*.
MyRow -> [? Row{i} for i in [0,2]].
  constraint (|Productions(MyRow)| = 1)
```

Here, we use the standard Kleene star to represent a table with an arbitrary number of rows. (We could equally well have written the usual recursive, context-free definition.). To force every row in the table to have the same shape, we demand that the kind of row used in the table (*i.e.*, `MyRow`) be exactly one of the `Row{i}` non-terminals. The notation "[? ... for i in range]" defines several possible productions, one for each value of `i`. In this case, it is a choice amongst the many possible lengths of row.

Figure 3 presents our progress so far on defining a metagrammar for simple CSV formats. At the top, we have "imported" a couple of useful non-terminal definitions—definitions for `String` and `Number`. Users can write such definitions from scratch, but we have developed a modest library of them to facilitate quick construction of parsers for ad hoc data formats.

```
1  import String, Number

3  S -> Table.

5  Table -> MyRow ("\n" MyRow)*.

7  MyRow -> [? Row{i} for i in [0,2]].
8    constraint(|Productions(MyRow)| = 1)

10 Sep -> ? ","  ? "|" ? ";".
11   constraint(|Productions(Sep)| = 1).

13 Row{i in [0,2]} ->
14   ? if (0 = i) then Cell{i}
15   ? if (0 < i) then Row{i-1} Sep Cell{i}.

17 Cell{i in [0,2]} ->
18   ? Number
19   ? String.
20 for(i in [0,2])
21   constraint(|Productions(Cell[i])| = 1)
```

Fig. 3. CSV Metagrammar

C. Ranking Grammars

Consider the following example data.

```
0,1,Hello world!
1,2,Programming
0,3,rocks!
```

A human would probably choose the column types to be `Number`, `Number`, `String`. However, if `Numbers` can be `Strings` then the column types could be `String`, `String`, `String`. Without guidance, an algorithm will not know how to choose between potential grammars.

SAGGITTARIUS allows users to steer the underlying grammar induction algorithm towards the grammar of choice by expressing preferences for one grammar over another. Such preferences are expressed using `prefer` clauses, which have a similar syntax to `constraint` clauses. Such clauses assign

```
1  Cell{i in [0,2]} ->
2    ? Number as Num{i}
3    ? String as Str{i}.
4  for(i in [0,infty))
5    constraint (|productions(Cell[i])| = 1).

7  prefer{i in [0,2]} 2.0 Num{i}.
8  prefer{i in [0,2]} 1.0 Str{i}.
```

Fig. 4. A Metagrammar with Preferences

floating point numbers to boolean formulas. The ranking of a synthesized grammar is the sum of all satisfied boolean formulas. SAGGITTARIUS produces grammars with a maximal ranking. Figure 4 illustrates the use of preferences to force CSV formats to infer `Number` cells when they can and `String` cells otherwise.

D. Selecting Grammars

After a grammar engineer has specified the grammatical domain, the next task is for the instance engineer to specialize it to a testbed of positive and negative examples. But after the instance engineer has provided these examples, how does Saggittarius induce which grammar in the domain is the correct one?

This is done through a combination of full parsing and MaxSMT. Our full parsing algorithm parses a forest of all possible parse trees for the examples, assuming every rule were included. These parse trees are then, along with the constraints and preferences provided by the user, encoded as logical formulas, and sent to a MaxSMT solver. This solver then identifies a subset of the grammar productions that can be used to parse the positive examples and omit the negative examples, while optimizing for the user-provided preferences.

E. Existential Quantification

So far, our CSV example can handle files with at most three columns. Of course, we could have chosen a larger bound, but in general a CSV file can have an arbitrary number of columns, which we cannot know in advance. To fully specify the CSV metagrammar, we need a way to allow rules with an arbitrary number of productions.

To address this, we plan to add existentially-quantified variables to represent unknown values. Figure 5 demonstrates how we might use existentials to define the rows of a CSV file.

Unfortunately, inferring grammars with existentially quantified variables is not guaranteed to terminate. We can attempt to find solutions by iteratively setting rowlen to a constant number (resulting in a finite metagrammar), and increasing that constant if our algorithm fails. However, if no solution grammar exists, this process may continue indefinitely.

Existential variables also allow us to re-think our implementations of prior metagrammars. For example, previously we enforced types by having one `Cell` rule for each column. Instead we could have one `Cell` rule for each *type*, together with a *local* existential, as shown in Figure 6.

```
Row{len:nat} ->
  | if (len = 0) Cell
  | if (len > 0) Cell Sep Row{len-1}

Rows{len:nat} ->
  | Row{len}
  | Row{len} Newline Rows{len}

exists rowlen:nat

S -> Rows{rowlen}
```

Fig. 5. CSV metagrammar snippet with arbitrary length rows

```
Cell{type:[0,1]} ->
  | if (type = 0) then Number
  | if (type = 1) then String

Row{len:nat} ->
  exists type:[0,1].
  | if (len = 0) Cell{type}
  | if (len > 0) Cell{type} Sep Row{len-1}
```

Fig. 6. CSV metagrammar snippet using local existentials

We interpret the local `exists` to mean that a different type exists for each different `len` argument, effectively enforcing that each column is parsed using the same rule. This format has a number of benefits; for one, we have reduced an arbitrary number of `Cell` rules to just two, regardless of the number of columns. Furthermore, local exists also have the potential to replace constraints entirely – for example, notice that a constraint that `Cell` has only one production would be redundant. This has the potential to simplify our internal representation of the metagrammar, although we expect to keep constraints in the surface language as a convenience for users.

III. Related Work

Grammar induction: Grammar induction traces back to at least the 60s when Gold [5] began studying models for language learning and their properties. Later, Angluin [6], [7] developed her famous L* algorithm for learning regular languages. As mentioned earlier, however, such algorithms, on their own, often require large numbers of examples, even to synthesize simple regular expressions. More recently, FlashProfile [8] has shown that regular-expression-like *patterns* can be learned from positive examples, by (1) clustering by syntactic similarity, and (2) inducing programs for given clusters. Inference of context-free grammars is considerably more difficult than inference of regular expressions and patterns, and results are limited, but it has been tackled, for instance, by Stolcke and Omohundro [9], who use probabilistic techniques to infer grammars. Fisher *et al.* [10] explored inference of grammars for "ad hoc" data, such as system logs, in the context of the PADS project [11]. Lee [12] developed more efficient search strategies for regular languages in the context of a tool for teaching automata theory. Both these latter tools tackled restricted kinds of grammars. Scaling to complex formats using few examples remains a challenge in either case. The GLADE [13] tool is a more recent approach to synthesizing grammars. Similarly to L*, GLADE uses an active learning algorithm, and generalizes to full context-free grammars, rather than merely regular expressions, while requiring relatively fewer membership queries to hone-in on the desired grammar. The key contributions of this paper are largely orthogonal to these advances in grammar induction algorithms over the years. In particular, we introduce the idea of "grammatical domains," and a novel language for defining meta-grammars, to restrict the set of grammars under consideration during the induction process; doing so has the potential to improve the performance of almost any grammar induction algorithm.

Grammatical inference becomes more tractable when one can introduce bias or constraints—meta-grammars are one way to introduce such bias but there are others. For instance, Chen *et al.* [14] use a combination of examples and natural language to speed inference of a constrained set of regular expressions. Internally, their system generates an "h-sketch" as an intermediate result. These h-sketches are partially-defined regular expressions that may include holes for unknown regular expressions. Such h-sketches play a similar role to our meta-grammars: They denote sets of possible regular expressions and they constrain the search space for grammatical inference. However, our language is an extension of YACC and is designed for humans rather than being a regular-expression-based intermediate language. We also introduce the idea of "grammatical domains," and provide natural examples such as date and phone number domains, that may be reused across data sets; each h-sketch is an intermediate representation used once inside a compiler pipeline.

Related to the notion of grammatical inference is that of expression *repair*. RFIXER [15] uses positive and negative examples to fix erroneous regular expressions. Both RFIXER and Saggittarius use similar algorithms for finding grammars that ensure positive examples are in the generated language, and negative examples are not. Both of these tools encode these constraints as MaxSMT formulas to ensure the generated grammars are optimal. Because RFixer does not have a metagrammar to orient the search, their constraints can only be used to find character sets that distinguish between the grammars. Saggittarius permits any constraints that can be expressed in propositional logic, and the constraints can be over arbitrary productions, not merely character set choices. In effect, one could see their algorithm as an instance of our algorithm, where the meta-grammar they are using is one of character sets.

Syntax-guided Program Synthesis: . Our work was inspired by the progress on syntax-guided program synthesis over the past decade or so [16], [17], [18], [19]. Much of that work has focused on data transformations, including spreadsheet manipulation [20], [21], [22], string transformations [23], [24], [25], and information extraction [26], [27]. Such problems have much in common with our work, but they have typically been set up as searches over a space of program transformation

operations rather than searches over collections of context-free grammar rules. Particularly inspiring for our work was the development of FlashMeta [28] and Prose [29]. These systems are "meta" program synthesis engines—they help engineers design program synthesis tools for different domain-specific languages. Similarly, SAGGITTARIUS is a "meta" framework for syntax-guided grammar induction, helping users perform grammar induction in domain-specific contexts. Of course, SAGGITTARIUS, FlashMeta and Prose differ greatly when it comes to specifics of their language/system designs and the underlying search algorithms implemented.

Logic Program Synthesis: We were also inspired by work on Inductive Logic Programming [30], and Logic Program Synthesis [31], [32]. Parsing with context-free grammars is a special case of logic programming so it was natural to investigate whether inductive logic programming algorithms would work well here. ProSynth [31] is a state-of-the-art algorithm in this field so we experimented with it as a tool for grammatical inference. However, we found our custom algorithm almost always outperformed ProSynth on grammatical inference tasks.

IV. Conclusion and Continuing Research

Grammatical domains are sets of related grammars. Such domains appear whenever a common datatype like a date or a phone number has multiple textual representations. They also often appear when data sets are communicated via ASCII text files, as is the case for CSV files. In this paper, we introduce the concept of grammatical domains and provide a variety of examples of such domains.

We also design a language, called SAGGITTARIUS, for specifying meta-grammars, which define grammatical domains. SAGGITTARIUS includes features for defining finite and infinite sets of candidate productions, for constraining the candidate productions that may and may not appear, and for ranking the generated grammars.

The next steps in our research agenda are to experiment with grammar induction algorithms capable of inferring the intended grammar from a grammatical domain when supplied with example data. Our initial investigation into this topic suggests it is possible generate SMT constraints that describe the necessary productions to parse example formula and then to use a MaxSMT solver to find the highest ranked solution. Still, more research and experimentation is required.

We also look forward to extending SAGGITTARIUS with semantic actions, which are needed to turn SAGGITTARIUS into a full blown parser generator system like YACC.

Acknowledgements: This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under the SafeDocs program. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] https://www.darpa.mil/program/safe-documents, 2020.

[2] G. J. J. van den Burg, A. Nazábal, and C. Sutton, "Wrangling messy csv files by detecting row and type patterns," Nov. 2018, arXiv:1811.11242v1. [Online]. Available: https://arxiv.org/pdf/1811.11242.pdf

[3] C. Carmony, X. Hu, H. Yin, A. V. Bhaskar, and M. Zhang, "Extract me if you can: Abusing PDF parsers in malware detectors," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/extract-me-if-you-can-abusing-pdf-parsers-malware-detectors.pdf

[4] K. Liu, "Dig into the attack surface of pdf and gain 100+ cves in 1 year," https://www.blackhat.com/docs/asia-17/materials/asia-17-Liu-Dig-Into-The-Attack-Surface-Of-PDF-And-Gain-100-CVEs-In-1-Year-wp.pdf, 2017.

[5] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[6] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, vol. 39, no. 3, pp. 337–350, 1978.

[7] ——, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[8] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. Millstein, "Flashprofile: A framework for synthesizing data profiles," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276520

[9] A. Stolcke and S. M. Omohundro, "Inducing probabilistic grammars by bayesian model merging," *CoRR*, vol. abs/cmp-lg/9409010, 1994.

[10] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: Fully automatic tool generation from ad hoc data," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 421–434.

[11] K. Fisher and D. Walker, "The pads project: An overview," in *Proceedings of the 14th International Conference on Database Theory*, ser. ICDT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 11–17.

[12] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," in *ACM SIGPLAN International Conference on Generative Programming*, 2016, pp. 70–80.

[13] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *SIGPLAN Not.*, vol. 52, no. 6, p. 95–110, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062349

[14] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 487–582.

[15] R. Pan, Q. Hu, G. Xu, and L. D'Antoni, "Automatic repair of regular expressions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360565

[16] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," *SIGPLAN Not.*, vol. 40, no. 6, p. 281–294, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065045

[17] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, p. 404–415, Oct. 2006. [Online]. Available: https://doi.org/10.1145/1168919.1168907

[18] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 1–8.

[19] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, p. 84–93, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3208071

[20] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of P rogramming Languages*, ser. POPL '11. ACM, 2011.

[21] X. Wang, I. Dillig, and R. Singh, "Synthesis of data completion scripts using finite tree automata," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133886

[22] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn, "Flashrelate: Extracting relational data from semi-structured spreadsheets using examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming*

Language Design and Implementation, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 218–228.

[23] A. Miltner, S. Maina, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, "Synthesizing symmetric lenses," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341699

[24] A. Miltner, K. Fisher, B. C. Pierce, D. Walker, and S. ve Zdancewic, "Synthesizing bijective lenses," in *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pr ogramming Languages*, ser. POPL 2018, 2018.

[25] X. Wang, I. Dillig, and R. Singh, "Program synthesis using abstraction refinement," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158151

[26] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," *ACM SIGPLAN Notices*, vol. 49, 06 2014.

[27] M. Raza and S. Gulwani, "Automated data extraction using predictive program synthesis," in *AAAI*, 2017, pp. 882–890.

[28] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 107–126.

[29] M. Research, "Prose," https://www.microsoft.com/en-us/research/group/prose/, 2020.

[30] L. De Raedt, "Logical and relational learning," in *Advances in Artificial Intelligence - SBIA 2008*, G. Zaverucha and A. L. da Costa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–1.

[31] M. Raghothaman, J. Mendelson, D. Zhao, M. Naik, and B. Scholz, "Provenance-guided synthesis of datalog programs," 2020. [Online]. Available: https://doi.org/10.1145/3371130

[32] X. Si, M. Raghothaman, K. Heo, and M. Naik, "Synthesizing datalog programs using numerical relaxation," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 6117–6124. [Online]. Available: https://doi.org/10.24963/ijcai.2019/847