Grammatical Domains and Syntax-guided Grammar Induction

ANDERS MILTNER, Princeton University, USA
ARNOLD MONG, Princeton University, USA
KATHLEEN FISHER, Tufts University, USA
DAVID WALKER, Princeton University, USA

Grammatical domains are sets of related grammars. Such domains appear naturally whenever a common datatype like a date, a phone number or an address has multiple textual representations. While the grammar induction problem may be intractable in general, when specialized to such grammatical domains it is often not only feasible but efficient. In this paper, we introduce the concept of grammatical domains and propose to learn grammars within such domains using *Syntax-guided Grammar Induction* (SGI), a refinement of the traditional grammar induction problem that includes a syntactic specification of the grammars to search through. To support the definition of grammatical domains, we design Saggitarius, the first user-facing meta-language for specifying grammatical domains. The Saggitarius system comes equipped with an algorithm for solving SGI problems based on encoding them as MaxSMT instances. We also define a benchmark suite for SGI and evaluate the effectiveness of the algorithm on our benchmark suite, showing it typically returns a satisfying grammar within a few seconds. We describe a case study in which we use Saggitarius to detect classes of XML documents and find that Saggitarius metagrammars can be used to tame a highly ambiguous search space. Finally, we perform a second case study in which we use Saggitarius to detect CSV dialects. Our general purpose tool infers grammars for 84% of the files in the case study within the specified timeout. On these grammars, it has comparable accuracy to custom-built dialect detection tools.

1 INTRODUCTION

Grammar induction is the long-studied problem of inferring a grammar from positive and negative example data [3, 4, 13, 16, 17, 31, 38, 46]. Despite its obvious value, and much research effort over the years, instances of the problem often remain intractable, requiring either a great deal of example data or vast computational resources to solve. Indeed, even the restricted case of regular expression inference is a challenge: Chen [8] observed that Angluin's classic L* algorithm makes 679 queries just to infer the simple regular expression [a-zA-Z]+, and Lee [24] shows that the search space for regular expressions grows at a rate of c^{2^d-1} where d is the depth of the regular expression and c is the number of regular expression operators (c is 7 when working with a binary alphabet). While recent research [8, 24] has taken impressive steps forward in the special case of regular languages, scaling to larger data formats remains a challenge. For instance, AlphaRegex still classifies some regular expressions over a binary alphabet with 10-20 symbols as "hard" [24, Table 3]. Scaling to richer classes of grammars, such as the context-free languages, is even more daunting.

One "solution" to this problem is to give up: The grammar induction problem cannot be solved accurately in general—there are simply too many degrees of freedom. But after having given up, one must ask: What's next?

As with any artificial intelligence problem, one way to give shape to, or cut down, the search space, and thereby render an intractable problem tractable, is to associate some priors, constraints, or background knowledge with instances of the problem. For example, over the last decade or so, great progress has been made in the closely related problem of program synthesis not by tackling the problem in general, but by picking out key subdomains in which to operate. Typically, this

Authors' addresses: Anders Miltner, Princeton University, Princeton, NJ, USA, amiltner@cs.princeton.edu; Arnold Mong, Princeton University, Princeton, NJ, USA, among@princeton.edu; Kathleen Fisher, Tufts University, Medford, MA, USA, kfisher@cs.tufts.edu; David Walker, Princeton University, Princeton, NJ, USA, dpw@cs.princeton.edu.

selection works by defining a restrictive domain-specific language (DSL), specializing the ranking functions for the programs in the DSL, and then learning a ranked list of DSL programs from examples or other kinds of specifications. In this way, it has been possible to build extremely effective program synthesizers for spreadsheet transformations [22], database queries [49], tree transformations [50] and lenses [26–28] to name just a few. Program induction of this form is often referred to as *Syntax-guided Program Synthesis* (SyGuS), because one of the key elements of such systems is the fact that the syntax of programs is carefully constrained, via the DSL, ahead of time. Now, there are meta-languages like Prose [37] to help experts build syntax-guided program synthesis systems.

In this paper, we investigate whether similar techniques may be applied productively to the problem of grammar induction. In doing so, the natural first question was whether there were any *grammatical domains* akin to the *programming domains* where SyGuS has been effective. Such grammatical domains should include a set of related grammars, just as a programming domain contains a set of related programs. Moreover, programmers should easily be able to identify when the grammar they need falls within the given domain so they are able to select the tool or library to apply to their problem.

Grammatical Domains. A clear example of an interesting grammatical domain is the set of "comma-separated-value" formats, the *CSV domain*. While one mostly thinks of comma-separated value formats as simple tables, separated by, well, commas, there is actually great variation in these formats [44]. Tabs, vertical bars, semi-colons, spaces, or carats are all sometimes used rather than 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 integers required to appear in one column and perhaps strings in another. One of the consequences of this mess is that the appropriate way to parse a CSV file can be ambiguous [44]. Hence, to avoid incorrectly interpreting a given CSV file, one must determine the correct CSV grammar or "dialect" to use for a given data set. There are some tools to help users with this particular special case. For instance, Microsoft Excel provides a wizard that allows a user to select the kind of delimiter to use to import 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.* [44] built their own tool to analyze CSV files and infer the grammar best suited to parse it.

CSV formats are just the tip of the iceberg: there are many other grammatical domains that arise in practice. Some are similar to CSV formats in that they provide a textual representation of a large, structured data set. Examples include the set of possible JSON formats, XML formats, or S-expression formats. In each case, there is a general specification of the layout, but the general case is often specialized in particular applications, to, for instance, constrain the types of values or the schema that may appear.

Another sort of grammatical domain is a domain associated with a data type that, over time, has come to have multiple different textual representations. Here, dates are a good example. Dates may be formatted using MM/DD/YY strings or DD/MM/YY strings, as well as in a myriad of other ways. Since these different formats are ambiguous, it is necessary to identify and deploy the specific format used in a particular data set. Moreover, since dates often appear nested within other formats, those formats are themselves ambiguous until the internal date formats are properly determined. Other examples of data types with a variety of textual representations or specializations include phone numbers, postal codes, street addresses, the names of states or provinces that have multiple common abbreviations, timezones, times, date ranges, *etc.* Any kind of data whose representation needs to be adjusted to handle internationalization is a likely candidate for a grammatical domain.

Finally, this work is partly movtivated by the US DARPA SafeDocs program [9], which is currently exploring the *PDF grammatical domain*: While PDF has been standardized, different tools implement subsets or supersets of the standard. Moreover, some of these PDF variants, and the tools used to process them, contain vulnerabilities [7, 25]. Consequently, SafeDocs has a number of teams investigating ways to define safe subsets of PDF.

We collect the various grammatical domains we have studied into a benchmark suite described in §6.

Syntax-guided Grammar Induction. A domain-specific grammar induction problem is specified via a set of grammars (the domain), a set of positive examples and a set of negative examples. A solution to such a problem is a grammar drawn from the domain that parses all the positive examples and none of the negative examples. A natural way to specify such a domain is to constrain the syntax of the grammars that may be chosen, by, for instance, specifying the candidate productions available. We call problems specified this way *Syntax-guided Grammar Induction (SGI) Problems*.

In SGI, or in domain-specific grammar induction more broadly, there are two types of user. The first kind of user, the *grammar engineers*, are responsible for defining grammatical domains. The second, the *instance engineers*, infer grammars from within the defined grammatical domains by supplying positive and negative examples. Ideally, the domains produced by grammar engineers are heavily reused, making their investment of time and energy well spent. For instance, the domain of dates can be reused in the synthesis of any grammar for any data containing a date. Likewise, there are many many CSV files, meaning such a domain, once defined, can be used by many instance engineers who need only supply example data and need not possess the skills of grammar engineers.

Saggitarius. We designed and implemented Saggitarius, the first user-facing language and system to provide support for defining syntax-guided grammar induction problems. Roughly speaking, Saggitarius may be viewed as an extension of a standard YACC-based parser generator. However, whereas YACC defines a single grammar, Saggitarius defines a *set* of possible grammars—*i.e.,* a grammatical domain. To do so, Saggitarius allows grammar engineers to specify certain grammatical productions as optional, much like a SyGuS language, such as Sketch [2, 40, 41], declares certain subexpressions of a program optional. Saggitarius 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.

The Saggitarius grammar induction algorithm uses both the definition of the grammatical domain and the supplied examples to select 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, Saggitarius allows grammar engineers to provide functions to rank the generated grammars. Grammar engineers play a role similar to designers of a domain-specific language in syntax-guided synthesis; instance engineers play a role similar to users of synthesized functions. While grammar engineers require sophisticated knowledge of a grammatical domain and the Saggitarius tool, instance engineers need only supply appropriate examples from the domain in question and then use the generated parser.

Contributions.

- We introduce the concepts of *grammatical domains* and the *Syntax-Guided Grammar Induction (SGI) Problem*.
- We design a language, Saggitarius, for specifying grammatical domains and implement an SGI algorithm for inferring grammars within defined domains.

- We demonstrate that a variety of grammatical domains exist in the wild and use them to create the first SGI benchmark suite.
- We evaluate the performance of Saggitarius on our benchmark suite, showing that for many domains, it returns a grammar satisfying the example constraints within a few seconds.
- We describe a case study in which we develop an XML metagrammar. In it, we show how Saggitarius's programmatic control over the search space can be used to speed up synthesis times for challenging tasks.
- We describe a second case study in which we test the use of Saggitarius on the problem of dialect detection for CSV files. Our general purpose tool infers grammars for 84% of the files in the case study. On these grammars, it has comparable accuracy to custom-built dialect detection tools [45].

2 MOTIVATING EXAMPLES

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

2.1 Example 1: Calendar Dates

Dates are formatted in many different ways. Because the various formats are ambiguous (causing confusion as to whether one is reading a MM/DD or DD/MM format, for instance), date parsers must be specialized to a particular data set. Said another way, dates formats form a natural grammatical domain and different data sets adhere to different grammars within the domain.

Saggitarius programs specify grammatical domains through the use of *metagrammars* ($\mathcal{M}$), where a metagrammar is a set of *candidate productions* (*aka*, *candidate rules*) together with some constraints that limit the sets of productions that may appear in the grammars belonging to the domain and some preferences that describe what an optimal grammar looks like.

The simplest Saggitarius 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.

Saggitarius 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 Saggitarius 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 -> 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)| = 2).
```

Here, the constraint specifies that the number of production rules for `Day` can only be 2. 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 SingleDigitDays
       ? "0" ["1" - "9"] as ZeroPrefixDays
       | ["1" - "2"] Digit | "30" | "31".
constraint(SingleDigitDays + ZeroPrefixDays = 1).
```

Here, we have given names to each of the rule candidates which represent indicator variables in the constraint expression; the indicator variable `SingleDigitDays` evaluates to 0 if a given grammar does not use that production and 1 if it does. In addition to defining constraints using arithmetic, one may use logical connectives. For instance, a constraint of the form `R1` $\implies$ `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 (`Sep`), months (`Month`), and dates as a whole (`Date`). 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 `12/31/72` and `01/01/72`, marking them as positive examples. Having done so, the Saggitarius 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. By adding in the preference:

```
avoid 1.0 productions(*).
```

one can specify that a solution grammar should have a minimal number of productions. In this case, the solution presented in Figure 2 is unique. If one preferred European dates, one could supply different example data and extract a different grammar from the domain.

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.

```
Sep ->  exactly 1 of {? ","  ? "/"  ? "-" }.

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

```
Sep -> "/".

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

Year -> Digit Digit.

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

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

Date -> Day Sep Month Sep Year.

S -> Date
```

Fig. 2. Date Solution Grammar

2.2 Example 2: CSV

Dates, phone numbers, addresses and the like are simple data types with many textual representations. Saggitarius is also capable of specifying domains that contain larger aggregate data types. Here, the domain of comma-separated-value formats is a good example.

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 a spreadsheet with three columns (numbered 0-2) and each column can contain either a number or a string value. We might define the ith 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.

The domain of 3-column CSVs is likely a rare one! Fortunately, users can also declare collections of indexed non-terminals with unbounded size.

```
Cell{i in [0,infty)} -> ...
```

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,infty)} ->
  ? 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` `Cell`s. To create a table with many rows, we might write the following definition.

```
S     -> Table.
Table -> MyRow ("\n" MyRow)*.
MyRow -> ??{i in [0,infty)} (? Row{i}).
  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. Here, the notation "??" defines a "big option," *i.e.,* a choice amongst many alternatives. In this case, it is a choice amongst amongst the many possible `Row{i}` alternatives.

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

[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 momentarily.

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 -> ??{i in [0,infty)} (Row{i}).

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

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

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

Fig. 3. CSV Metagrammar, Version 1

2.3 Ranking Grammars

Consider the following example data.

```
0,1,15,Hello world!
1,2,23,Programming
0,3,-2,rocks!
```

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

Saggitarius 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 floating point numbers to boolean formulas. The ranking of a synthesized grammar is the sum of all satisfied boolean formulas. Saggitarius 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.

3 THE SYNTAX-GUIDED GRAMMAR INDUCTION PROBLEM

A context free grammar $G = (S, \mathcal{I}, \Sigma, R)$ with start symbol S, non-terminals $\mathcal{I}$, terminals Σ, and productions (also called rules) R is defined in the usual way. A Metagrammar $\mathcal{M} = (S, \mathcal{I}, \Sigma, R, f, h)$ includes the following components.

- S, the grammar start symbol
- $\mathcal{I}$, the set of grammar non-terminals

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

prefer{i in [1,infty)} 2.0 Num{i}.
prefer{i in [1,infty)} 1.0 Str{i}.
```

Fig. 4. A Metagrammar with Preferences

- Σ, the set of grammar terminals
- R, the finite set of possible candidate productions
- Boolean-valued constraint function f: $2^R \rightarrow bool$
- Real-valued preference function h: $2^R \rightarrow \mathbb{R}$

We say that a grammar G belongs to $\mathcal{M}$ (written $G \in \mathcal{M}$) when the start symbol of G is the start symbol of $\mathcal{M}$, the terminal and non-terminal symbols are the same in G and $\mathcal{M}$, the productions of G are a subset of the productions of M, and $f(G.R)$ is true (where $G.R$ are the productions of G).

An instance of the syntax-guided grammar induction problem is specified via a triple ($\mathcal{M}$, $Ex+$, $Ex-$) where $\mathcal{M}$ is a metagrammar and $Ex+$ and $Ex-$ are positive and negative examples respectively. The solution to an instance of the syntax-guided grammar induction problem is a grammar $G \in \mathcal{M}$ that contains all of the positive examples, none of the negative examples and has maximal ranking. In other words, for all $G' \in \mathcal{M}$, $h(G.R) >= h(G'.R)$.

4 SAGGITARIUS **CORE CALCULUS**

Saggitarius provides users with a convenient syntax for writing metagrammars (MG) that consists of a sequence of statements, where each statement can be either a production, a constraint, or a preference. The formal syntax follows.

$$
\begin{array}{rl}
MG ::= & prod.\ MG \\
& \mid \text{const } c.\ MG \\
& \mid \text{prefer } pref.\ MG \\
& \mid \epsilon \\
prod ::= & i \rightarrow p \\
c ::= & i \rightarrow p \quad \mid \neg c \\
& \mid c_1 \wedge c_2 \quad \mid c_1 \vee c_2 \\
pref ::= & c\ r
\end{array}
$$

Productions in a Saggitarius metagrammar are of the form $i \rightarrow p$, where i is a nonterminal, and p is a sequence of terminals and non-terminals.

Constraints are boolean formulas, where the predicates are productions. Any accepting grammar must satisfy the boolean formula, where included productions are interpreted as *true* and productions not included are interpreted as *false*.

Preferences consist of a boolean formula c and a real value r. If the boolean formula c is satisfied by a given grammar, the reward of that grammar is increased by r.

We assume there is some designated set of terminals, non-terminals and start symbol. The semantics of a Saggitarius metagrammar is given by a function $[[\cdot]]$ that maps a metagrammar MG to a triple (R, f, h) of productions, constraint function and preference function. The function $C[[c]]$ maps a constraint c to a function f from grammars to type $bool$. The function $\mathcal{P}[[pref]]$

maps a preference *pref* to a function from grammars to $\mathbb{R}$. We formally define these three functions below.

$$
\begin{aligned}
[[prod.MG]] &= (R \cup \{prod\}, f, h) \\
&\quad \text{where } [[MG]] = (R, f, h) \\
[[\text{const } c.MG]] &= (R, f, h) \\
&\quad \text{where } [[MG]] = (R, f_1, h) \\
&\quad \text{and } C[[c]] = f_2 \\
&\quad \text{and } f(x) = f_1(x) \wedge f_2(x) \\
[[\text{prefer } pref.MG]] &= (R, f, h) \\
&\quad \text{where } [[MG]] = (R, f, h_1) \\
&\quad \text{and } \mathcal{P}[[pref]] = h_2 \\
&\quad \text{and } h(x) = h_1(x) + h_2(x) \\
[[\epsilon]] &= (\{\}, f, h) \\
&\quad \text{where } f(x) = true \\
&\quad \text{and } h(x) = 0 \\
C[[i \rightarrow p]] &= f \text{ where } f(x) = i \rightarrow p \in x \\
C[[\neg c]] &= f \text{ where } f(x) = \neg(C[[c]](x)) \\
C[[c_1 \wedge c_2]] &= f \text{ where } f(x) = C[[c_1]](x) \wedge C[[c_2]](x) \\
C[[c_1 \vee c_2]] &= f \text{ where } f(x) = C[[c_1]](x) \vee C[[c_2]](x) \\
\mathcal{P}[[c\ r]] &= f \text{ where } f(x) = \text{if } C[[c]](x) \text{ then } r \text{ else } 0
\end{aligned}
$$

4.1 Syntactic Sugar by Example

Saggitarius includes a number of other features to enable rapid development of complex meta-grammars that do not appear in the core calculus. Except in the case of infinite rule spaces, these extensions are simple syntactic sugar. We illustrate how these constructs are compiled to the core calculus informally by example below.

? Choices. Using the ? combinator prior to a production body signifies a rule is optional. A single production may also include a series of optional bodies. For instance, the rules

```
A -> ? B ? "c".
B -> "b".
```

compile to

```
A -> B.
A -> "c".
B -> "b".
const B -> "b".
```

The constraint guarantees the production for B must be included; the lack of constraints for A's productions indicate they may or may not be included.

Regular Expression Productions. We compile regular expression syntax to context-free grammar rules in the usual way, by introducing new nonterminals and rules as appropriate. Such collections of such rules can be included or excluded as a group using constraints. For example, `A -> ?"b"*` may be compiled as follows, where Temp is a fresh variable.

```
A -> Temp as r1.
Temp -> "" as r2.
Temp -> "b"Temp as r3.
const r1 <=> r2 <=> r3.
```

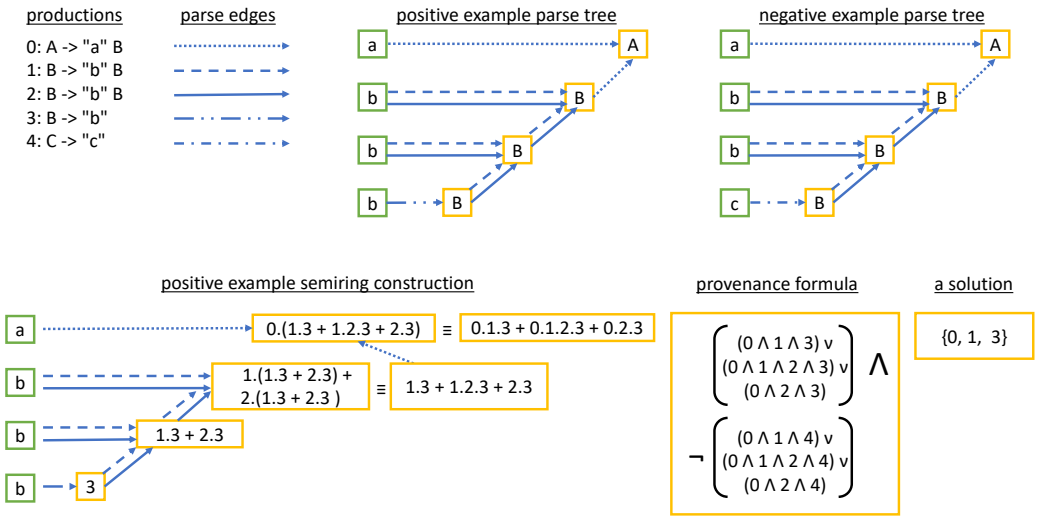

Fig. 5. Semiring parsing and provenance formula construction

Rules with finite index sets. Indexed rule sets with finite ranges are simply expanded at compile time. Hence `Cell{i in [0,2]} -> ...` is expanded to definitions for three non-terminals: `Cell_0`, `Cell_1`, and `Cell_2`. Computations that depend upon the indices, such as conditions, are also resolved at compile time.

Rules with infinite index sets. When a Saggitarius metagrammar contains rules with an infinite index set such as `Cell{i in [0,infty)} -> ...`, it is not compiled to a single syntax-guided grammar induction problem. Rather, it is compiled to a series of SGI problems, each with finite size. The system increases the size of the problem until it is able to find a solution. For example, `Cell{i in [0,infty)}` is first compiled to `Cell{i in [0,20)}`. If no solution is found, it is then compiled to `Cell{i in [0,40)}`, and so on. Termination is not guaranteed for grammars with infinite indexing – if there is no solution Saggitarius will search for one indefinitely.

5 GRAMMAR INDUCTION ALGORITHM

A syntax-guided grammar induction algorithm must determine the set of candidate parsing rules that parse the positive examples, do not parse the negative examples, satisfy all constraints and maximize the preference score. We have experimented with several possible algorithms that interleave parsing and constraint solving in different ways (See §6 for an evaluation of the trade-offs); we outline the most successful algorithm here.

In a nutshell, our algorithm parses all examples simultaneously and uses the results to generate a logical formula that describes the rules required to successfully parse all the positive examples and none of the negative examples. This formula is conjoined with the user-specified constraints and shipped to Z3 [10], which uses its MaxSMT algorithms to find a preference-optimal solution. This algorithm is formalized in Figure 6.

Our parser must be able to process the highly ambiguous metagrammars generated by Saggitarius. For this task, we chose to implement a modified Earley algorithm [12, 43], though other general context-free parsing algorithms, such as a GLR algorithm [30], would also have been suitable starting points. The key modifications to the algorithm come in how we process the grammatical

```
    Dependencies:
(1) Provenance Parser P : Productions → Σ* → Constraint
(2) SMT-Solver Z: Constraint → Preference → Grammar option
    Input: (M, Ex+, Ex−)
    Output: A Grammar G ∈ M that parses Ex+, and does not parse Ex−

(R, f, h) := [[M]]
φ := f
for each negative example e− ∈ Ex−:
    φ := φ ∧ ¬(P(M, e−))
for each positive example e+ ∈ Ex+:
    φ := φ ∧ ¬(P(M, e+))
match Z(φ, h) with
    | None → return "No viable grammar"
    | Some G → return G
```

Fig. 6. Saggitarius's grammar inference algorithm. This algorithm relies on a "Full Parser," which parses a string according to some grammar, then outputs a formula that represents what rules must be included to get a viable parse; and a SMT-Solver, which can find an assignment to the rules that satisfies a given formula, while maximizing the output of another formula.

rules, for which we follow Goodman's thesis [18]: Rather than construct a conventional parse tree, we implement a *semiring parser* [18, 19], where the semiring in question is the *conditional table semiring* [20], which interprets multiplication as conjunction and addition as disjunction. The elements of the semiring are the rules.

Figure 5 illustrates the process on a simple example. In the upper left, we present five candidate productions (numbered 0-4) that define nonterminals A, B and C. To the right of each production is an example of the kind of edge we use in the diagrams to indicate the use of the corresponding production. For instance, production 1 is denoted by a dashed edge and production 2 is denoted by a solid edge. Productions 1 and 2 are identical in this example; a solution can use either one. They help illustrate the choices the algorithm must make. In the upper right, Figure 5 presents two example parse trees, one for the positive example "abbb" and one for the negative example "abbc." On the lower left-hand side, Figure 5 presents the rule provenance computation that the parser implements when processing the positive example (the corresponding computation for the negative example is elided). Here, the non-terminals are replaced by algebraic expressions that represent the rules used to implement that subtree. For instance, the orange box at the bottom left contains the algebraic expression "3," which represents the fact that rule 3 must be used to parse the underlying string. The box above it contains the expression "1.3 + 2.3," which indicates that either the rules 1 and 3 or the rules 2 and 3 must be used to parse "bb." Above that, there appears another provenance expression, which may be reduced to the equivalent expression "1.3 + 1.2.3 + 2.3" thanks to the fact we are working in the conditional table semiring. The final expression produced is "0.1.3 + 0.1.2.3 + 0.2.3." This expression may then be converted to the first conjunct shown in the provenance formula by replacing "." with conjunction and "+" with disjunction. The corresponding logical formula was a positive one since we are processing a positive example. The formula corresponding to a negative example is negated. A MaxSMT solver such as Z3 will return a solution with maximal weight. One possible solution is the set of rules $\{0, 1, 3\}$.

6 EXPERIMENTAL RESULTS

This section answers the following research questions.

SGI Benchmark Suite		
Name	**Size**	**Description**
States	368	US State identifiers. Permits acronyms, full names, and abbreviations.
Phone #s	257	Phone numbers. Permits local and international phone numbers.
Times	201	Time of day in a variety of formats.
Floats	110	Floating-point numbers. Includes grammars for different scientific notations and standard decimal form.
Emails	490	Email addresses. Includes grammars that accept emails from specific or arbitrary domains.
Names	139	Human identifiers. Includes grammars specifying salutations, post-nominal titles, and acronyms.
Streets	164	US street identifiers. Includes grammars that demand specific suffixes and directions.
Dates	311	Calendar dates. Includes month-first, day-first, and year-first formats.
Addresses	588	US street addresses. Uses the States and Streets metagrammars to identify those portions of the address.
XML	492	XML Files. Permits 10 classes of XML elements. It discovers the identifiers for element classes and the recursive schemes. In effect, it imputes the structural component of a schema definition.
RFCCSV	470	CSV: Exactly RFC 4180 [21]. Automatically infers cell type.
IdealCSV	500	CSV: Generalized RFC 4180 [21]. Admits more kinds of separators including comma, tab, semi-colon than the RFC. Automatically infers cell type.

Fig. 7. Information on metagrammars for the SGI benchmark suite. **Name** identifies the grammatical domain. **Size** is the AST count of the metagrammar. **Description** characterizes the grammatical domain, including an incomplete description of how some induced grammars differ from each other.

- **Expressiveness.** Can Saggitarius specify real-world grammatical domains?
- **Efficiency.** Is Saggitarius fast enough to be used in day-to-day development?
- **Algorithmic Improvements.** Does our grammar induction algorithm perform well compared to prior work?

This section concludes with two case studies: one using Saggitarius for learning an XML grammar and the other for detecting CSV dialects.

6.1 Expressiveness

To evaluate the expressiveness of Saggitarius, we developed a benchmark suite of 11 grammatical domains. Figure 7 describes each domain briefly. We have written metagrammars for each of the domains. On average, it took 327 AST nodes to describe such a benchmark metagrammar.

Two of the entries in the benchmark suite are of particular note because we use them in our case study: RFCCSV and IdealCSV. Metagrammar RFCCSV is the Saggitarius definition of the CSV specification given in RFC 4180. IdealCSV is a slight generalization of the specification. In both cases, CSV files containing errors, such as inconsistent use of separators or erroneous escapes or quotes, will be rejected. Both Saggitarius CSV definitions use grammar induction to infer the types of cells in a column.

On occasion, our initial domain definitions were erroneous, returning undesired results on certain test data sets. We found such errors were relatively easy to fix by adjusting preferences. §6.4 presents a case study that further tests the expressiveness of Saggitarius on messy, erroneous CSV files.

6.2 Efficiency

Benchmarks. To evaluate the efficiency of Saggitarius, we developed a benchmark suite of 110 induction tasks, ten tasks for each of the first eleven grammatical domains in Figure 7.[2] In constructing our ten tasks, we varied the number of positive examples (PEs) and negative examples (NEs), and whether or not there was a grammar in the domain that satisfied the examples. The ten tasks are:

(1) Induce a grammar from 1 PE.
(2) Induce a grammar from 1 NE.
(3) Induce a grammar from 1 PE and 1 NE.
(4) Induce a grammar from 10 PEs.
(5) Induce a grammar from 5 PEs and 5 NEs.
(6) Induce a grammar from 1 PE and 20 NEs.
(7) Induce a grammar from 20 PEs and 20 NEs.
(8) Attempt to induce a grammar from 1 PE and 1 NE, where there is no grammar in the domain that accepts the PEs while rejecting the NEs.
(9) Attempt to induce a grammar from 10 PEs and 10 NEs, where there is no grammar in the domain that accepts the PEs while rejecting the NEs.
(10) Attempt to induce a grammar from 20 PEs and 1 NE, where there is no grammar in the domain that accepts the PEs while rejecting the NEs.

Algorithms. To experiment with the effectiveness of different grammar induction algorithms, we ran Saggitarius in three different modes.

- Saggitarius: The algorithm described in §5.
- **ProSynth**: In this mode, Saggitarius iteratively guesses subsets of candidate rules, parses the data with the given subset, and then checks to see whether the subset succeeds to parse the positive but not the negative examples. If it fails, it uses a counter-example-guided algorithm to generate a new set of candidate rules and repeats the process. This algorithm was inspired by the work on ProSynth [29], an inductive logic programming engine. Since parsing via context-free grammar is a refinement of logic programming, the ProSynth algorithm may be effective. In particular, if it is able to find a candidate set early, it may avoid the heavy cost of parsing all data with a large ambiguous grammar.
- **ProSynth++**: In this mode, Saggitarius parses the example data once using all candidate productions. It constructs a full provenance tree as in Saggitarius. Rather than build a single large MaxSMT formula, it considers candidate rule sets in a counter-example guided loop, as in **ProSynth**. This mode avoid constructing a large SAT formula, but pays a price by making many SAT calls.

Experimental setup. All experiments were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 RAM running macOS Catalina. We ran each benchmark 10 times, with a timeout of 60 seconds and report the average time. If any of the 10 runs times out then we consider the benchmark as a whole to have timed out.

Results. Figure 8 summarizes the performance of Saggitarius in the three different modes on our our benchmark suite. It shows the number of benchmarks Saggitarius can complete in a given amount of time in each mode.

If we consider first the effectiveness of our primary algorithm (Saggitarius), we can see that Saggitarius completed 101 of the 110 benchmarks in under 1 minute. In 6 of the 9 instances

[2]We did not include RFCCVS since it is a subset of IdealCSV.

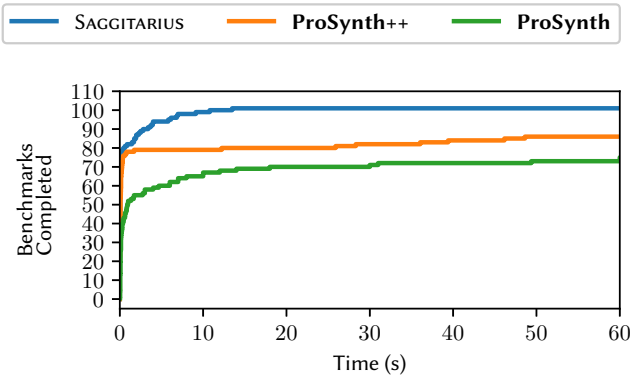

Fig. 8. Number of benchmarks that terminate in a given time in different modes.

the system timed out, it did so because no grammar matched the example data (by experiment design) and Saggitarius looped, attempting to forever extend the range of one of its unbounded `[0..infty)` index sets. A useful improvement to the system would be to look for heuristics that might detect data sets that will not be satisfied by any grammar in a given infinite metagrammar. The remaining 3 of the 110 experiments ran long due to particularly slow SMT calls. Interestingly, a large portion of this slowdown appears to be due to preferences—removing all preferences drastically speeds up these tasks, allowing them to complete in under a minute (though the returned grammars are not the desired ones).

When comparing Saggitarius to alternative algorithms **ProSynth** and **ProSynth++**, several trends emerge. First, we found that Saggitarius solved every benchmark but one faster than either **ProSynth** and **ProSynth++**. That one benchmark is the one of the two on which Saggitarius did not complete in a minute due to the slow SMT call. **ProSynth++** makes multiple smaller SMT calls, which usually take longer in aggregate, but in this one case avoided generating a particular challenging SMT formula (and was solved in 48 seconds).

More broadly, the cost of all of the algorithms grows substantially with the number of rules in the metagrammar. The ProSynth-inspired algorithms, in particular, suffer significantly as rule sets increase because it takes them more iterations to build up a sufficiently constrained SAT call to find a solution. Saggitarius is also negatively impacted by having more rules, but in a different (and less substantial) way. More rules results in slower parsing, which is the typical bottleneck for Saggitarius.

Broadly, across all the algorithms, we found that the *number* of examples did not affect system performance substantially. However, the *length* of an example can have a significant impact when the metagrammar is highly ambiguous. Early parsing with large numbers of ambiguous rules is a costly enterprise that can grow cubically with the length of the input in the worst case. This parsing cost tends to dominate as examples grow in length.

6.3 Case Study: XML Metagrammar Development

XML is a general-purpose document class often used for data storage and data transfer. We have written an *XML metagrammar* that can generate a custom grammar from a set of positive and negative examples. Figure 9 presents a simplified version of this XML metagrammar and Figure 10 presents an XML document with which to specialize the metagrammar.

The metagrammar itself permits 8 different types of *elements*, each of which can be a *data element*, a *container element*, or both. A *data element* is an XML element that contains some text before its

```
S -> ??{i in [0,8)} Element{i}.

Element{i in [0,8)} ->
  ? Header{i} Text Footer{i}
  ? Header{i} Container{i} Footer{i}

Container{i} ->
  ? ""
  ??{j in [0,8)} Element{j} Elements{i}
```

Fig. 9. Simplified XML metagrammar.

```
<class>
  <name>Intro to Computer Science</name>
  <students>
    <student>Jane Doe</student>
    <student>John Public</student>
  </students>
</class>
```

Fig. 10. Example XML document describing an introduction to computer science class.

closing tag. A *container element* is an XML element that contains some XML before its closing tag. For expository purposes, Figure 9 specifies only these two types of XML elements, and does not include additional features such as XML attributes, though the metagrammar we defined and used in our experiments does.

Our example XML document contains four different XML elements: `class`, `name`, `students`, and `student`. The `class` and `students` elements are container elements while the `name` and `student` elements are data elements.

If you look carefully, you will notice something unfortunate about this simple XML metagrammar: It contains a lot of redundancy. For instance, consider the grammars G_1:

```
S -> Element{0}

Element{0} -> "<a>" Container{0} "</a>"
Element{1} -> "<b>" Text "</b>"

Container{0} -> Element{1}
```

and G_2:

```
S -> Element{0}

Element{0} -> "<a>" Container{0} "</a>"
Element{2} -> "<b>" Text "</b>"

Container{0} -> Element{2}
```

These two grammars are clearly semantically equivalent, but syntactically distinct. The presence of syntactic redundancy in a search space is a well-known problem in program synthesis in general—in any SyGuS system, such redundancy enlarges the search space and slows down synthesis. Unsurprisingly, such redundancy has an impact on Saggitarius as well. More specifically, we

found that for sufficiently complex examples, when the amount of redundancy is too large, the Max-SMT calls use to search the grammar space are prohibitively slow.

```
S -> Element{0}.

Element{i in [0,8)} ->
  ? Header{i} Text Footer{i}
  ? Header{i} Container{i} Footer{i}

Container{i} ->
  ? ""
  ??{j in [0,8)} Element{j} Elements{i}

constraint &&{i in [1,8)} (Productions(Element{i}) > 0) => (Productions(Element{i-1}) > 0).
```

Fig. 11. Simplified XML metagrammar.

Fortunately, our metagrammar language affords a solution: The programmer can add constraints to cut the search space back down. In Figure 11, we have added constraints to ensure that if `Element{i}` has a nonzero number of productions, then `Element{i-1}` must *also* have a nonzero number of productions. We further require that the first element must be `Element{0}`. Therefore, the singleton grammar with `Element{1}` is no longer a valid grammar. Furthermore, if `Element{0}` must contain exactly one text element within it, and no other elements are used, that element will always be `Element{1}`. Because `Element{1}` has no productions, `Element{2}` cannot be the single text element within `Element{0}`. This is a tactic that can be applied to general recursive data – when constructing a metagrammar with multiple mutually recursive elements, providing a default ordering in which those elements are used helps constrain the search space. In our case, `Element{0}` gets used first, followed by `Element{1}`, and so on.

We run Saggitarius on both of these metagrammars, and show the difference in run times in Figure 12. By minimizing the ambiguity of the metagrammar, XML inference becomes much more tractable even though the language is just as expressiveness as it was before. Additionally, while synthesis failed on one benchmark (the benchmark corresponding to task (7), involving inducing a grammar from 20 PEs and 20 NEs, we found that the time taken per additional example increased at a substantially slower rate than without our optimization.

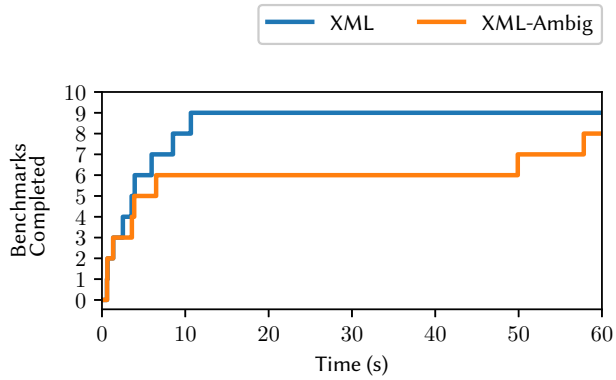

Fig. 12. Number of benchmarks that terminate in a given time on XML and XML-Ambig.

Takeaways. One of the strengths of SGI lies in the ability of the metagrammar developer to control the search space. By refactoring the metagrammar, the metagrammar developer can encode the same grammatical domain in a way that is more amenable to synthesis. Doing so provides fundamental advantages over general-purpose, unguided grammatical inference techniques where the search space is overwhelmingly large and unconstrained.

6.4 Case Study: CSV Dialect Detection

RFC 4180 provides the standardized definition of CSV, but there are many real-world data files that do not conform to the standard. Such data files may contain irregular "table-like" data in which rows have differing numbers of columns, cells are malformed (containing dangling delimiters), and the entire table is surrounded by unstructured text. To handle such messy CSV-like data, Van den Burg *et al.* [44] designed CSV Wrangler, a tool for inferring the *dialect* of messy, possibly erroneous CSV documents. A CSV dialect is a triple of a cell delimiter, a quote character, and an escape character. The delimiter separates cells; the quote character surrounds strings, allowing delimiters to appear within a cell, and the escape character admits nested quotes. For example, the row below should be parsed with a comma delimiter, a quotation mark as the quote symbol and a backslash as the escape character.

```
0, "\"How's the weather?\" she asked.", "1"
```

CSV dialects are typically ambiguous: The row above can also be parsed as a single string with no delimiter, quote character, or escape character. In general, any file can be parsed as a single string with the trivial dialect. Van den Burg *et al.* has defined a custom algorithm that scores CSV dialects based on how consistent the resulting rows are (the *pattern score*) and how well-typed the cells are (the *type score*) [44], attempting to mimic the human process of recognizing the data in a messy CSV file. Note that this algorithm does not "parse" the CSV per se, it is merely designed to detect the dialect. Of course, once a dialect is detected, a file may be parsed later using the inferred delimiter, quote and escape characters, though the presence of errors, such as missing end-quotes or escapes, may give rise to unintended results. Python's CSV sniffer [35] is a similar sort of dialect detector. Dialect detection is difficult because many real CSV dialects are ambiguous. For example, a CSV in our case study contains many lines of the form

```
(tab)(tab)   C1(tab)   C2,C3(tab)  C4,C5(tab)  C6
```

The resulting dialect is ambiguous as it could be part of a well-formed CSV with either the tab or comma delimiter. Under RFC guidelines, the CSV should be parsed with the comma delimiter, but the CSV Wrangler tool and the human labeler used in experiments to evaluate CSV Wrangler chose the tab delimiter [45].

To experiment with CSV dialect detection using Saggitarius, we developed two additional CSV specifications, named *Strict* and *Lax*. Strict requires (1) (unescaped) quote characters not be used nested within a cell (they may wrap the entire contents of a cell), and (2) delimiters (commas, semicolons, tabs and vertical bars) internal to a cell must be quoted. The latter constraint arises because strict will not break ties between these potential delimiters. Lax imposes neither requirement, but its preferences attempt to mimic the human priors that bias towards one dialect or another. Like Van den Burg's tool, Lax prefers dialects that maximize the number of rows with similar length and that lead to cells that do not violate the Strict restrictions. Both Lax and Strict generate grammars when supplied with data files; those grammars reveal the dialect the specifications have identified.

Lax is considerably more costly to execute than Strict: Lax is highly ambiguous as it admits dangling quotations (adding to the complexity, the quotation character is not fixed) and ill-formed cells. It uses a preference system to sort through the multitude of possible parses. Because Lax is

Detector	Yes	No	No Dialect	Timeout
RFC	41	14	44	0
Strict	67	12	8	13
StrictLax	70	14	0	16
CSV Wrangler	86	13	1	N/A
Python Sniffer	81	14	5	N/A

Fig. 13. CSV Analysis. **Yes**: Number of files on which the tool aligns with the human label. **No**: Number of files on which the tool does not align with the human label. **No Dialect**: Number of files for which no dialect is reported. **Timeout**: Number of files on which the tool times out.

more lenient than Strict, the two processes can be pipelined—only when Strict fails do we attempt Lax. We call the pipelined system StrictLax.

Benchmarks. We measured the performance of Saggitarius on 100 ASCII-128 CSVs with human labels drawn from Van den Burg *et al.*'s GitHub repo [45]. More than half of the benchmark suite (59/100) does not obey the CSV RFC. In such situations, it is difficult for humans to unambiguously identify the dialect of files, as we discuss below.

Experiments. We attempted dialect detection using 5 different tools: (1) RFC (the Saggitarius specification of the CSV RFC), (2) Strict, (3) StrictLax (Strict followed by Lax), (4) CSV Wrangler (Van den Burg's tool), and (5) the Python Sniffer. In these experiments, Earley parsing was the bottleneck for the Saggitarius system on large CSV benchmarks. To speed up processing, we truncated the CSV files to 20 lines prior to processing them with Strict or RFC and to 5 lines prior to processing them with Lax. When Strict timed out on a 20-line file, we truncated the file to 5 lines and tried again. We use the same process for RFC evaluation, but do not pass any results forward to Lax.

Figure 13 presents the results from our tool as well as corresponding results we retrieved from Van den Burg *et al.*'s GitHub repository [45] for CSV Wrangler and Python Sniffer. A first observation is that RFC misclassifies a number of the CSV files relative to a human labeller. It does so when, for instance, it is able to interpret an entire line as a single cell because that line uses a non-standard separator, such as a semi-colon, that the RFC does not recognize. A second observation is that the error rate in all the tools is fairly similar. In other words, identifying the dialect of these messy CSV files is fairly difficult for any tool. The main difference between Saggitarius and the custom tools is that Saggitarius's grammar inference engine will sometimes fail to terminate.

We investigated some of the circumstances in which Saggitarius varied from human-labelled data. Figure 14 summarizes our findings. In all, we identified 8 different situations in which the StrictLax results disagreed with the human labels. Specifically, we believe that in scenarios 2, 3, and 8 (5 tests), our tool is correct, or at least not wrong. In addition, we might also argue, even after hand inspection, that the tool returns reasonable answers in situations 1, 6 and 7. One author believes spaces should not be used as delimiters in CSV so it seems we have disagreements among authors as well (situation 5). The failures caused by truncation (situation 4) are clear failures by the tool because of decisions to limit the common case time spent analyzing data.

Takeaways. First, we illustrate that it is possible to develop more than one specification for an ambiguous grammatical domain. Multiple generated tools can then be arranged in a pipeline and take advantage of time/accuracy tradeoffs. Second, real-world data is often messy. Even hand-tuned, custom tools such as Python's CSV sniffer or the CSV Wrangler can make mistakes; humans often disagree. Nevertheless, our SGI-generated tool reports "no dialect" a similar number of times to these custom-built tools. The CSV sniffer and the CSV Wrangler align with outside human labeller

Id	Saggitarius behavior	Frequency
1**	CSV preferences allowed algorithm to return no delimiter character	4
2*	chose different delimiter than hand-label but file was ambiguous.	2
3*	did not find any characters because of leading metadata	2
4	did not find quote or escape character because of truncation	2
5***	did not accept space as a valid delimiter	1
6**	did not identify quote in ill-quoted file	1
7**	unquoted "," character lead to false "," delimiter	1
8*	did not find improperly used escape character	1

Fig. 14. Reasons for StrictLax misclassification. In our analysis, cases marked * are inherently ambiguous or mislabelled by the human. Cases marked ** also represent reasonable behavior by our system in our judgement. We could change our specification to accommodate space as a separator, though doing so may have unanticipated consequences.

a few more times than our tool, but the authors of this paper actually disagree with that outside human labeller in almost all such cases.[3] Hence, when it comes to the unambiguous formats where humans agree, our SGI-generated tool is very effective.

7 RELATED WORK

Grammar induction. Grammar induction traces back to at least the 60s when Gold [17] began studying models for language learning and their properties. Later, Angluin [3, 4] 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 synthesis simple regular expressions. More recently, FlashProfile [32] 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 [42], who use probabilistic techniques to infer grammars. Fisher *et al.* [15] explored inference of grammars for "ad hoc" data, such as system logs, in the context of the PADS project [14]. Lee [24] 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 [6] 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

[3]It may be that the outside labeller and the authors of the CSV Wrangler and CSV sniffer are aware of some other criteria for disambiguating CSV data that we are not, which is why they and the tools align but we do not. In such a case, we might be able to add this criteria to our SGI specifications as well.

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.* [8] 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 [33] uses positive and negative examples to fix erroneous regular expressions. Both RFixer and Saggitarius 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. Saggitarius 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 [1, 2, 40, 41]. Much of that work has focused on data transformations, including spreadsheet manipulation [5, 22, 48], string transformations [27, 28, 47], and information extraction [23, 36]. 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 [34] and Prose [37]. These systems are "meta" program synthesis engines—they help engineers design program synthesis tools for different domain-specific languages. Similarly, Saggitarius is a "meta" framework for syntax-guided grammar induction, helping users perform grammar induction in domain-specific contexts. Of course, Saggitarius, 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 [11], and Logic Program Synthesis [29, 39]. 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 [29] 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.

8 CONCLUSION

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, provide a variety of examples of such domains in the wild, and propose Syntax-guided Grammar Induction as method for supporting efficient grammar induction within such domains.

We also design a language, called Saggitarius, for specifying meta-grammars, which define grammatical domains. Saggitarius 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. We illustrate the use of Saggitarius on a variety of examples and develop a grammar induction algorithm for the system that uses semiring parsing to generate MaxSMT formulae that can be solved via an off-the-shelf theorem prover.

In the future, we look forward to extending Saggitarius with semantic actions, which would enable Saggitarius to be incorporated into existing parser-generators like YACC.

ACKNOWLEDGMENTS

REFERENCES

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

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93. https://doi.org/10.1145/3208071

[3] Dana Angluin. 1978. On the Complexity of Minimum Inference of Regular Sets. *Information and Control* 39, 3 (1978), 337–350.

[4] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (Nov. 1987), 87–106.

[5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 218–228.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. *SIGPLAN Not.* 52, 6 (June 2017), 95–110. https://doi.org/10.1145/3140587.3062349

[7] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. 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. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/extract-me-if-you-can-abusing-pdf-parsers-malware-detectors.pdf

[8] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal Synthesis of Regular Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 487–582.

[9] DARPA SafeDocs Program 2020. https://www.darpa.mil/program/safe-documents.

[10] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[11] Luc De Raedt. 2008. Logical and Relational Learning. In *Advances in Artificial Intelligence - SBIA 2008*, Gerson Zaverucha and Augusto Loureiro da Costa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–1.

[12] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

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

[14] Kathleen Fisher and David Walker. 2011. The PADS Project: An Overview. In *Proceedings of the 14th International Conference on Database Theory (ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 11–17.

[15] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. 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 (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 421–434.

[16] P. Garcia and E. Vidal. 1990. Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12, 9 (1990), 920–925.

[17] E. M. Gold. 1967. Language Identification in the Limit. *Information and Control* 10, 5 (1967), 447–474.

[18] Joshua Goodman. 1998. Parsing Inside-Out. , 19–28 pages. https://dash.harvard.edu/bitstream/handle/1/24829603/tr-07-98.pdf?sequence=1

[19] Joshua Goodman. 1999. Semiring Parsing. *Comput. Linguist.* 25, 4 (Dec. 1999), 573–605.
[20] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535
[21] Network Working Group. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://tools.ietf.org/html/rfc4180. Request for Comments 4180.
[22] 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 P rogramming Languages (POPL '11)*. ACM.
[23] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. *ACM SIGPLAN Notices* 49 (06 2014).
[24] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *ACM SIGPLAN International Conference on Generative Programming*. 70–80.
[25] Ke Liu. 2017. 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.
[26] Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Quotient Lenses. *Proc. ACM Program. Lang.* 2, ICFP, Article 80 (July 2018), 29 pages. https://doi.org/10.1145/3236775
[27] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Ste ve Zdancewic. 2018. Synthesizing Bijective Lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pr ogramming Languages (POPL 2018)*.
[28] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proc. ACM Program. Lang.* 3, ICFP, Article 95 (July 2019), 28 pages. https://doi.org/10.1145/3341699
[29] David Zhao Mayur Naik Bernhard Scholz Mukund Raghothaman, Jonathan Mendelson. 2020. Provenance-Guided Synthesis of Datalog Programs. https://doi.org/10.1145/3371130
[30] Rahman Nozohoor-Farshi. 1991. *GLR Parsing for ϵ-Grammers*. Springer US, Boston, MA, 61–75. https://doi.org/10.1007/978-1-4615-4034-2_5
[31] Jose Oncina and Pedro Garcia. 1992. Identifying Regular Languages In Polynomial Updated Time. In *Pattern Recognition and Image Analysis*, N Perez de la Blanca, A. Sanfeliu, and E Vidal (Eds.). World Scientific, 49–61.
[32] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276520
[33] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360565
[34] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126.
[35] Python Software Foundation. 2020. CSV File Reading and Writing. https://docs.python.org/3/library/csv.html#csv.Sniffer.
[36] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction using Predictive Program Synthesis. In *AAAI*. 882–890.
[37] Microsoft Research. 2020. Prose. https://www.microsoft.com/en-us/research/group/prose/.
[38] R. L. Rivest and R. E. Schapire. 1989. Inference of Finite Automata using Homing sequences. In *Twenty-first Annual Conference on Theory of Computing*. 411–420.
[39] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. 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, 6117–6124. https://doi.org/10.24963/ijcai.2019/847
[40] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. *SIGPLAN Not.* 40, 6 (June 2005), 281–294. https://doi.org/10.1145/1064978.1065045
[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. https://doi.org/10.1145/1168919.1168907
[42] Andreas Stolcke and Stephen M. Omohundro. 1994. Inducing Probabilistic Grammars by Bayesian Model Merging. *CoRR* abs/cmp-lg/9409010 (1994).
[43] Loup Vaillant. 2020. Earley Parsing Explained. http://loup-vaillant.fr/tutorials/earley-parsing/
[44] Gerrit J. J. van den Burg, Alfredo Nazábal, and Charles Sutton. 2018. Wrangling Messy CSV Files by Detecting Row and Type Patterns. https://arxiv.org/pdf/1811.11242.pdf arXiv:1811.11242v1.

[45] Gerrit J. J. van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. CSV_Wrangling. https://github.com/alan-turing-institute/CSV_Wrangling.

[46] Enrique Vidal. 1994. Grammatical Inference: An Introduction Survey. In *Second International Colloquium on Grammatical Inference and Applications (Machine Perception and Artificial Intelligence)*. 1–4.

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

[48] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133886

[49] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

[50] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM.