FIDEX: Filtering Spreadsheet Data using Examples

Xinyu Wang
UT Austin, USA
xwang@cs.utexas.edu

Sumit Gulwani Rishabh Singh
Microsoft Research, USA
sumitg@microsoft.com risin@microsoft.com

Abstract

Data filtering in spreadsheets is a common problem faced by millions of end-users. The task of data filtering requires a computational model that can separate intended positive and negative string instances. We present a system, FIDEX, that can efficiently learn desired data filtering expressions from a small set of positive and negative string examples.

There are two key ideas of our approach. First, we design an expressive DSL to represent disjunctive filter expressions needed for several real-world data filtering tasks. Second, we develop an efficient synthesis algorithm for incrementally learning consistent filter expressions in the DSL from very few positive and negative examples. A DAG-based data structure is used to succinctly represent a large number of filter expressions, and two corresponding operators are defined for algorithmically handling positive and negative examples, namely, the intersection and subtraction operators. FIDEX is able to learn data filters for 452 out of 460 real-world data filtering tasks in real time (0.22s), using only 2.2 positive string instances and 2.7 negative string instances on average.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Human Factor

Keywords Program Synthesis, Data Filtering, Regular Expressions, Programming By Examples

1. Introduction

Data filtering in spreadsheets is a common problem faced by millions of spreadsheet users. In spreadsheets with large amounts of data, users often want to work with a subset of data. Spreadsheet systems like Excel allow some basic auto-filtering using concrete strings and also more advanced filtering capabilities using regular expressions. Unfortunately, these features are either not powerful enough or not accessible to most of the end-users who have diverse backgrounds and lack necessary programming skills to write customized regular expressions to filter their data [15].

These end-users have to resort to online help forums to ask for help from experts with the desired filtering expressions. We performed an extensive study of these forums and observed that these users were able to specify their intent using examples. The experts would ask for additional examples in case there was still some ambiguity and in many cases were able to provide the desired filtering expressions after a few iterations. In this paper, we present a system that allows end-users to perform data filtering tasks automatically using a similar example based interface.

Data filtering as a unit operation is also an important subtask in many other task domains such as data extraction and transformation. In the data extraction domain, a user might want to extract data from only specific entries which are located by a filter expression whereas in the data transformation domain, a user might want to perform different transformations on different types of data which are separated by filtering expressions. These operations are a key part of *data wrangling* [18, 19], a new term coined for the overall process of converting data from its raw format to a more structured form that is amenable to querying and analysis for drawing insights. It is estimated that data scientists sometimes spend upto 80% of their time wrangling data [9, 33].

The automation of data filtering from examples requires learning a computational model that can separate the intended positive and negative string instances. In order to learn a customized model, end-users should be able to guide the learning process. Since 99% of end-users are non-programmers and struggle with repetitive data manipulation tasks [15], only simpler specification mechanisms such as example based interfaces are a viable alternative. *Automatically learning data filtering models from examples* is also motivated by reasons that are different than those behind traditional automata based inference [3, 14] and machine learning based filtering techniques [7]. In the fuzzy tasks, machine learning techniques have the potential to learn black-box filters that might achieve better precision than models produced by human experts. In contrast, we aim to produce

OOPSLA'16 October 25–30, 2016, The Netherlands

ACM [to be supplied]. . . $15.00

human-readable filter expressions with perfect precision for relatively simple and well-defined data filtering tasks.

We present a system, FIDEX (stands for FIltering Data using EXamples), that efficiently learns intended data filtering expressions from a small set of positive and negative string examples. Our approach has two key components: First, we have identified an appropriate computational model for such filtering tasks in the form of a Domain-Specific Language (DSL) that corresponds to a restricted and structured form of regular expressions. We show that such a model is expressive enough to represent the expressions needed for real-world data filtering tasks and at the same time is restricted enough to enable efficient learning. Second, we present a sound, complete, and efficient learning algorithm for learning the filter expressions in the DSL from a few examples.

There are two key technical challenges in FIDEX. First, an efficient representation of the desirable expressions is necessary for better scalability since there are typically a large number of expressions in the DSL that conform with a set of examples. Second, an efficient algorithm to handle both positive and negative examples is necessary for efficient search. Our synthesis algorithm leverages a Directed Acyclic Graph (DAG) data structure based representation to succinctly represent the set of all filter expressions in the underlying DSL that are consistent with the examples provided by the user. The algorithm leverages the intersection operator and a novel subtraction operator over the DAG to incrementally refine the set of filter expressions that are consistent with both the positive and negative instances. Learning an intended filter expression from as few examples as possible is critical from the usability standpoint [37]. We define a ranking structure over the underlying space of the DSL expressions and develop an algorithm to efficiently identify the highest ranked filter expression from expressions represented in a DAG.

Our DAG data structure is inspired from the version-space algebra based DAG representation in FlashFill [16]. However, unlike FlashFill where a DAG succinctly represents a large set of string concatenation expressions, a DAG in FIDEX allows for more concise representation with multiple start nodes, where each path in the DAG represents a set of token sequences. Moreover, unlike FlashFill and other version-space algebra based synthesis approaches [17], FIDEX uses a novel subtraction operator over the DAG structure to algorithmically support negative examples.

We evaluate FIDEX on 460 benchmarks taken from various real-world data filtering tasks on online forums and industrial sources. FIDEX is able to learn the desired filter expressions on 452 benchmarks as compared to 343 solved by the baseline approach. It takes about 0.22 seconds on average and requires on average only 2.2 positive string instances and 2.7 negative string instances for each benchmark.

This paper makes the following key contributions:

- We present a DSL that can express filter expressions for several real-world data filtering tasks (§3).
- We present a DAG data structure for succinctly representing the DSL expressions, and the intersection ($\bigotimes$) and subtraction ($\ominus$) operators for handling positive and negative string instances, respectively (§5).
- We present a sound, complete, and efficient synthesis algorithm to learn filter expressions in the DSL using the $\bigotimes$ and $\ominus$ operators in an incremental fashion (§6).
- We present a comprehensive evaluation of our system FIDEX on 460 real-world data filtering tasks (§7).

2. Motivation

In this section, we present a few examples taken from online help forums that show the diverse kinds of filtering operations requested by the spreadsheet users. We observed that in most of these forum posts, users typically describe their intent using natural language and examples. The experts on the forum then ask for additional examples and a sample of their data in case there is some ambiguity in the task description. FIDEX presents a similar interface for spreadsheet users that allows them to provide positive and negative examples on a sample of their data. The positive examples correspond to data rows that they would like to see after the filtering operation, whereas the negative examples correspond to rows they would like to be removed from the result. FIDEX then learns a data filtering predicate from the provided examples and runs it on the remaining spreadsheet. After inspecting the results, a user can provide additional examples in case some strings are not filtered as intended, and the learning process continues until FIDEX learns the desired filter expression.

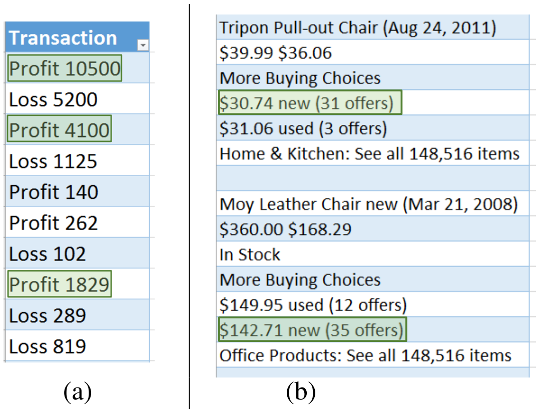
Example 1 (Matches)**.** *On StackOverflow*[1]*, a user posted the following task to find a regular expression that matches the strings* `"RJ123456"`*,* `"PY654321"` *and* `"DD321234"` *but not the strings* `"DDD12345"` *and* `"12DDD123"`*.*

There were some additional interactions on the forum that involved experts asking if strings such as `"ABC1234567"` were possible, if the first two letters can also be lowercase, and if the digits can only be Western Arabic numerals (0-9) or digits from other Numeral systems. Some suggested solutions were: `[A-Z]{2}\d{6}`, `[a-zA-Z]{2}\d{6}`, etc. FIDEX learns a filter expression to match strings consisting of a sequence of two letters followed by six digits and filters out other strings, given these examples.

Example 2 (StartsWith)**.** *An Excel user had a list of stock transactions in a column and wanted to filter for rows for Profit transactions with a value greater than 1000 for further analysis as shown in Figure 1(a).*

The highlighted data entries are considered as positive examples and the table cells that are not highlighted until the last positive example are implicitly considered as negative

[1] http://stackoverflow.com/questions/10439666/regex-pattern-any-two-letters-followed-by-six-numbers



Transaction
Profit 10500
Loss 5200
Profit 4100
Loss 1125
Profit 140
Profit 262
Loss 102
Profit 1829
Loss 289
Loss 819

(a)

```
Tripon Pull-out Chair (Aug 24, 2011)
$39.99 $36.06
More Buying Choices
$30.74 new (31 offers)
$31.06 used (3 offers)
Home & Kitchen: See all 148,516 items

Moy Leather Chair new (Mar 21, 2008)
$360.00 $168.29
In Stock
More Buying Choices
$149.95 used (12 offers)
$142.71 new (35 offers)
Office Products: See all 148,516 items
```

(b)

Figure 1. (a) Filtering for data rows for profit transactions with value greater than 1000, (b) Filtering for data rows corresponding to the new item price.

examples. A user can double click a highlighted entry for it to be considered as a negative example. Given the first three positive examples (rows 1, 3, and 8), FIDEX learns the desired filter expression StartsWith("Profit "·d·d·d·d) for this task, where the token d matches any digit from 0-9.

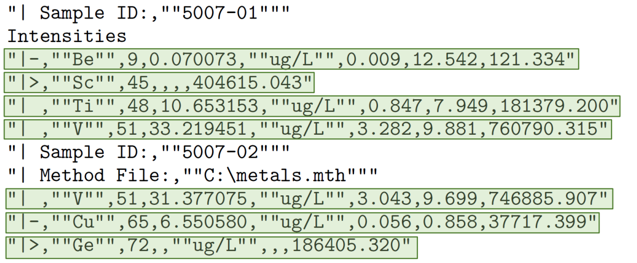
Example 3 (Contains). *A user wanted to filter for rows that correspond to the* new *item price of a product in a csv file consisting of a long list of products as shown in Figure 1(b). The csv file was obtained by scraping the product information from the Amazon website. The highlighted lines are positive examples and all other lines are negative examples.*

Learning the regular expression to filter only rows that correspond to the new item price is quite challenging. The expression that looks for lines beginning with a "$" does not suffice since there are other lines that also begin with a "$". Similarly, looking for lines containing the string "new" (or "new (") also does not suffice. Given 2 positive and 3 negative examples, FIDEX learns the filter expression that checks for the presence of the pattern "w"·ws·lparen·num, where "w" matches the constant string "w", ws matches a white space, lparen matches a left parenthesis, and num matches a number.

Example 4 (EndsWith). *A chemist wanted to select lines corresponding to the intensity readings of different elements from a lab report as shown in Figure 2.*

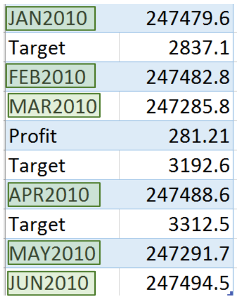
There is no consistent StartsWith logic as there is no prefix shared by all desired strings. An expression that checks for presence of constant string "ug/L" (Contains) does not suffice because some readings have missing field values, and Matches does not work for the same reason. For this task, FIDEX learns the filter expression that matches lines ending with the expression num · dquote, where dquote matches the double quote character.

Example 5 (Matches). *An Excel user wanted to filter for rows that correspond to only monthly transaction values as shown in Figure 3. The spreadsheet column also included other data items such as revenue, target, profit numbers etc. that made the filtering task challenging for the user.*



```
"| Sample ID:,""5007-01"""
Intensities
"|-,""Be"",9,0.070073,""ug/L"",0.009,12.542,121.334"
"|>,""Sc"",45,,,,404615.043"
"|  ,""Ti"",48,10.653153,""ug/L"",0.847,7.949,181379.200"
"|  ,""V"",51,33.219451,""ug/L"",3.282,9.881,760790.315"
"| Sample ID:,""5007-02"""
"| Method File:,""C:\metals.mth"""
"|  ,""V"",51,31.377075,""ug/L"",3.043,9.699,746885.907"
"|-,""Cu"",65,6.550580,""ug/L"",0.056,0.858,37717.399"
"|>,""Ge"",72,,""ug/L"",,,186405.320"
. . . . . . .
```

Figure 2. Selecting element intensity measurements for all elements from a lab experiment report.



JAN2010	247479.6
Target	2837.1
FEB2010	247482.8
MAR2010	247285.8
Profit	281.21
Target	3192.6
APR2010	247488.6
Target	3312.5
MAY2010	247291.7
JUN2010	247494.5

Figure 3. Filtering for all the Month-Year rows.

The challenge in this problem is to learn a filter expression to match all the month-year strings in the spreadsheet. Given 1 positive and 2 negative examples, FIDEX learns a filter predicate Matches(Word · Num) that matches the string completely for the occurrence of a Word followed by a Num.

Example 6 (Disjunction). *A user had a list of links in a spreadsheet columns and needed to filter the rows that contained only the* http *and* ftp *links.*

This example requires a disjunction and FIDEX learns a filter expression that corresponds to the predicate expression StartsWith("http" $\vee$ "ftp").

3. Domain Specific Language

We now present our domain specific language (DSL) to encode data filtering predicates that can distinguish positive data from negative data. The DSL was designed iteratively based on a large-scale empirical study of the real-world benchmark problems collected from online help forums and other industrial sources. The key idea underlying its design is to impose a structure on the space of possible expressions to enable efficient learning while keeping the language expressiveness to encode real-world data filtering tasks.

The syntax and semantics of the language is presented in Figure 4 and Figure 5 respectively. We use the symbol ϵ to denote an empty string. The notation $[s : l]$ denotes a list of strings with s being the first string in the list and l being the remaining list. Let $s[i, j]$ denote the substring of a string s starting at position i (inclusive) and ending at position j (exclusive), and let $|s|$ denote the length of the string s.

$$\begin{array}{rcl}
\texttt{Filter}\ f & := & \texttt{Filter}(p, L) \\
\texttt{Predicate}\ p & := & \texttt{StartsWith}(v, r) \\
 & | & \texttt{EndsWith}(v, r) \\
 & | & \texttt{Matches}(v, r) \\
 & | & \texttt{Contains}(v, r) \\
\texttt{DisjExpr}\ r & := & \texttt{Disjunct}(ts, r)\ \mid ts \\
\texttt{TokenSeq}\ ts & := & \texttt{Seq}(T, ts)\ \mid T
\end{array}$$

Figure 4. The syntax of the DSL, where T denotes a token, L denotes a list of strings, and v denotes a string variable.

3.1 Tokens and Token Sequences

Tokens in the DSL are picked from a token set that contains two types of tokens: i) Constant tokens and ii) General tokens. A constant token matches only one particular string, e.g. `<A>` matches only the string "A", whereas a general token can match multiple strings, e.g. `<Alpha>` matches a sequence of alphabet letters and `<Num>` matches a sequence of digits. The semantics of token matching is defined by the construction of the token unambiguously.

Specifically, the token set in the DSL consists of constant tokens for i) each uppercase and lowercase letter, ii) each digit between 0-9, and iii) special characters such as hyphen, dot, semicolon, colon, comma, left/right parenthesis/bracket, forwardslash, backwardslash, white space, etc. It includes general tokens for digits, alphabet letters, a sequence of digits, a sequence of alphabet letters, a sequence of uppercase letters, a sequence of lowercase letters, etc. The token set also includes a few more higher-level general tokens, such as date, phone number, etc, to capture the frequently used patterns in practice.

A token sequence ts is a sequence of tokens. The semantics of a token sequence ts matching a string s includes three rules: i) an empty string ϵ is not matched by any token sequence, ii) if ts is simply a token T, then ts matches a string s if T matches s, and iii) if $ts = \texttt{Seq}(T, ts')$ consists of more than one token, it first looks for the longest prefix $s[0, i]$ of s that is matched by the first token T in ts, and then recursively checks if the remaining token sequence ts' matches the remaining substring $s[i, |s|]$. For example, $ts =$ `Seq(<Alpha>,<Num>)` matches string "ABC123", whereas it does not match string "123ABC" or "ABC123DEF". *Note that the number of tokens in a token sequence is unbounded.*

3.2 Disjunctive Expressions

A disjunctive expression r is defined as a disjunction of token sequences: if at least one token sequence in r matches a string s, then r is defined to match s. Adding the disjunction expression enables the DSL to: i) construct expressions that can match "incompatible" strings and ii) simulate the effects of Kleene star, both of which increases its expressiveness.

3.3 Predicates and Filter Expressions

On top of the disjunctive expressions, the DSL has `Predicate` expressions. As shown in Figure 4, it consists of four predicates: `StartsWith`, `EndsWith`, `Matches`, and `Contains`. Predicates generalize the semantics of disjunctive expressions, such that it allows a disjunctive expression r to match a prefix (`StartsWith`), a suffix (`EndsWith`), or a substring (`Contains`) of the string s in addition to matching the whole string (`Matches`).

A filter expression $\texttt{Filter}(p, L)$ maps an input list L of m strings to an output list of n strings, where the predicate p holds for the n output strings ($n \leq m$), i.e., it filters out the other (m-n) strings in L for which p does not hold.

3.4 Design of the DSL

The main design choices we made in design the DSL were regarding: 1) the kinds of predicates, 2) the `Disjunct` expression at the level of token sequences (and not at the level of predicates), and 3) the set of tokens. These design choices enable us to not only express all of our benchmarks, but also to perform efficient learning of such expressions. Although the DSL disallows compositions of predicates, it is already quite expressive because of the use of disjunctions, a large token set, and token sequences with arbitrary lengths.

The DSL can express any data filtering task that i) contains a finite number of strings and ii) each string is of finite length. This is because the token set in the DSL consists of a constant token for each possible character (extensible to arbitrary unicode characters) and the DSL supports disjunctive expressions over token sequences of arbitrary length.

In general, FIDEX only requires the constant tokens, the `Matches` predicate, and the disjunctive expressions of any arbitrary length token sequences to perform any data filtering task with a finite number of finite-length strings. We added other general tokens and predicates to enable FIDEX to efficiently learn more general and simpler expressions.

4. Overview of the Synthesis Algorithm

In this section, we present an overview of the synthesis algorithm in FIDEX on a small running example which is a slightly modified version of Example 1 in Figure 6.

Example 7. *Given a list of strings: [*`"RJ1"`*,* `"PY65"`*,* `"DDD"`*,* `"DD32K"`*,* `"D1"`*,* `"12D"`*], a user wants to filter them to obtain the list of strings: [*`"RJ1"`*,* `"PY65"`*,* `"DDD"`*,* `"DD32K"`*].*

A possible filter expression in our DSL for this task is $\texttt{Filter}(\texttt{StartsWith}(v, \texttt{Seq(<l>,<l>)}), L)$, which filters out the strings that do not start with two alphabet letters. We now sketch the interactive synthesis process in FIDEX to learn such an expression from examples. For simplicity, assume the token set used in FIDEX consists of four tokens: {`<l>`, `<a>`, `<d>`, `<n>`}, where the tokens `<l>`, `<a>`, `<d>`, `<n>`

$$
\begin{array}{rcl}
\llbracket \texttt{Filter}(p, [\,]) \rrbracket\, \sigma & = & [\,] \\
\llbracket \texttt{Filter}(p, [s : l]) \rrbracket\, \sigma & = & \texttt{if}\ \ (\llbracket p(v, r) \rrbracket\, \sigma)\ \ \texttt{then}\ \ s : \llbracket \texttt{Filter}(p, l) \rrbracket\, \sigma\ \ \texttt{else}\ \ \llbracket \texttt{Filter}(p, l) \rrbracket\, \sigma, \texttt{where}\ \ \sigma = \sigma[v \leftarrow s] \\
\llbracket \texttt{StartsWith}(v, r) \rrbracket\, \sigma & = & \exists\, 0 < j \leq |s|, \texttt{such that}\ \ \llbracket r \rrbracket\, s', \texttt{where}\ \ s' = s[0, j], s = \sigma(v) \\
\llbracket \texttt{EndsWith}(v, r) \rrbracket\, \sigma & = & \exists\, 0 \leq j < |s|, \texttt{such that}\ \ \llbracket r \rrbracket\, s', \texttt{where}\ \ s' = s[j, |s|], s = \sigma(v) \\
\llbracket \texttt{Contains}(v, r) \rrbracket\, \sigma & = & \exists\, 0 \leq i < j \leq |s|, \texttt{such that}\ \ \llbracket r \rrbracket\, s', \texttt{where}\ \ s' = s[i, j], s = \sigma(v) \\
\llbracket \texttt{Matches}(v, r) \rrbracket\, \sigma & = & \llbracket r \rrbracket\, s, \texttt{where}\ \ s = \sigma(v) \\
\llbracket \texttt{Disjunct}(ts, r) \rrbracket\, s & = & (\llbracket ts \rrbracket\, s) \vee (\llbracket r \rrbracket\, s) \\
\llbracket ts \rrbracket\, \epsilon & = & \texttt{False} \\
\llbracket \texttt{Seq}(T, ts) \rrbracket\, s & = & \exists\, 0 < i < |s|, \forall i < j < |s|, \texttt{such that}\ \ (\llbracket T \rrbracket\, s[0, i]) \wedge \neg(\llbracket T \rrbracket\, s[0, j]) \wedge (\llbracket ts \rrbracket\, s[i, |s|]) \\
\llbracket T \rrbracket\, s & = & T\ \ \texttt{matches}\ \ s
\end{array}
$$

Figure 5. The semantics of the DSL with an environment σ mapping a string variable v to a string in a list L.

match an alphabet letter, a sequence of alphabet letters, a digit, and a sequence of digits, respectively.

Iteration 1: The user provides a positive example string, "RJ1", to the FIDEX system. FIDEX first tries to learn a filter expression using the StartsWith predicate. It first constructs *all token sequences* (denoted as $\mathcal{D}_1$) for StartsWith predicate that match "RJ1", such as StartsWith(v,<a>), StartsWith(v,<l>), StartsWith(v,Seq(<l>,<l>)) etc. It then picks the top-ranked sequence amongst them (say StartsWith(v,<a>)) and returns the result of the corresponding filter expression (Filter(StartsWith(v,<a>), L)).

Iteration 2: After inspecting the result, the user observes that the string "D1" is not filtered out and provides "D1" as a negative example. Again, FIDEX finds *all the token sequences* (denoted as $\mathcal{D}_2$) for StartsWith predicate to match "D1" and *removes* the token sequences in $\mathcal{D}_2$ from $\mathcal{D}_1$ to obtain *a refined set of token sequences* $\mathcal{D}_3$ ($\underline{\mathcal{D}_3 = \mathcal{D}_1 \bigominus \mathcal{D}_2}$). The token sequences in $\mathcal{D}_3$ match the positive string "RJ1" but not the negative string "D1". As in the first iteration, FIDEX returns the result of the top-ranked filter expression, say Filter(StartsWith(v,Seq(<l>,<l>,<n>)), L).

Iteration 3: However, in this case the synthesized filter expression filters out a desired string "DDD" and the user provides "DDD" as another positive example. The system computes *all token sequences* $\mathcal{D}_4$ for StartsWith predicate for "DDD" and *removes* token sequences in $\mathcal{D}_2$ from $\mathcal{D}_4$ (since $\mathcal{D}_2$ contains all token sequences that match a negative string) to obtain *a refined set* $\mathcal{D}_5$ ($\underline{\mathcal{D}_5 = \mathcal{D}_4 \bigominus \mathcal{D}_2}$). Note that in order to recognize both the first (positive) example and the third (positive) example, we have to consider token sequences in both $\mathcal{D}_3$ and $\mathcal{D}_5$. FIDEX finds *all common token sequences* $\mathcal{D}_6$ in *both* sets $\mathcal{D}_3$ and $\mathcal{D}_5$ ($\underline{\mathcal{D}_6 = \mathcal{D}_3 \bigotimes \mathcal{D}_5}$) and returns the filtering result corresponding to the filter expression Filter(StartsWith(v,Seq(<l>,<l>)), L) in which the token sequence is selected from $\mathcal{D}_6$.

The synthesized filter expression filters out the undesired strings and the synthesis process terminates with 2 positive examples and 1 negative example. There are two interesting points worth mentioning: 1) In the third iteration, since there exists a common token sequence in both sets ($\mathcal{D}_3$ and $\mathcal{D}_5$), the filter expression is guaranteed to match the positive example strings but not the negative example string. However, for more complex cases, it might be the case that there is no common token sequence amongst the two sets. In such cases, FIDEX recognizes all positive examples by generating disjunctive expressions. 2) Using a StartsWith predicate is sufficient for this example task, however, for other scenarios, there might not exist a filter expression with the StartsWith predicate to perform the task. For such cases, FIDEX iterates over other predicates in the synthesis process.

There are several technical challenges in the synthesis algorithm. First, we need a data structure to efficiently compute and represent the set of all expressions in the DSL that are consistent with a given set of positive and negative example strings. Second, we need to design sound and complete synthesis algorithms to incrementally incorporate new positive/negative examples. Finally, we need an efficient and effective ranking algorithm so that users don't need to provide a lot of examples for FIDEX to learn the desired filter expressions. We tackle these challenges in §5 and §6.

5. DAG Data Structure

We use a directed acyclic graph (DAG) based data structure to succinctly represent a large set of token sequences. The set of disjunctive expressions is represented by a collection of DAGs. We note that the notions of a token sequence and a disjunction of token sequences are intrinsically different and we use two different data structures to represent them. Representing them in one data structure might be possible, but it would lead to extra efforts to distinguish them and we might potentially lose the ability to learn disjunctions in an efficient, lazy and clean fashion.

The key idea in the DAG data structure is to represent token sequences as paths in the graph such that tokens (represented as edge labels) can be shared across multiple token

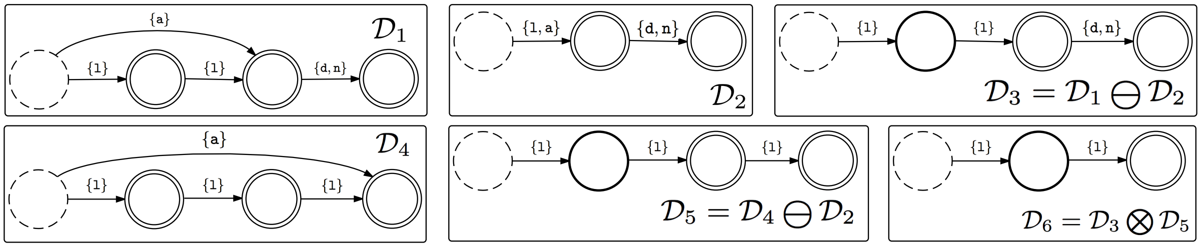

{a}
$\mathcal{D}_1$
{1}
{1}
{d,n}
{1,a}
{d,n}
$\mathcal{D}_2$
{1}
{1}
{d,n}
$\mathcal{D}_3 = \mathcal{D}_1 \ominus \mathcal{D}_2$
{a}
$\mathcal{D}_4$
{1}
{1}
{1}
{1}
{1}
{1}
$\mathcal{D}_5 = \mathcal{D}_4 \ominus \mathcal{D}_2$
{1}
{1}
$\mathcal{D}_6 = \mathcal{D}_3 \bigotimes \mathcal{D}_5$

Figure 6. An example for illustrating the steps of the synthesis algorithm for Example 7. Dash-line nodes are start nodes and double-line nodes are end nodes. $\mathcal{D}_1$ is the DAG learnt from the first (positive) example string `"RJ1"` and $\mathcal{D}_2$ is the DAG learnt from the second (negative) example string `"D1"`. The list of DAGs after the first two interactions is $\widetilde{\mathcal{D}} = [\mathcal{D}_3]$. $\mathcal{D}_4$ is the DAG learnt from the third (positive) example string `"DDD"`. We have $\widetilde{\mathcal{D}} = [\mathcal{D}_3, \mathcal{D}_5]$ after the third interaction. The resulting DAG $\mathcal{D}_6$ is obtained by merging $\mathcal{D}_3$ and $\mathcal{D}_5$. The learning process terminates in three iterations.

$$
\begin{array}{rcl|rcl}
\widetilde{ts} & := & \mathcal{D}(Q, S, F, E, W), & [\![\widetilde{ts}]\!] = & [\![\mathcal{D}]\!] = & \{\texttt{Seq}(t_1, \cdot\cdot, t_n) \mid t_i \in W(e_i), \texttt{where } e_1, \cdot\cdot, e_n \in E \texttt{ form} \\
 & & \texttt{where } W : E \to 2^{\widetilde{T}} & & & \quad \texttt{a path between any } s \in S \texttt{ and any } f \in F\} \\
\widetilde{r} & := & [\mathcal{D}_1, \cdot\cdot, \mathcal{D}_n] & [\![\widetilde{r}]\!] = & [\![\widetilde{\mathcal{D}}]\!] = & \{ts_1 \vee \cdot\cdot \vee ts_n \mid ts_1 \in [\![\mathcal{D}_1]\!], \cdot\cdot, ts_n \in [\![\mathcal{D}_n]\!], \mathcal{D}_i \in \widetilde{\mathcal{D}}, i = 1, \cdot\cdot, n\}
\end{array}
$$

Figure 7. The syntax and semantics of: 1) a DAG data structure $\mathcal{D}$ to succinctly represent a large set of token sequences, and 2) a list of DAGs $\widetilde{\mathcal{D}}$ to represent a large set of disjunctive expressions.

sequences (represented as paths). The syntax and semantics of the DAG data structure is shown in Figure 7. A DAG is represented by a 5-tuple $\mathcal{D}(Q, S, F, E, W)$, where Q is a set of nodes containing two sets of distinctly marked start and end nodes $S \subset Q$ and $F \subset Q$ respectively, E is a set of edges over nodes in Q that induces the DAG, and W maps each edge $e \in E$ to a set of tokens $\tilde{t}$ which is the subset of the token set in the DSL. The set of token sequences represented by a DAG $\mathcal{D}(Q, S, F, E, W)$ includes those token sequences that can be obtained by concatenating tokens along any path (one token for each edge) from a start node $s \in S$ to an end node $f \in F$. A list of DAGs represents a set of disjunctive expressions which are disjunctions of the token sequences represented by the DAGs in the list.

Note that the DAG data structure represents an exponential number of token sequences using polynomial space. The list of DAGs also represents an exponential number of disjunctive expressions. We use the notation $\mathcal{D}$ to denote a DAG and use $\widetilde{\mathcal{D}}$ to denote a list of DAGs.

5.1 DAG Operators

We define two binary operators over the DAG data structure: $\bigotimes$ and $\ominus$. The semantics of the two operators is shown in Figure 8. The $\bigotimes$ operator intersects the sets of token sequences represented by $\mathcal{D}_1$ and $\mathcal{D}_2$ to find the common token sequences, and the $\ominus$ operator removes the token sequences represented by $\mathcal{D}_2$ from those represented by $\mathcal{D}_1$. These operators are used in the synthesis algorithms to handle positive and negative examples.

$$[\![\mathcal{D}_1 \bigotimes \mathcal{D}_2]\!] = [\![\mathcal{D}_1]\!] \cap [\![\mathcal{D}_2]\!] \qquad [\![\mathcal{D}_1 \ominus \mathcal{D}_2]\!] = [\![\mathcal{D}_1]\!] \setminus [\![\mathcal{D}_2]\!]$$

Figure 8. The semantics of $\bigotimes$ and $\ominus$ operators.

5.2 Algorithms for Implementing Operators

The algorithm for the $\bigotimes$ operator for computing token sequences that are common to two DAGs is shown in Figure 10. It constructs the product graph of two DAGs, while at the same time intersecting the token labels on the edges.[2] The complexity[3] of $\bigotimes$ operator is $\mathcal{O}(n^2)$ where n is the number of nodes in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. The result obtained by the algorithm is still a DAG.

$$
\begin{array}{l}
\underline{\mathcal{D} = \mathcal{D}_1 \bigotimes \mathcal{D}_2} \\
Q_{\mathcal{D}} := \{(q_1, q_2) \mid q_1 \in Q_{\mathcal{D}_1}, q_2 \in Q_{\mathcal{D}_2}\} \\
S_{\mathcal{D}} := \{(s_1, s_2) \mid s_1 \in S_{\mathcal{D}_1}, s_2 \in S_{\mathcal{D}_2}\} \\
F_{\mathcal{D}} := \{(f_1, f_2) \mid f_1 \in F_{\mathcal{D}_1}, f_{\mathcal{D}_2} \in F_2\} \\
E_{\mathcal{D}} := \{((q_1, q_2), (q_1', q_2')) \mid (q_1, q_1') \in E_{\mathcal{D}_1}, (q_2, q_2') \in E_{\mathcal{D}_2}\} \\
W_{\mathcal{D}}((q_1, q_2), (q_1', q_2')) := W_{\mathcal{D}_1}((q_1, q_1')) \cap W_{\mathcal{D}_2}((q_2, q_2'))
\end{array}
$$

Figure 10. The algorithm for implementing the $\bigotimes$ operator.

Theorem 1. *Figure 10 implements the semantics of* $\bigotimes$*:* $\mathcal{D} = \mathcal{D}_1 \bigotimes \mathcal{D}_2 \Rightarrow [\![\mathcal{D}]\!] = [\![\mathcal{D}_1]\!] \cap [\![\mathcal{D}_2]\!]$.

Example 8 (DAG intersection)**.** *The result of the intersection* $\mathcal{D}_6 = \mathcal{D}_3 \bigotimes \mathcal{D}_5$ *in Example 7 is shown in Figure 6.*

[2] An extra step to remove the unreachable nodes in the resulting DAG is necessary to reduce the DAG size for efficiency.

[3] We use the number of nodes in the resulting DAG as a measure of the complexity of an algorithm in the following sections of the paper.

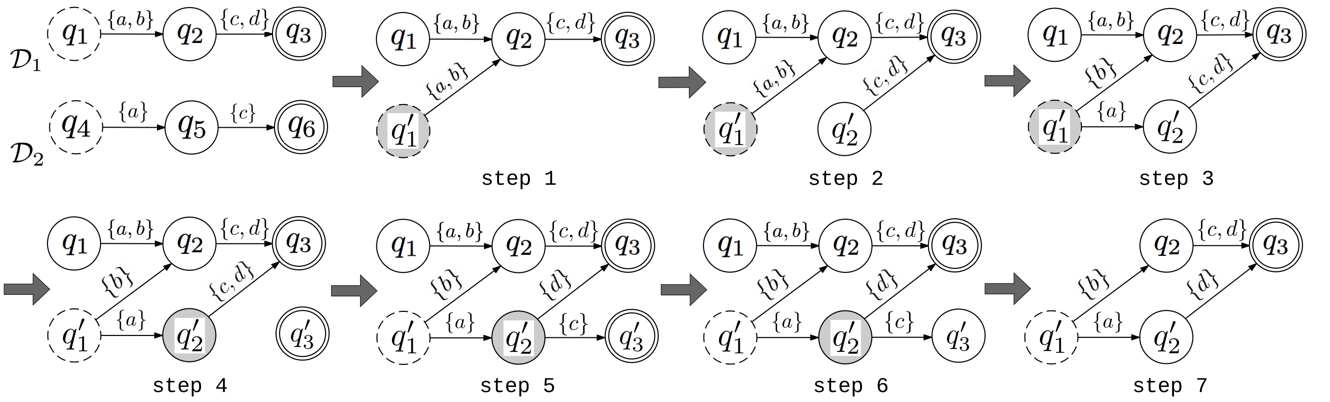


Figure 9. The different steps of the $\ominus$ operator for removing token sequences in DAG $\mathcal{D}_2$ from $\mathcal{D}_1$. The algorithm creates copies of nodes in $\mathcal{D}_1$ and isolates the paths to be removed by splitting the edge tokens between the copied nodes.

The algorithm for the $\ominus$ operator removes token sequences present in DAG $\mathcal{D}_2$ from another DAG $\mathcal{D}_1$. A naïve algorithm to compute the difference would be to first enumerate the token sequences in $\mathcal{D}_1$ and $\mathcal{D}_2$, and then remove the token sequences in $\mathcal{D}_2$ from those in $\mathcal{D}_1$. This algorithm is clearly prohibitively expensive. There are two key ideas in our algorithm to efficiently perform the DAG subtraction: i) it performs a synchronized traversal of paths in $\mathcal{D}_1$ based on corresponding nodes in $\mathcal{D}_2$, and ii) it splits the token sets on edges to remove multiple paths in $\mathcal{D}_2$ simultaneously. These key ideas allow our algorithm to lazily explore only the relevant paths in $\mathcal{D}_1$ based on paths in $\mathcal{D}_2$.

Before describing the algorithm formally, we briefly illustrate the different steps of the algorithm for removing token sequences in $\mathcal{D}_2$ from those in $\mathcal{D}_1$ for a simple example shown in Figure 9. $\mathcal{D}_1$ consists of three nodes q_1, q_2, and q_3, and represents the token sequences {`Seq(a,c)`, `Seq(a,d)`, `Seq(b,c)`, `Seq(b,d)`}, whereas $\mathcal{D}_2$ consists of three nodes q_4, q_5 and q_6, and represents one token sequence {`Seq(a,c)`}. We can observe that the algorithm can not simply remove tokens `a` or `c` from edges (q_1, q_2) or (q_2, q_3) as that would result in removing valid token sequences, such as `Seq(a,d)` or `Seq(b,c)`. The key intuition behind our subtraction algorithm is to recursively isolate the token sequences (paths) that need to be removed from $\mathcal{D}_1$ by creating copies of nodes and edges along those paths. The algorithm only deletes the isolated paths, without affecting other valid token sequences that might share some edges with the paths to be deleted.

First, the algorithm creates a copy of the start node q_1 in $\mathcal{D}_1$ (and it also makes q_1 a non-starting node) to get a new start node q'_1 and copies the corresponding outgoing edges (`step 1`). This results in a new DAG $\mathcal{D}$ that is equivalent to the original DAG in terms of the token sequences being represented. It then iterates the outgoing edges from nodes q'_1 in $\mathcal{D}$ and q_4 in $\mathcal{D}_2$ in a pairwise fashion, say edges (q'_1, q_2) and (q_4, q_5), as a way to synchronously traverse all the paths from nodes q'_1 and q_4. In each iteration, it considers one of the edges (q'_1, q_2) in $\mathcal{D}$, and creates a copy of q_2 to obtain q'_2 and copies the outgoing edges of q_2 as well (`step 2`). It then removes tokens from the edge (q'_1, q_2) that are common to the tokens on the edge (q_4, q_5) in $\mathcal{D}_2$, and adds an edge (q'_1, q'_2) with those common tokens (`step 3`). Finally, the algorithm proceeds recursively by considering outgoing edges of the nodes q'_2 in $\mathcal{D}$ and q_5 in $\mathcal{D}_2$ (`steps 4-5`), and removes the isolated paths in $\mathcal{D}$ by making corresponding end nodes as non-ending nodes (`step 6`, q'_3 becomes a non-ending node since the node q_6 in $\mathcal{D}_1$ is an end node). The DAGs in this example consist of only one start node, but the general algorithm performs this operation over all pairs of start nodes in $\mathcal{D}_1$ and $\mathcal{D}_2$ in order to be sound. The algorithm also handles DAGs with multiple end nodes.

We now formally describe the algorithm for the $\ominus$ operator shown in Figure 11.

```
  $\mathcal{D} = \mathcal{D}_1 \ominus \mathcal{D}_2$
1 $\mathcal{D} :=$ Copy($\mathcal{D}_1$)
2 foreach $s \in S_{\mathcal{D}}, s_2 \in S_{\mathcal{D}_2}$ :
      create a new node $\ddot{s}$ in $\mathcal{D}$
3     $\ddot{s} :=$ new Node(), $Q_{\mathcal{D}} := Q_{\mathcal{D}} \cup \ddot{s}$
      make $\ddot{s}$ a start node and $s$ a non-starting node
4     $S_{\mathcal{D}} := (S_{\mathcal{D}} \setminus \{s\}) \cup \{\ddot{s}\}$
      copy outgoing edges and update edge labels
5     $E_{\mathcal{D}} := E_{\mathcal{D}} \cup \{(\ddot{s}, q) \mid (s, q) \in E_{\mathcal{D}}\}$
6     foreach $q \in Q_{\mathcal{D}} : W_{\mathcal{D}}((\ddot{s}, q)) := W_{\mathcal{D}}((s, q))$
      remove token sequences for $(s, s_2)$
7     SubPartialDAG($\mathcal{D}, \ddot{s}, \mathcal{D}_2, s_2$)
```

Figure 11. The algorithm for implementing the $\ominus$ operator.

The subtraction algorithm first copies $\mathcal{D}_1$ to $\mathcal{D}$ (line 1) and then iterates over all pairs of start nodes (s, s_2) in $\mathcal{D}$ and $\mathcal{D}_2$ (line 2). In each iteration, it creates a fresh node $\ddot{s}$ (line 3), replaces s by $\ddot{s}$ (s is not removed) as the new start node (line 4), connects $\ddot{s}$ to the outgoing nodes of s (line 5) and assigns tokens on outgoing edges of $\ddot{s}$ (line 6). Then it subtracts the partial DAG in $\mathcal{D}_2$ rooted at s_2 from the partial DAG in $\mathcal{D}$ rooted at $\ddot{s}$ by calling the `SubPartialDAG` function (line 7).

Figure 13 shows the algorithm of SubPartialDAG function, the core component of the subtraction algorithm. Given two nodes $q_1 \in Q_{\mathcal{D}_1}$ and $q_2 \in Q_{\mathcal{D}_2}$, it iterates over all pairs of outgoing edges of q_1 and q_2 (line 1). In each iteration for $(q_1, q'_1) \in E_{\mathcal{D}_1}$ and $(q_2, q'_2) \in E_{\mathcal{D}_2}$, it performs the following two steps, which are also illustrated in Figure 12.

```
   SubPartialDAG($\mathcal{D}_1, q_1, \mathcal{D}_2, q_2$)
1  foreach $(q_1, q'_1) \in E_{\mathcal{D}_1}$, $(q_2, q'_2) \in E_{\mathcal{D}_2}$:
       create a new node $\ddot{q}'_1$ in $\mathcal{D}_1$
2      $\ddot{q}'_1$ := new Node(), $Q_{\mathcal{D}_1} := Q_{\mathcal{D}_1} \cup \ddot{q}'_1$
       copy outgoing edges and update edge labels
3      $E_{\mathcal{D}_1} := E_{\mathcal{D}_1} \cup \{(\ddot{q}'_1, q''_1) \mid (q'_1, q''_1) \in E_{\mathcal{D}_1}\}$
4      foreach $q \in Q_{\mathcal{D}_1}$: $W_{\mathcal{D}_1}((\ddot{q}'_1, q)) := W_{\mathcal{D}_1}((q'_1, q))$
       connect $q_1$ and $\ddot{q}'_1$
5      $E_{\mathcal{D}_1} := E_{\mathcal{D}_1} \cup \{(q_1, \ddot{q}'_1)\}$
       update edge labels
6      $W_{\mathcal{D}_1}((q_1, \ddot{q}'_1)) := W_{\mathcal{D}_1}((q_1, q'_1)) \cap W_{\mathcal{D}_2}((q_2, q'_2))$
7      $W_{\mathcal{D}_1}((q_1, q'_1)) := W_{\mathcal{D}_1}((q_1, q'_1)) \setminus W_{\mathcal{D}_2}((q_2, q'_2))$
       mark $\ddot{q}'_1$ as an end node
8      if $q'_1 \in F_{\mathcal{D}_1}$: $F_{\mathcal{D}_1} := F_{\mathcal{D}_1} \cup \{\ddot{q}'_1\}$
       mark $\ddot{q}'_1$ as a non-ending node (delete paths)
9      if $q'_2 \in F_{\mathcal{D}_2}$: $F_{\mathcal{D}_1} := F_{\mathcal{D}_1} \setminus \{\ddot{q}'_1\}$
       remove token sequences for $(\ddot{q}'_1, q'_2)$
10     SubPartialDAG($\mathcal{D}_1, \ddot{q}'_1, \mathcal{D}_2, q'_2$)
```

Figure 13. The algorithm for SubPartialDAG function.

- Path isolation (lines 2-8): It copies the node q'_1 to $\ddot{q}'_1$ (line 2) together with the outgoing edges of q'_1 and token labels on those edges (lines 3-4) and obtains $\mathcal{D}'_1$. It also adds an edge $(q_1, \ddot{q}'_1)$ in $\mathcal{D}'_1$ (line 5). Then, the algorithm splits the original token set $W_{\mathcal{D}_1}((q_1, q'_1))$ on edge (q_1, q'_1) into two token sets: $W_{\mathcal{D}_1}((q_1, q'_1)) \cap W_{\mathcal{D}_2}((q_2, q'_2))$ and $W_{\mathcal{D}_1}((q_1, q'_1)) \setminus W_{\mathcal{D}_2}((q_2, q'_2))$, and it assigns the first token set on edge $(q_1, \ddot{q}'_1)$ (line 6) and replaces the token set on edge (q_1, q'_1) by the second one (line 7). The algorithm sets $\ddot{q}'_1$ as an end node if q'_1 is an end node in $\mathcal{D}_1$ (line 8). This step is also shown in Figure 12. Intuitively in this step it isolates one edge on the path that needs to be removed, and we have $\llbracket \mathcal{D}'_1 \rrbracket = \llbracket \mathcal{D}_1 \rrbracket$ after this isolation step.
- Path deletion (line 9): In this step, if q'_2 is an end node in $\mathcal{D}_2$, the algorithm removes from $\mathcal{D}_1$ the paths in $\mathcal{D}_2$ that ends at q_2 by making $\ddot{q}'_1$ a non-ending node (line 9), otherwise, it does not do anything. Figure 12 shows this step for the case where q'_2 is an end node.

Finally, the algorithm recurses on $\ddot{q}'_1$ and q'_2 (line 10) until it reaches the termination condition where at least a node has no outgoing edges. [4]

Theorem 2. *Figure 11 implements the semantics of* $\ominus$: $\mathcal{D} = \mathcal{D}_1 \ominus \mathcal{D}_2 \Rightarrow \llbracket \mathcal{D} \rrbracket = \llbracket \mathcal{D}_1 \rrbracket \setminus \llbracket \mathcal{D}_2 \rrbracket$.

Example 9 (DAG subtraction)**.** *The results of the subtractions* $\mathcal{D}_3 = \mathcal{D}_1 \ominus \mathcal{D}_2$ *(iteration* 2*) and* $\mathcal{D}_5 = \mathcal{D}_4 \ominus \mathcal{D}_2$ *(iteration* 3*) in Example 7 are shown in Figure 6.*

Relationship to NFA The DAG representation in FIDEX can be considered as a form of non-deterministic finite automata (NFA)[5] with tokens as edge labels. The algorithms for implementing $\otimes$ and $\ominus$ operators can be regarded as computing the intersection and difference over two NFAs. Note that DAGs are less general than NFAs because: i) they don't have epsilon transitions, and ii) there are no cycles.

For implementing the $\otimes$ operator we borrowed the idea from NFA intersection algorithm and achieve the optimal complexity $\mathcal{O}(n^2)$. For implementing the $\ominus$ operator, however, we developed a novel algorithm in which we perform synchronized traversal on both DAGs. This is different from the traditional algorithm for computing the NFA difference in which it intersects the first NFA with the complement of the second NFA. Since complementing an NFA requires a determinization step, the complexity is exponential. However, by the synchronized traversal, the intermediate DAGs computed by our algorithm are sparse, i.e., the out-degree of most nodes is small (close to 1) and this leads to better practical complexity. In the worst case, the complexity of the SubPartialDAG algorithm is $\mathcal{O}(n^2)$ (for dense graphs), and the number of nodes in the DAG after each iteration in the $\ominus$ algorithm grows in the following manner: $n, n^2, \cdot\cdot, n^{n^2+1}$ (at most n^2 iterations), which leads to the worst-case complexity of $\mathcal{O}(n^n)$. However, the intermediate graphs are usually sparse, which causes the number of node copies to be small during the synchronized traversal of SubPartialDAG (a node in $\mathcal{D}_1$ is copied for a few nodes in $\mathcal{D}_2$). The size of a DAG grows *almost linearly* after each iteration. We show this phenomenon of $\mathcal{O}(n)$ complexity empirically in §7.7.

Our subtraction algorithm is also different from the antichain method [42]. The anti-chain method exploits the special structure of the subset construction by treating set inclusion as a partial order, which also turns out to be a simulation relation for reachability problems. These anti-chain techniques are proven to be theoretically sound and also efficient in practice on large random automata. However, these techniques have been only developed for binary properties such as language inclusion, emptiness, and universality. We instead need a method to perform efficient language subtraction over two DAGs and we need to compute and keep the final resulting DAG after the subtraction. Moreover, the subtraction algorithm in FIDEX does not use the notion of partial order over sets of states to prune the states during the operation, instead it does a synchronized pass over two DAGs to copy nodes and remove edges from the first DAG (while synchronizing with the second DAG) to perform subtrac-

[4] We note three important optimizations for an efficient implementation: 1). caching intermediate nodes to avoid unnecessary node copies, 2). immediately removing edges with empty token sets, and 3). periodically cleaning unreachable nodes in the DAG.

[5] Recall that a DAG can have multiple start nodes.

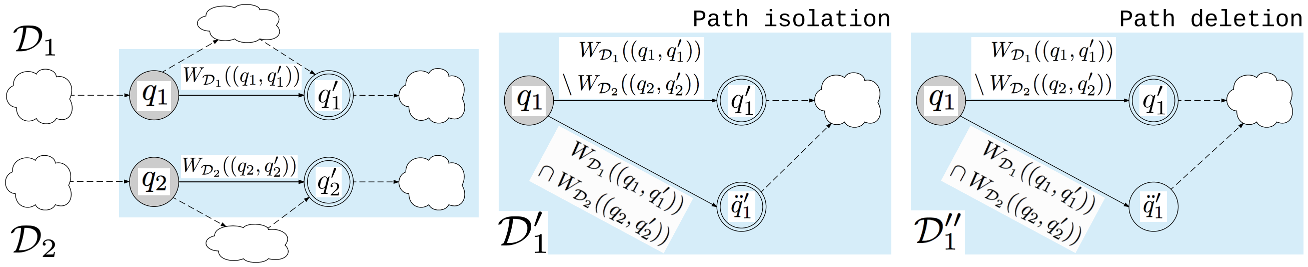

$\mathcal{D}_1$
$W_{\mathcal{D}_1}((q_1,q_1'))$
q_1
q_1'
q_2
q_2'
$W_{\mathcal{D}_2}((q_2,q_2'))$
$\mathcal{D}_2$
Path isolation
$W_{\mathcal{D}_1}((q_1,q_1'))$
$\setminus W_{\mathcal{D}_2}((q_2,q_2'))$
$W_{\mathcal{D}_1}((q_1,q_1'))$
$\cap W_{\mathcal{D}_2}((q_2,q_2'))$
$\ddot{q}_1'$
$\mathcal{D}_1'$
Path deletion
$\mathcal{D}_1''$

Figure 12. The SubPartialDAG($\mathcal{D}_1$,q_1,$\mathcal{D}_2$,q_2) function.

tion. The practical efficiency of our subtraction algorithm exploits the property of our DAG data structure (sparsity in edges) that arises from the practical filter expression synthesis benchmarks.

6. Synthesis Algorithms

In this section, we present a sound and complete synthesis algorithm for incrementally learning a Filter expression in the DSL that is consistent with the given list of positive and negative strings. We first describe the synthesis algorithms for learning consistent token sequences and consistent disjunctive expressions for a given predicate, respectively. We then present an *incremental* algorithm for learning consistent disjunctive expressions for a given predicate. Finally, we describe the *incremental* learning algorithm for learning consistent Filter expressions using a ranking algorithm.

6.1 Overview of the Synthesis Algorithm

We first present a high-level overview of the synthesis algorithm as shown in Figure 14. Given a set of positive example strings S^+ and negative example strings S^-, the LearnFilter algorithm learns a filter expression in the DSL that is consistent with the examples. It first learns the set of all filter expressions $\tilde{f}$ that are consistent with the first positive example string $S^+[0]$, and uses a ranking algorithm Rank to find a filter expression $f \in \tilde{f}$. It then returns f as the desired filter expression if f is consistent with all the positive and negative examples. Otherwise, the algorithm finds a counterexample string s that is not consistent with f, and then refines the set of filter expressions such that they are consistent with the counterexample string s in addition to the previous examples. This iterative process continues until it finds a consistent filter expression (or fails to find one).

There are several technical challenges in the LearnFilter algorithm for efficiently learning the desired filter expression. First, we need an efficient LearnFilterExpr algorithm to compute (as well as represent) a large set of filter expressions in the DSL that are consistent with a given example string. Second, we need an efficient ranking algorithm Rank to efficiently compute the most likely filter expression amongst the large set of consistent filter expressions. Third, we need to develop algorithms to refine the set of consistent filter expressions with additional positive and negative example strings. Finally, we need to develop incremental algorithms to efficiently incorporate new examples and compute the set of consistent filter expressions.

```
LearnFilter($S^+, S^-$)
  Learn consistent filters with $1^{st}$ positive string
  $\tilde{f}$ := LearnFilterExpr($S^+[0]$)
  Select a top-ranked filter expression
  $f$ := Rank($\tilde{f}$)
  Find a counterexample string inconsistent with f
  while ($\exists s \in S^+ : \neg[\![f]\!]s \vee \exists s \in S^- : [\![f]\!]s$)
      Refine the set of filters with a counterexample
      $\tilde{f}$ := RefineFilterExpr($\tilde{f}, s$)
      if $\tilde{f} = \emptyset$: return null
      $f$ := Rank($\tilde{f}$)
  return $f$
```

Figure 14. An overview of the top-level synthesis algorithm for learning a filter expression consistent with a set of positive and negative examples.

6.2 Preliminaries

We first define the notion of the consistency of Filter expressions with a given list of positive and negative strings and describe the GenDAG algorithm for learning a DAG $\mathcal{D}$ from a string s for different predicates.

Definition 1 (Consistency)**.** *Given a list of strings* $L = [s_1^+, \cdot\cdot, s_n^+, s_1^-, \cdot\cdot, s_m^-]$ *in which* s_i^+ *is a positive string and* s_j^- *is a negative string, we define a* Filter *expression* f *to be consistent with* L *if we have* $[\![f(p, L)]\!]\ \sigma = [s_1^+, \cdot\cdot, s_n^+]$, *i.e. if the predicate* p *is only true for the positive strings* s_i^+ *and false for all negative strings* s_j^-.

GenDAG Algorithm: Given a string s and a predicate Pred in the DSL, we use the GenDAG algorithm shown in Figure 15 to learn the set of all token sequences for Pred (represented succinctly using a DAG $\mathcal{D}$) such that the Pred expression with any token sequences in $\mathcal{D}$ matches string s. The key idea of the GenDAG algorithm for a string s is to construct a DAG $\mathcal{D}$, where the nodes represent different indices of the string and the edges denote the tokens that are consistent with different substrings of s. For example, for the

StartsWith predicate and a string s, the $\texttt{GenDAG}_{\texttt{StartsWith}}$ algorithm first constructs $|s|+1$ number of nodes $Q := \{0, \cdot\cdot, |s|\}$ and assigns the set of start nodes and end nodes as $S_1 := \{0\}$ and $F_1 := \{1, \cdot\cdot, |s|\}$, respectively. It then adds an edge (i, j) between each pair of nodes i and j such that $0 \leq i < j \leq |s|$ and labels each edge (i, j) with a set of tokens $W((i, j))$ each of which matches the substring $s[i, j]$ but not any substring $s[i, k]$, $k > j$. For other predicates, the algorithm learns the DAG similarly as shown in Figure 15. The only difference is in terms of assigning the set of start nodes and end nodes. The running time complexity of the GenDAG is $\mathcal{O}(k \cdot n^2)$, where k is the size of the token set $\widetilde{T}$ in the DSL and $n = |s|$.

$$
\begin{aligned}
&Q := \{0, \cdot\cdot, |s|\}\\
&E = \{(i, j) \mid 0 \leq i < j \leq |s|)\}\\
&W((i, j)) := \{T \mid \texttt{Token } T \in \widetilde{T} \texttt{ matches } s[i, j] \wedge \nexists k > j\\
&\qquad\qquad\qquad\qquad \texttt{s.t. } T \texttt{ matches } s[i, k]\} \quad \forall (i, j) \in E\\
&S_1 := \{0\},\ S_2 := \{0, \cdot\cdot, |s| - 1\}\\
&F_1 := \{1, \cdot\cdot, |s|\},\ F_2 := \{|s|\}\\
&\texttt{GenDAG}_{\texttt{StartsWith}}(s) := \mathcal{D}(Q, S_1, F_1, E, W)\\
&\texttt{GenDAG}_{\texttt{EndsWith}}(s) := \mathcal{D}(Q, S_2, F_2, E, W)\\
&\texttt{GenDAG}_{\texttt{Matches}}(s) := \mathcal{D}(Q, S_1, F_2, E, W)\\
&\texttt{GenDAG}_{\texttt{Contains}}(s) := \mathcal{D}(Q, S_2, F_1, E, W)
\end{aligned}
$$

Figure 15. The GenDAG algorithm for learning the DAG that succinctly represents all token sequences conforming to a given string s for different predicates.

Lemma 1. *The GenDAG algorithm is sound and complete, i.e., for a predicate* Pred *and a string $s = \sigma(v)$, it learns a DAG $\mathcal{D}$ such that i) $\forall$ token sequence $ts \in [\![\mathcal{D}]\!]$ we have $[\![\texttt{Pred}(v, ts)]\!]\ \sigma = \texttt{True}$, and ii) if there exists a token sequence ts in the DSL for a predicate* Pred *such that $[\![\texttt{Pred}(v, ts)]\!]\ \sigma = \texttt{True}$, then we have $ts \in [\![\mathcal{D}]\!]$.*

Example 10 (DAG generation in the running example)**.** *The generated DAGs $\mathcal{D}_1, \mathcal{D}_2$ and $\mathcal{D}_4$ for strings "RJ1" (iteration 1), "D1" (iteration 2) and "DDD" (iteration 3) in Example 7 are shown in Figure 6.*

6.3 Learning Token Sequences

We first present the $\texttt{LearnTokenSeqs}_{\texttt{Pred}}$ algorithm, shown in Figure 16, for learning the set of token sequences in the DSL for the predicate Pred that are consistent with a given list of positive (S^+) and negative (S^-) strings. The key idea of the algorithm to first learn the DAGs for each example string using the $\texttt{GenDAG}_{\texttt{Pred}}$ algorithm. For the positive example strings, the algorithm intersects the DAGs using the $\bigotimes$ operator to compute a DAG consisting of all consistent token sequences, and then uses the $\ominus$ algorithm to remove token sequences present in the DAGs for the negative strings to compute the resulting DAG.

The algorithm first learns a DAG $\mathcal{D}$ for the first positive string $S^+[0]$ using the corresponding $\texttt{GenDAG}_{\texttt{Pred}}$ algorithm. It then learns a DAG $\mathcal{D}^+$ for every other positive string in S^+ and applies the $\bigotimes$ operator on $\mathcal{D}^+$ and $\mathcal{D}$ to obtain the common token sequences for Pred. The resulting DAG $\mathcal{D}$ now represents the set of all token sequences for a predicate Pred expression that are consistent with the list of positive strings S^+. The algorithm then learns a DAG $\mathcal{D}^-$ for each negative string and subtracts the token sequences in $\mathcal{D}^-$ from those in $\mathcal{D}$ using the $\ominus$ operator. The resulting DAG $\mathcal{D}$ therefore consists of all token sequences for Pred that are consistent with $[S^+, S^-]$.

```
LearnTokenSeqs_Pred($S^+, S^-$)
$\mathcal{D}$ := GenDAG_Pred($S^+[0]$)
for $i = 1$ to size($S^+$) $- 1$:
    $\mathcal{D}^+$ := GenDAG_Pred($S^+[i]$)
    $\mathcal{D} := \mathcal{D} \bigotimes \mathcal{D}^+$
foreach $s \in S^-$:
    $\mathcal{D}^-$ := GenDAG_Pred($s$)
    $\mathcal{D} := \mathcal{D} \ominus \mathcal{D}^-$
return $\mathcal{D}$
```

Figure 16. The LearnTokenSeqs algorithm to learn all token sequences for a predicate Pred to be consistent with the given positive strings S^+ and negative strings S^-.

Definition 2 (Soundness)**.** *We define a synthesis algorithm to be sound for a DSL fragment if the expressions in the DSL fragment learnt by the algorithm are all consistent with the given list of positive strings (S^+) and negative strings (S^-).*

Definition 3 (Completeness)**.** *We define a synthesis algorithm to be complete for a DSL fragment if the synthesis algorithm is guaranteed to learn an expression in the DSL fragment that is consistent with the given list of positive strings (S^+) and negative strings (S^-) whenever there exists such an expression in that DSL fragment.*

Theorem 3. *The $\texttt{LearnTokenSeqs}_{\texttt{Pred}}$ algorithm is sound and complete for the non-disjunctive predicate expression fragment in the DSL.*

6.4 Learning Disjunctive Expressions

The LearnTokenSeqs algorithm described in §6.3 can only learn the set of *token sequences* consistent with a given set of examples. For learning the *disjunctive expressions* in the DSL, we present the $\texttt{LearnDisjExprs}_{\texttt{Pred}}$ algorithm shown in Figure 17. The key idea of the algorithm is to maintain a list of DAGs to represent a set of disjunctive expressions. It first constructs a list of DAGs consisting of individual DAGs for each positive string, and then uses the DAGs for negative example strings to remove the token sequences individually from each positive DAG in the list. It finally merges the DAGs in the resulting list to minimize the number of disjunctions in the resulting expression.

The algorithm first learns a DAG $\mathcal{D}^+$ for each positive string in S^+ and stores them in a list of DAG structures $\widetilde{\mathcal{D}}$. It then learns a DAG $\mathcal{D}^-$ for each negative string in

```
LearnDisjExprs_Pred($S^+, S^-$)
$\widetilde{\mathcal{D}} = [\ ]$
foreach $s \in S^+$:
    $\mathcal{D}^+ :=$ GenDAG_Pred($s$)
    $\widetilde{\mathcal{D}}$.Append($\mathcal{D}^+$)
foreach $s \in S^-$:
    $\mathcal{D}^- :=$ GenDAG_Pred($s$)
    for $i = 0$ to size($\widetilde{\mathcal{D}}$) $- 1$:
        $\widetilde{\mathcal{D}}[i] := \widetilde{\mathcal{D}}[i] \ominus \mathcal{D}^-$
        if $\widetilde{\mathcal{D}}[i] = \emptyset$: return [ ]
return MergeDAGs($\widetilde{\mathcal{D}}$)
```

Figure 17. The LearnDisjExprs algorithm for learning all disjunctive expressions for a predicate Pred to be consistent with the given positive strings S^+ and negative strings S^-.

S^- and subtracts $\mathcal{D}^-$ from each DAG $\widetilde{\mathcal{D}}[i] \in \widetilde{\mathcal{D}}$. If there is any empty DAG in the resulting list of DAGs $\widetilde{\mathcal{D}}$, the algorithm returns [ ] denoting that there does not exist any disjunctive expression in the DSL for the predicate Pred that is consistent with $[S^+, S^-]$. Otherwise, it returns the result of merging the list of DAGs $\widetilde{\mathcal{D}}$, which represents all consistent disjunctive expressions for the predicate Pred.

The MergeDAGs algorithm, as shown in Figure 18, greedily merges the DAGs in $\widetilde{\mathcal{D}}$ into partitions such that the intersection of DAGs in any partition is non-empty, to reduce the number of disjunctions in the final expressions.

```
MergeDAGs($\widetilde{\mathcal{D}}$)
$\widetilde{\mathcal{D}}_{\texttt{res}} := [\ ]$, $\widetilde{\mathcal{D}}_{\texttt{res}}[0] := \widetilde{\mathcal{D}}[0]$
foreach $\mathcal{D} \in \widetilde{\mathcal{D}}$:
    if $\exists\ j : 0 \le j <$ size($\widetilde{\mathcal{D}}_{\texttt{res}}$) $\wedge\ \widetilde{\mathcal{D}}_{\texttt{res}}[j] \bigotimes \mathcal{D} \neq \emptyset$:
        $\widetilde{\mathcal{D}}_{\texttt{res}}[j] := \widetilde{\mathcal{D}}_{\texttt{res}}[j] \bigotimes \mathcal{D}$
    else: $\widetilde{\mathcal{D}}_{\texttt{res}}$.Append($\mathcal{D}$)
return $\widetilde{\mathcal{D}}_{\texttt{res}}$
```

Figure 18. The MergeDAGs algorithm for greedily finding the largest partitions of a list of DAGs $\widetilde{\mathcal{D}}$.

Theorem 4. *The LearnDisjExprs$_{Pred}$ algorithm is sound and complete for all predicate expressions in the DSL.*

Example 11 (DAG merging in the running example)**.** *The DAG merging (intersection) $\mathcal{D}_6 = \mathcal{D}_3 \bigotimes \mathcal{D}_5$ in the third iteration in Example 7 is shown in Figure 6.*

6.5 Learning Disjunctive Expressions Incrementally

The LearnDisjExprs algorithm described previously in §6.4 learns the set of disjunctive expressions in the DSL that are consistent with a given set of positive and negative examples. A big issue with this algorithm is that it assumes the set of all positive and negative examples are provided a priori. If we use this algorithm in the LearnFilter algorithm for refining the learnt set of expressions with counterexample strings, it will result in an inefficient algorithm as the LearnDisjExprs algorithm starts from scratch on every invocation. Moreover, for FIDEX's application in Excel, a user might provide examples interactively by inspecting the intermediate results instead of providing the complete list of positive and negative strings at once. This motivates the need of an *incremental* learning algorithm for efficiently learning the set of consistent disjunctive expressions.

```
LearnDisjExprsInc_Pred($\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-, s$)
$\mathcal{D} :=$ GenDAG_Pred($s$)
if IsPosStr($s$):
    foreach $\mathcal{D}^- \in \widetilde{\mathcal{D}}^-$:
        $\mathcal{D} := \mathcal{D} \ominus \mathcal{D}^-$
        if $\mathcal{D} := \emptyset$: return ([ ], $\widetilde{\mathcal{D}}^-$)
    $\widetilde{\mathcal{D}}$.Append($\mathcal{D}$)
else: // IsNegStr($s$)
    for $i = 0$ to size($\widetilde{\mathcal{D}}$) $- 1$:
        $\widetilde{\mathcal{D}}[i] := \widetilde{\mathcal{D}}[i] \ominus \mathcal{D}$
        if $\widetilde{\mathcal{D}}[i] := \emptyset$: return ([ ], $\widetilde{\mathcal{D}}^-$)
    $\widetilde{\mathcal{D}}^-$.Append($\mathcal{D}$)
return ($\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-$)
```

Figure 19. The LearnDisjExprsInc algorithm to incrementally learn all disjunctive expressions for a predicate Pred to be consistent with a new string s.

We present an incremental synthesis algorithm in Figure 19 for learning all disjunctive expressions in the DSL for a predicate Pred that are consistent with an example string s and all previous examples. The key idea of the algorithm is to maintain two lists of DAGs: 1) $\widetilde{\mathcal{D}}$ to store all the disjunctive expressions such that the predicate expression Pred with any of those disjunctive expressions is consistent with all positive and negative strings in the past, and 2) $\widetilde{\mathcal{D}}^-$ consisting of DAGs for each negative string in the past. The algorithm uses these lists of DAGs to incrementally refine the set of filter expressions given a new example string.

If the current input string s is a positive example, the algorithm learns a DAG $\mathcal{D}$ for string s and subtracts each DAG $\mathcal{D}^- \in \widetilde{\mathcal{D}}^-$ from $\mathcal{D}$. If the resulting DAG $\mathcal{D}$ is empty, it returns the empty list as the result denoting no desired disjunctive expression for Pred exists. Otherwise, it updates the current list of DAGs $\widetilde{\mathcal{D}}$ by appending $\mathcal{D}$ to $\widetilde{\mathcal{D}}$. If the current input string s is a negative example, it updates the current $\widetilde{\mathcal{D}}$ by subtracting $\mathcal{D}$ from each $\mathcal{D}' \in \widetilde{\mathcal{D}}$. If any DAG in $\widetilde{\mathcal{D}}$ becomes empty, it returns the empty list. Otherwise, it updates $\widetilde{\mathcal{D}}^-$ by appending $\mathcal{D}^-$ to it.

The final set of consistent expressions would be the same for incremental and non-incremental algorithms, but the expression returned to the user and running time would depend on the order of examples as we might get different DAGs representing the same set of expressions.

Theorem 5. *The* LearnDisjExprsInc$_{Pred}$ *algorithm is sound and complete for all predicate expressions in the DSL.*

Example 12 (Incremental learning in the running example)**.** *The DAG lists* $\widetilde{\mathcal{D}} = [\mathcal{D}_3, \mathcal{D}_5]$ *and* $\widetilde{\mathcal{D}}^- = [\mathcal{D}_2]$ *learnt in the third iteration in Example 7 are shown in Figure 6.*

6.6 Ranking Disjunctive Expressions

A list of DAGs $\widetilde{\mathcal{D}}$ succinctly represents a large number of disjunctive expressions and we use ranking to efficiently select the most likely one amongst them. The ranking algorithm RankDAG for finding the top ranked token sequence in a DAG $\mathcal{D}$ is shown in Figure 20, whereas the ranking algorithm RankDAGs for finding the top ranked disjunctive expression in a list of DAGs $\widetilde{\mathcal{D}}$ is shown in Figure 21.

```
RankDAG($\mathcal{D}$)
foreach $e \in E_{\mathcal{D}}$:
    maxTok[$e$] := $\arg\max_T\{$score($T$) $\mid T \in W(e)\}$
    Each token has an empirically predefined score
    score[$e$] := score(maxTok[$e$])
foreach $q \in Q_{\mathcal{D}}$:
    maxEdge[$q$] := $\arg\max_e\{$score[$e$] $\mid e = (q, q') \in E_{\mathcal{D}}\}$
    Get the target (second) node of the edge maxEdge[$q$]
    nextNode[$q$] := maxEdge[$q$]._2
    score[$q$] := score(maxEdge[$q$])
$q_c := \arg\max_s\{$score[$s$] $\mid s \in S_{\mathcal{D}}\}$
AvgScore := score[$q_c$], path := [$q_c$]
while($q_c \notin F_{\mathcal{D}} \vee$ AvgScore $<$ $\frac{\text{size(path)}\cdot\text{AvgScore}+\text{score}[q'_c]}{\text{size(path)}+1}$)
        $q'_c$ := nextNode[$q_c$]
        AvgScore := $\frac{\text{size(path)}\cdot\text{AvgScore}+\text{score}[q'_c]}{\text{size(path)}+1}$
        path.Append($q'_c$)
        $q_c := q'_c$
$ts := \epsilon$
foreach $q \in$ path:
    $T$ := maxEdge[maxTok[$q$]]
    $ts$ := Seq($T, ts$)
return $ts$
```

Figure 20. The ranking algorithm RankDAG for efficiently identifying the top ranked token sequence in a DAG $\mathcal{D}$. The $\arg\max_T\{$score(T) $\mid T \in W(e)\}$ returns the element T with the maximum score(T) value.

```
RankDAGs($\widetilde{\mathcal{D}}$)
$r := \epsilon$
foreach $\mathcal{D} \in \widetilde{\mathcal{D}}$:
    $ts$ := RankDAG($\mathcal{D}$)
    $r$ := Disjunct($ts, r$)
return $r$
```

Figure 21. The ranking algorithm for efficiently identifying the top ranked disjunctive expression in a list of DAGs $\widetilde{\mathcal{D}}$.

The key idea of the ranking algorithm in Figure 20 is to find a path from one of the start nodes in $\mathcal{D}$ to one of the end nodes such that the average score of that path is maximized *locally*. The scores for tokens are assigned empirically according to their generality, i.e., a general token has a higher score than a constant token. The algorithm uses a greedy strategy to find a path in $\mathcal{D}$ by selecting nodes based on their ranks starting from the set of start nodes S. It stops the search when the path ends at an end node f and when there is no improvement on the average score AvgScore by taking an outgoing edge from f. It finally returns the token sequence r corresponding to tokens on the edges along the path. The complexity of RankDAG is $\mathcal{O}(n \cdot k)$ where n is the number of edges in $\mathcal{D}$ and k denotes size of the token set $\widetilde{T}$ in the DSL.

The RankDAGs algorithm applies the RankDAG algorithm on each DAG $\mathcal{D} \in \widetilde{\mathcal{D}}$ and disjoins the selected token sequences to get the top ranked disjunctive expression in $\widetilde{\mathcal{D}}$. The complexity is $\mathcal{O}(n \cdot k \cdot l)$ where $l = \text{size}(\widetilde{\mathcal{D}})$.

6.7 Learning Filter Expressions Incrementally

We now have all the components for describing the top-level LearnFilter algorithm for learning a Filter expression that is consistent with a given set of positive and negative strings as shown in Figure 22. The algorithm learns the Predicate expression in the Filter expression by iterating over all available predicates in the DSL in a certain order: $\{$StartsWith, EndsWith, Matches, Contains$\}$ and learning a disjunctive expression for the predicate expression.

```
LearnFilter($S^+$, $S^-$)
1  foreach Pred $\in \{$StartsWith, EndsWith, Matches, Contains$\}$
2      ($\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-$) := LearnDisjExprsInc$_{\text{pred}}$([ ], [ ], $S^+[0]$)
3      if $\widetilde{\mathcal{D}}$ = [ ]: continue
4      $r$ := RankDAGs(MergeDAGs($\widetilde{\mathcal{D}}$))
5      $p$ := Pred($v, r$)
6      while ($\exists s \in S^+ : \neg[\![p]\!]s \vee \exists s \in S^- : [\![p]\!]s$)
7          ($\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-$) := LearnDisjExprsInc$_{\text{Pred}}$($\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-, s$)
8          if $\widetilde{\mathcal{D}}$ = [ ]: break
9          $r$ := RankDAGs(MergeDAGs($\widetilde{\mathcal{D}}$))
10         $p$ := Pred($v, r$)
11     if $\widetilde{\mathcal{D}} \neq$ [ ]: return Filter($p, L$)
12 return null
```

Figure 22. The LearnFilter algorithm for learning a filter expression in the DSL that is consistent with positive strings S^+ and negative strings S^-.

For a predicate Pred, it first learns all disjunctive expressions represented by $\widetilde{\mathcal{D}}$ which are consistent with the first positive string $S^+[0]$ using the LearnDisjExprsInc$_{\text{Pred}}$ algorithm (line 2) and uses the MergeDAGs and RankDAGs algorithms to compute the corresponding predicate expression p (lines 4-5). It then looks for a counterexample string s in $[S^+, S^-]$ such that either p does not match a positive string $s \in S^+$ or p matches a negative string $s \in S^-$ (line 6). If no such counterexample exists, the algorithm returns the filter expression Filter(p, L) (line 11). Otherwise, the algorithm continues the learning procedure by giving that counterexample as a new example to LearnDisjExprsInc$_{\texttt{Pred}}$ (lines 6-10) until it finds a predicate expression p that is consistent with all the example strings (line 11) or it fails to find a consistent predicate expression (line 12).

Note that the counterexample strings in the while loop (lines 6-10) in Figure 22 can also be provided by a user. The FIDEX system supports such an interface where the user first provides a positive example and a set of test strings. The system learns the corresponding Filter expression and returns the result of matching the learnt expression on the test strings. The user can then inspect the result and provide additional positive/negative examples. FIDEX then updates the Filter expression to be also consistent with the additional examples, and this interaction repeats until the result is as desired by the user or the FIDEX system realizes there does not exists a consistent expression in the DSL.

Theorem 6. *The LearnFilter algorithm is sound and complete for the Filter expression in the DSL.*

An example execution of LearnFilter: The intermediate DAGs computed during the LearnFilter algorithm for the filter task in Example 7 are shown in Figure 6.

Iteration 1: The DAG list for the first (positive) example "RJ1" is $\widetilde{\mathcal{D}} = [\mathcal{D}_1]$ and the DAG list for negative examples is $\widetilde{\mathcal{D}}^- = [\,]$ (line 2). The top-ranked disjunctive expression <a> is computed at line 5 and the corresponding predicate expression StartsWith$(v,$ <a>$)$ is constructed at line 5.

Iteration 2: When StartsWith$(v,$ <a>$)$ is applied on all the input strings (line 6), it also matches string "D1" which is further given as the second (negative) example. After invoking LearnDisjExprsInc$_{\texttt{StartsWith}}$, we have $(\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-) = ([\mathcal{D}_3], [\mathcal{D}_2])$ (line 7). The predicate with the top-ranked disjunctive expression, StartsWith$(v,$ Seq(<l>,<l>,<n>)$)$, is computed at lines 9-10.

Iteration 3: When StartsWith$(v,$ Seq(<l>,<l>,<n>)$)$ is applied on all input strings (line 6), it does not recognize string "DDD" which is further given as the third (positive) example. After invoking LearnDisjExprsInc$_{\texttt{StartsWith}}$ (line 7), we have $(\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^-) = ([\mathcal{D}_3, \mathcal{D}_5], [\mathcal{D}_2])$. At line 9, the algorithm performs $\mathcal{D}_6 = \mathcal{D}_3 \bigotimes \mathcal{D}_5$ to merge DAGs $\mathcal{D}_3$ and $\mathcal{D}_5$ in $\widetilde{\mathcal{D}}$ since they share token sequences in common, selects the top-ranked disjunctive expression from $\mathcal{D}_6$, and then it constructs the corresponding predicate expression StartsWith$(v,$ Seq(<l>,<l>)$)$ at line 10. This expression can perform the desired filtering task (line 6) and the algorithm returns Filter(StartsWith$(v,$ Seq(<l>,<l>)$)$) as the final filter expression to the user at line 11.

7. Experiments

We implemented FIDEX in C# (and also as an add-in for Microsoft Excel) and report its evaluation on 460 benchmarks. The experiments were performed on a quad-core Intel Core i7 2.67GHz CPU with 6GB RAM.

7.1 Benchmarks

We have collected 460 benchmarks in total, among which 334 benchmarks were collected from online help forums and Microsoft Excel team, and the other 126 benchmarks were provided by Microsoft Bing team. Each benchmark consists of a set of strings $\mathcal{U}$ and a set of positive strings $\mathcal{S}$ where we have $\mathcal{S} \subseteq \mathcal{U}$ (the strings in $\mathcal{U} \setminus \mathcal{S}$ are negative strings). A data filtering task is to learn a filter expression to select exactly $\mathcal{S}$ out of $\mathcal{U}$.

The benchmarks on average consist of 47.4 positive string instances and 470.1 negative string instances, and the maximum number of positive and negative strings for any benchmark is 24961 and 3195, respectively. The average length of each positive and negative string is 68.8 and 54.2, and the maximum length is 3485 and 3323, respectively.

7.2 Experimental Setup

We evaluate FIDEX using the incremental learning algorithm LearnFilter (with all four predicates) shown in Figure 22. We also evaluate FIDEX using one predicate to compare the performance of different predicates with one another. FIDEX uses a predefined token set of 114 tokens that includes both general tokens and constant tokens.

We compare FIDEX with the following baseline strategies: (1) **NoDisj:** A variant of FIDEX that learns expressions (represented in a DAG) with only token sequences (without disjunctions). (2) **Enum:** A variant of FIDEX that learns disjunctive expressions by enumerating all the token sequences (without using a DAG representation). We implemented 5 different variants Enum-k (for $k \in \{3,6,9,12,\texttt{inf}\}$), where k denotes the maximum length of token sequences considered by the algorithm. Only Enum-inf variant is a complete algorithm that doesn't impose any restriction on the token sequence lengths. (3) **FE:** The conditional learning engine in FlashExtract [24] that learns non-disjunctive expressions consisting of sequences of at most 3 tokens. We set 10 seconds as the timeout for each predicate synthesis task.

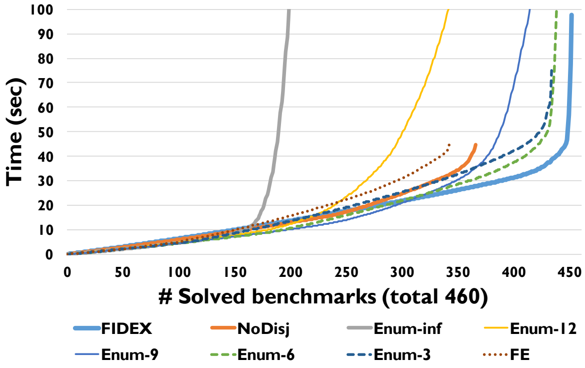


Figure 23. The running times for solving benchmarks for FIDEX and other baseline strategies.

	FIDEX	NoDisj	Enum-inf	Enum-12	Enum-9	Enum-6	Enum-3	FE
StartsWith	**424**(**45**)	317(23)	175(123)	330(497)	400(214)	383(41)	315(18)	N.A.
EndsWith	**417**(**36**)	269(19)	151(81)	283(554)	398(293)	386(39)	332(19)	N.A.
Matches	**438**(**44**)	243(16)	200(113)	311(190)	278(24)	186(9)	112(5)	N.A.
Contains	**374**(**128**)	323(75)	142(93)	174(257)	288(505)	373(214)	363(38)	N.A.
All predicates	**452**(**98**)	366(45)	212(314)	410(480)	442(343)	444(157)	434(75)	343(46)

Table 1. The number of solved benchmarks by FIDEX and other baseline strategies. The first four rows show the number of solved benchmarks using one predicate, and the last row shows the number of solved benchmarks using all predicates. The number of benchmarks solved by FE is shown only in the last row. XX(YY) means XX benchmarks are solved in YY seconds.

Time (s)	≤ 0.1	$(0.1, 0.2]$	$(0.2, 1]$	> 1
Percentage	81.2%	11.9%	6.0%	0.9%

# Examples	1	2	3-5	6-10	> 10
Percentage	13%	20%	42%	16%	9%

Table 2. Statistics of the running times and the number of examples (sum of both positive and negative examples) for FIDEX.

# Tokens	1	2-5	6-10	11-50	> 50
Percentage	19%	29%	23%	26%	3%

# Disjunctions	0	1	2	3-5	> 5
Percentage	75%	10%	6%	7%	2%

Table 3. Statistics of the complexity of the learnt disjunctive expressions (the number of tokens/disjunctions in the expression).

	GenDAG	$\otimes$	$\ominus$
StartsWith	66.3	1.1	5.0
EndsWith	66.0	1.2	4.8
Matches	66.0	1.7	2.5
Contains	66.3	20.7	178.4

Table 4. Average running time (ms) per benchmark for GenDAG algorithm, $\otimes$ and $\ominus$ operators.

(m,e,s,c)	65	(e,s,c,m)	74	(m,c,s,e)	90	(e,c,m,s)	94	(s,c,m,e)	100	(c,s,m,e)	312
(m,e,c,s)	69	(e,m,c,s)	77	(m,c,e,s)	90	(s,e,m,c)	98	(s,c,e,m)	113	(c,e,s,m)	312
(e,s,m,c)	73	(m,s,e,c)	77	(s,m,e,c)	92	(s,e,c,m)	98	(c,m,s,e)	286	(c,e,m,s)	312
(e,m,s,c)	73	(m,s,c,e)	84	(e,c,s,m)	94	(s,m,c,e)	99	(c,m,e,s)	286	(c,s,e,m)	324

Table 5. Total running time (seconds) for solving 460 benchmarks with different predicate orders in FIDEX, where s, e, m and c stand for StartsWith, EndsWith, Matches and Contains, respectively. Timeout is set as 10 seconds.

7.3 Number of Solved Benchmarks

Figure 23 shows the number of benchmarks solved by each strategy, which is also summarized in Table 1. FIDEX solves more benchmarks than any other baseline strategy (in less time). For StartsWith and EndsWith, FIDEX solves more benchmarks: 424 vs 400 (Enum-9) and 417 vs 398 (Enum-9), respectively. For Matches, FIDEX solves significantly more benchmarks: 438 as compared to 311 solved by Enum-12. Only for the Contains predicate, FIDEX and Enum-6 solve comparable number of benchmarks (374 vs 373), but FIDEX takes less time. Overall, FIDEX solves the maximum number of tasks 452/460[6], as opposed to fewer benchmarks solved by its variants (NoDisj and Enum-k) and the baseline FE. This confirms the need for not limiting the token sequence length and having disjunctions in the DSL, in order to improve the language expressiveness. The percentiles of the running times for FIDEX are shown in Table 2.

7.4 Effect of Predicate Order

The effect of using different predicate orders on the performance of FIDEX is shown in Table 5. The best running time is for the order {Matches, EndsWith, StartsWith, Contains} (65s), and the worst running time is for the order {Contains, StartsWith, EndsWith, Matches} (324s). The running time for the order used in FIDEX, {StartsWith, EndsWith, Matches, Contains}, is 98s, as highlighted in Table 5. We chose this predicate order based on the observations that (1) we believe users will find the learnt expression simpler and (2) the predicates have different computational complexities. Therefore, we have StartsWith at the beginning in order to learn simpler expressions, but since it solves fewer benchmarks than Matches, this order suffers from a little timeout penalty (0.07 seconds per benchmark on average). We have Contains predicate as the last predicate because it has the highest operator complexity.

7.5 Number of Examples

The number of positive and negative examples required by FIDEX to solve each benchmark task is shown in Figure 24. It requires 2.2 positive examples and 2.7 negative examples

[6] The 8 failed benchmarks timed out. After increasing the timeout to 100 seconds, only 1 benchmark failed because of timeout.

per benchmark on average, whereas the median values are 1 (positive) and 2 (negative). The ranking algorithm in FIDEX makes it learn the desired filter expressions from very few examples (and in fewer user iterations). The percentiles of the number of examples for FIDEX are shown in Table 2.

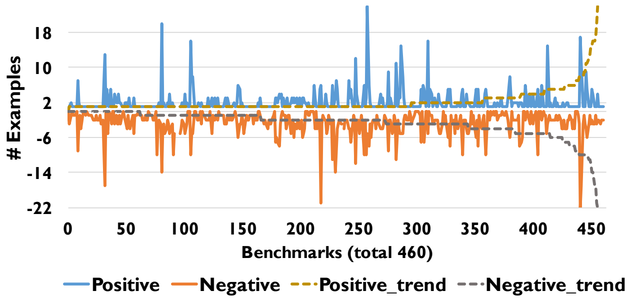


Figure 24. Number of positive and negative examples required by FIDEX for solving each benchmark.

Different ranking algorithms could also affect the number of positive/negative examples. Besides the one used in FIDEX, we evaluated two other ranking algorithms (one favors more general expressions and one favors more specific expressions). The ranking algorithm that favors more general expressions required more negative examples (because of underfitting), while the ranking algorithm that favors more specific expressions required more positive examples (because of overfitting).

7.6 Effect of Token Set Size

We evaluated FIDEX with two different token sets in the DSL (Large: 114 tokens v.s. Small: 46 tokens) and the learning time is shown in Figure 25. The small token set restricts the language expressiveness of FIDEX and causes it to solve fewer benchmarks. There is no big difference in the learning time for range of benchmarks that both versions can solve.

We do not observe much increase in the learning time of FIDEX with a larger token set because often the bit-vector representation of tokens on the edges are sparse (very few tokens amongst the whole token set match a given substring) and the ambiguous edge tokens are typically removed after a few intersection/subtraction operations.

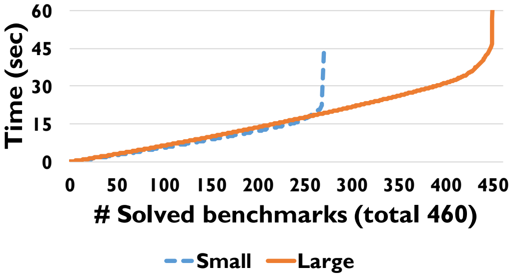


Figure 25. FIDEX with different token sets (114 v.s. 46).

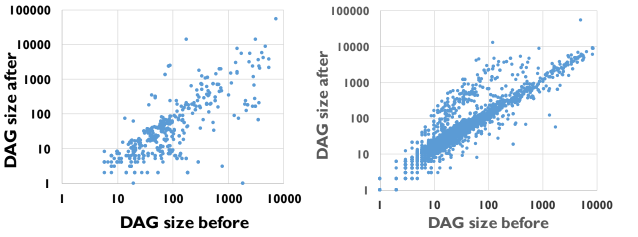


Figure 26. The DAG size before and after applying the $\otimes$ (left) and $\ominus$ (right) operators.

7.7 Complexity of $\otimes$ and $\ominus$ Operators

We present the size (# edges) of the learnt DAG before and after applying the $\otimes$ and $\ominus$ operators in Figure 26. As can be observed, the size of DAGs grows almost linearly in practice for both operators. For the $\ominus$ operator, there are a few cases (for `Contains` predicate) where the size does not grow linearly but it still grows in a polynomial fashion as opposed to the theoretical worst case bound of $\mathcal{O}(n^n)$. We hypothesize the reason for this phenomenon is the sparsity of edges in the DAGs obtained from real-world tasks.

7.8 Complexity of Learnt Expressions

The statistics about the complexity of predicate expressions in the `Filter` expressions learnt by FIDEX is shown in Table 3. About 75% of benchmarks can be solved without any disjunction and over 90% of benchmarks can be solved using at most 2 disjunctions. Almost half (48%) of token sequences in have lengths fewer than 5 and another half (49%) have lengths between 6 and 50. The most complex expression learnt by FIDEX has 222 tokens with 6 disjunctions.

FIDEX avoids learning too many disjunctions in the expressions because it employs a greedy `MergeDAGs` algorithm (shown in Figure 18) to introduce disjunctions lazily. The ranking algorithm (shown in Figure 24) in FIDEX helps find simpler expressions with fewer tokens.

7.9 Outlier Analysis

We studied the benchmarks for which (i) FIDEX learnt rather complex filter expressions (with more than 50 tokens or 5 disjunctions), (ii) FIDEX timed out (in 10 seconds), and (iii) FIDEX requires lots of examples (more than 20).

A common feature shared by these benchmarks is that they contain a large number (more than 1000) of long strings (with more than 500 characters), which requires FIDEX to learn longer (complex) token sequences and to spend longer time in computing intermediate results (line 6 in Figure 22). Another feature is that the positive and negative strings are logically similar (e.g., similar prefix and suffix regular expressions), which leads to more iterations (examples) to find the desired filter expressions. Very few benchmarks contain many incompatible positive strings which require different token sequences to recognize, and this causes FIDEX to learn many disjunctions. This phenomenon is also partly caused by the sub-optimality of the greedy `MergeDAGs` algorithm.

7.10 Threats to Validity

Although our benchmark set comes from a wide variety of sources such as online help forums and Microsoft Excel and Bing teams, it might not be representative of all kinds of filter transformations needed by the spreadsheet users. In our benchmarks, we assume that the correct set of positive and negative example strings are provided (or a user can provide such example strings based on manual inspection), but there might be some real-world cases of filtering that contain noisy labeling of examples.

8. Related Work

VSA-based data wrangling Version-space algebra (VSA) is a technique to succinctly represent a large set of consistent hypotheses and has been used in many domains including machine learning [27], Programming By Demonstrations (PBD) [23], and Programming By Examples (PBE) [32]. There are three recent PBE-based data wrangling systems that use VSA-based program representation: (i) FlashFill [16], (ii) FlashExtract [24], and (iii) FlashRelate [6]. FlashFill learns syntactic string transformations in spreadsheets using input-output examples. FlashExtract learns scripts to perform data extraction from semi-structured data using examples of the data items to be extracted in the file. FlashRelate learns programs to extract structured relational data from a semi-structured spreadsheet using examples of output relational tuples.

FIDEX is also a PBE system that uses VSA-based techniques to learn and succinctly represents a large number of filtering expressions given a set of examples, but differs from the previous PBE systems in four key ways. First, the application of data filtering is different from data extraction and transformation. Second, the DSL for FIDEX consists of filtering predicate expressions with disjunctions over token sequences of arbitrary length. The DSLs for FlashFill and FlashExtract have a small component of conditional expression learning, but they are limited to predicate expressions over token sequences of finite length (3 tokens). More importantly, there is no VSA-based sharing at the level of conditional expressions in these systems. Third, there is no algorithmic way to handle negative examples in these systems and FlashExtract uses enumerative brute-force techniques to discard programs that extract undesired data items. FIDEX, on the other hand, uses a novel DAG subtraction operator to algorithmically handle negative examples. Finally, unlike previous systems, the FIDEX synthesis algorithm uses intersection and subtraction operators for learning consistent DAGs, and can learn disjunctions in an incremental fashion.

Program synthesis The area of program synthesis is gaining a renewed interest [2]. In addition to data wrangling [6, 16, 24], the synthesis techniques have recently been developed for a wide range of problems including synthesizing efficient low-level code from partial programs [40], inference of efficient synchronization in concurrent programs [41], efficient compilation of declarative specifications [20], automated feedback generation for education [39], and compiler for low-power spatial architectures [30]. These synthesis techniques take as input different forms of specifications such as examples, partial programs, reference implementation, trace information, and etc. Among these specifications, example-based specifications, as used in FIDEX, have been recently used for interactive parser synthesis [25], synthesis of program transformations [26], synthesis of higher-order functional programs over recursive data types [12, 29], and synthesis of sequences of program refactoring [34]. In addition to a different application domain of data filtering, a major difference between our approach in FIDEX than these synthesis approaches is the ability to algorithmically handle negative examples using a novel DAG subtraction operator in a VSA-based representation of consistent hypotheses.

Handling negative examples FIDEX is the first PBE system that handles negative examples in an algorithmic way, which is achieved by the novel subtraction algorithm. Previous work either do not handle negative examples [16], or handle negative examples by running the synthesized programs [24]. In particular, FlashExtract [24] uses all positive examples to learn the set of consistent programs $\mathcal{P}$, and handles negative examples by running each program p in $\mathcal{P}$ on the input data and removing p from $\mathcal{P}$ if the extracted output contains a negative example. In contrast, FIDEX does not run any consistent program in $\mathcal{P}$, instead, it constructs the programs that conform to the negative example and removes them from the consistent programs $\mathcal{P}$ algorithmically by the novel subtraction operator.

Learning regular expressions by crowd-sourcing Recent work [8] investigated an approach to program synthesis called Program Boosting that is based on crowd-sourcing and genetic programming. It involves crowd-sourcing imperfect solutions to the problem of writing regular expressions for URLs or email addresses and then blending these programs together using geneting programming in a way that improves their correctness. FIDEX also learns regular-expression-like programs from an expressive DSL, but the learning algorithm is fully automated and does not involve crowd-sourcing techniques. The goal of FIDEX is to learn deterministic filter expression for well-defined tasks, whereas the Program Boosting approach tries to learn approximate regular expressions for fuzzy complex tasks relying on crowd wisdom.

Learning regular languages from positive/negative examples The problem of inducing regular languages from examples has been well studied in the past [1, 3–5, 10, 11, 13, 14, 22, 28, 31, 36]. It has been shown that finding the smallest automaton consistent with given input/output pairs is NP-complete [3, 14], and even finding an approximate solution for the given examples is intractable [31]. If one can only actively interact with the unknown automaton, it requires exponential time to identify the unknown automaton [4], and Firoiu et al. [13] developed algorithms for learning approximations of regular languages from positive evidence. Angluin proposed the L* algorithm [5] that can learn an automaton in polynomial time given an oracle that can perform membership and equivalence queries. In our setting, we do not have access to such an oracle and asking users to answer such queries is practically infeasible. The FIDEX DSL defines a structured subset of regular expressions that is not only able to express the different kinds of filtering tasks users need to perform in the real-world, but also enables efficient learning of expressions using few input-output examples.

Kushmerick [21] introduced a new technique for automatically learning wrappers (procedures to extract data from information sources) from example query responses. It identified a class of pre-defined wrappers whereas FIDEX defines a DSL corresponding to richer class of expressions. Moreover, the wrapper induction algorithm in [21] performs an enumerative search over the space of wrappers to learn a consistent wrapper to extract elements from a webpage, whereas the synthesis algorithm in FIDEX uses a version-space algebra based technique to learn the set of all consistent filter expressions given a set of examples.

9. Limitations and Future Work

A current limitation of the FIDEX system is that it cannot handle noisy examples. Since the algorithm is sound, it will try to learn an expression in the DSL that conforms exactly with the positive and negative examples. Another limitation is that the synthesis algorithm only performs a linear search order over predicates to learn filter expressions. We are currently looking at adding probabilistic reasoning to the synthesis algorithm to handle small quantities of noise in the examples and datasets [35, 38]. The FIDEX system learns a filter expression over a list of positive and negative example strings. We are also working on extending the support for filter expressions over richer data structures than strings that have more structural contexts.

10. Conclusion

In this paper, we have identified and addressed an important problem of data filtering that is present in many data wrangling tasks. We presented a system FIDEX that can efficiently learn a filtering expression given a set of positive and negative string instances in an incremental fashion. The key idea of our approach is to design an expressive DSL for string filtering tasks and then develop efficient synthesis algorithms to learn the desired DSL expressions using a DAG data structure. We presented the effectiveness as well as the efficiency of the FIDEX system on 460 real-world benchmarks involving string filtering tasks.

References

[1] R. Alquezar and A. Sanfeliu. Incremental grammatical inference from positive and negative data using unbiased finite state automata. In *In Proceedings of the ACL02 Workshop on Unsupervised Lexical Acquisition*, pages 291–300, 1994.

[2] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[3] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[4] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51:76–87, 1981.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[6] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. PLDI, pages 218–228, 2015.

[7] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 99(PrePrints):1, 2013.

[8] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes. Program boosting: Program synthesis via crowd-sourcing. POPL, pages 677–688, 2015.

[9] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., 1 edition, 2003.

[10] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004.

[11] P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, pages 222–237. Springer, 1996.

[12] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239, 2015.

[13] L. Firoiu, T. Oates, and P. R. Cohen. Learning regular languages from positive evidence. In *In Twentieth Annual Meeting of the Cognitive Science Society*, pages 350–355, 1998.

[14] M. E. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[15] M. Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, April 2009.

[16] S. Gulwani. Automating string processing in spreadsheets using input-output examples. POPL, pages 317–330, 2011.

[17] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.

[18] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR 2015*, 2015.

[19] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[20] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[21] N. Kushmerick. *Wrapper Induction for Information Extraction*. PhD thesis, 1997.

[22] K. J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 45–52. ACM, 1992.

[23] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, Oct. 2003.

[24] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. PLDI, pages 542–553, 2014.

[25] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *PLDI*, pages 565–574, 2015.

[26] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[27] T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2): 203–226, 1982.

[28] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. P. de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[29] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[30] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodík. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *PLDI*, page 42, 2014.

[31] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40:95–142, 1993.

[32] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. OOPSLA, pages 107–126, 2015.

[33] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[34] V. Raychev, M. Schäfer, M. Sridharan, and M. T. Vechev. Refactoring with synthesis. In *OOPSLA*, pages 339–354, 2013.

[35] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause. Learning programs from noisy data. In *POPL*, pages 761–774, 2016.

[36] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. STOC '89, pages 411–420. ACM, 1989.

[37] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV*, pages 398–414, 2015.

[38] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. POPL, pages 343–356. ACM, 2016.

[39] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[40] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[41] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, New York, NY, USA, 2010. ACM.

[42] M. D. Wulf, L. Doyen, T. A. Henzinger, and J. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, pages 17–30, 2006.

Appendix

We now present the proofs of the theorems in the paper.

Theorem 1. *Figure 10 implements the semantics of* $\bigotimes$*:* $\mathcal{D} = \mathcal{D}_1 \bigotimes \mathcal{D}_2 \Rightarrow \llbracket \mathcal{D} \rrbracket = \llbracket \mathcal{D}_1 \rrbracket \cap \llbracket \mathcal{D}_2 \rrbracket$.

Proof. We prove that for $\mathcal{D} = \mathcal{D}_1 \bigotimes \mathcal{D}_2$:

- $\forall\ ts \in \llbracket \mathcal{D}_1 \rrbracket \wedge ts \in \llbracket \mathcal{D}_2 \rrbracket$, we have $ts \in \llbracket \mathcal{D} \rrbracket$. This shows that we have $\llbracket \mathcal{D} \rrbracket \supseteq \llbracket \mathcal{D}_1 \rrbracket \cap \llbracket \mathcal{D}_2 \rrbracket$.
- $\forall\ ts \in \llbracket \mathcal{D} \rrbracket$, we have $ts \in \llbracket \mathcal{D}_1 \rrbracket \wedge ts \in \llbracket \mathcal{D}_2 \rrbracket$. This shows that we have $\llbracket \mathcal{D} \rrbracket \subseteq \llbracket \mathcal{D}_1 \rrbracket \cap \llbracket \mathcal{D}_2 \rrbracket$. Therefore, we can prove this theorem.

Suppose we have $\mathcal{D}_1 = \mathcal{D}(Q_1, S_1, F_1, E_1, W_1)$, $\mathcal{D}_2 = \mathcal{D}(Q_2, S_2, F_2, E_2, W_2)$, $\mathcal{D} = \mathcal{D}(Q, S, F, E, W)$ and $ts = \mathtt{Seq}(t_1, \cdot\cdot, t_n)$, and we prove each statement as following.

- Since $ts \in \llbracket \mathcal{D}_1 \rrbracket$, there exists a path $p_1 = (q_1^0, \cdot\cdot, q_1^n)$ in $\mathcal{D}_1$ such that $q_1^0 \in S_1, q_1^n \in F_1$, and $t_i \in W_1((q_1^{i-1}, q_1^i))$ where $i = 1, \cdot\cdot, n$. Similarly, since $ts \in \llbracket \mathcal{D}_2 \rrbracket$, there exists a path $p_2 = (q_2^0, \cdot\cdot, q_2^n)$ in $\mathcal{D}_2$ such that $q_2^0 \in S_2, q_2^n \in F_2$ and $t_i \in W_2((q_2^{i-1}, q_2^i))$ where $i = 1, \cdot\cdot, n$. Now consider the following nodes in $\mathcal{D}$: $q^0 = (q_1^0, q_2^0), \cdot\cdot, q^n = (q_1^n, q_2^n)$: we have $q^0 \in S$, $q^n \in F$, $(q^{i-1}, q^i) \in E$, $W((q^{i-1}, q^i)) = W_1((q_1^{i-1}, q_1^i)) \cap W_2((q_2^{i-1}, q_2^i)) \supseteq \{t_i\} \neq \emptyset$ where $i = 1, \cdot\cdot, n$. Thus, $p = (q^0, \cdot\cdot, q^n)$ is a valid path in $\mathcal{D}$, and we have ts to be one of the token sequences represented by p. Therefore, we have $ts \in \llbracket \mathcal{D} \rrbracket$.
- Since $ts \in \llbracket \mathcal{D} \rrbracket$, there exists a path $p = (q^0, \cdot\cdot, q^n)$ in $\mathcal{D}$ such that $q^0 \in S, q^n \in F$ and $t_i \in W((q^{i-1}, q^i))$, where $i = 1, \cdot\cdot, n$. Since $\mathcal{D} = \mathcal{D}_1 \bigotimes \mathcal{D}_2$, we have $q^i = (q_1^i, q_2^i)$, where $q_1^i \in Q_1$ and $q_2^i \in Q_2$. Also, since $(q^{i-1}, q^i) \in E$, we have $(q_1^{i-1}, q_1^{i-1}) \in E_1$ and $(q_2^{i-1}, q_2^i) \in E_2$. Furthermore, since $t_i \in W((q^{i-1}, q^i)) = W_1((q_1^{i-1}, q_1^i)) \cap W_2((q_2^{i-1}, q_2^i))$, we have $t_i \in W_1((q_1^{i-1}, q_1^i))$ and $t_i \in W_2((q_2^{i-1}, q_2^i))$, where $i = 1, \cdot\cdot, n$. Therefore, we have two valid paths $p_1 = (q_1^0, \cdot\cdot, q_1^n)$ and $p_2 = (q_2^0, \cdot\cdot, q_2^n)$ in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, and ts is one of the token sequences represented by both p_1 and p_2. Therefore, we have $ts \in \llbracket \mathcal{D}_1 \rrbracket \wedge ts \in \llbracket \mathcal{D}_2 \rrbracket$. □

Theorem 2. *Figure 11 implements the semantics of* $\ominus$*:* $\mathcal{D} = \mathcal{D}_1 \ominus \mathcal{D}_2 \Rightarrow \llbracket \mathcal{D} \rrbracket = \llbracket \mathcal{D}_1 \rrbracket \setminus \llbracket \mathcal{D}_2 \rrbracket$.

Proof. We prove that for $\mathcal{D} = \mathcal{D}_1 \ominus \mathcal{D}_2$:

- $\forall\ ts \notin \llbracket \mathcal{D}_1 \rrbracket$, we have $ts \notin \llbracket \mathcal{D} \rrbracket$. This shows that we have $\llbracket \mathcal{D} \rrbracket \subseteq \llbracket \mathcal{D}_1 \rrbracket$.
- $\forall\ ts \in \llbracket \mathcal{D}_1 \rrbracket \wedge ts \in \llbracket \mathcal{D}_2 \rrbracket$, we have $ts \notin \llbracket \mathcal{D} \rrbracket$. This further shows that we have $\llbracket \mathcal{D} \rrbracket \subseteq \llbracket \mathcal{D}_1 \rrbracket \setminus \llbracket \mathcal{D}_2 \rrbracket$.
- $\forall\ ts \in \llbracket \mathcal{D}_1 \rrbracket \wedge ts \notin \llbracket \mathcal{D}_2 \rrbracket$, we have $ts \in \llbracket \mathcal{D} \rrbracket$. This further shows that we have $\llbracket \mathcal{D} \rrbracket = \llbracket \mathcal{D}_1 \rrbracket \setminus \llbracket \mathcal{D}_2 \rrbracket$.

Suppose we have $\mathcal{D}_1 = \mathcal{D}(Q_1, S_1, F_1, E_1, W_1)$, $\mathcal{D}_2 = \mathcal{D}(Q_2, S_2, F_2, E_2, W_2)$ and $ts = \mathtt{Seq}(t_1, \cdot\cdot, t_n)$. We prove each statement as following.

- Since the algorithm does not add any token sequence in $\mathcal{D}_1$, thus if $ts \notin \llbracket \mathcal{D}_1 \rrbracket$, we have $ts \notin \llbracket \mathcal{D} \rrbracket$.
- Since $ts \in \llbracket \mathcal{D}_2 \rrbracket$, there exists a path $p_2 = (q_2^0, \ldots, q_2^n)$ in $\mathcal{D}_2$ where $q_2^0 \in S_2, q_2^n \in F_2$, and $t_i \in W_1((q_2^{i-1}, q_2^i))$, for $i = 1, \cdot\cdot, n$. Similarly, since $ts \in \llbracket \mathcal{D}_1 \rrbracket$, there exists at least one path $p_1 = (q_1^0, \cdot\cdot, q_1^n)$ in $\mathcal{D}_1$ where $q_1^0 \in S_1, q_1^n \in F_1$, and $t_i \in W_1((q_1^{i-1}, q_1^i))$ for $i = 1, \cdot\cdot, n$. Assume p_1 is an arbitrary path among these paths that contain ts as a token sequence. We show that we have $ts \notin \llbracket \mathcal{D} \rrbracket$ by showing that ts is removed from (the arbitrarily selected) p_1.

Since the algorithm exhausts all pairs of starting nodes in both $\mathcal{D}_1$ and $\mathcal{D}_2$, there exists an iteration in which it calls $\mathtt{SubPartialDAG}(\mathcal{D}_1, \ddot{q}_1^0, \mathcal{D}_2, q_2^0)$ where $\ddot{q}_1$ is a copy of q_1^0. We show that the token sequence ts represented in the path p_1 will only be contained along the path $(\ddot{q}_1^0, \cdot\cdot, \ddot{q}_1^n)$ where each $\ddot{q}_1^i$ is a copy of $q_1^i (i = 0, \cdot\cdot, n)$ after $\mathtt{SubPartialDAG}$ terminates (call it P). Then, since the algorithm makes $\ddot{q}_1^n$ a non-ending node, ts is removed from $(\ddot{q}_1^0, \cdot\cdot, \ddot{q}_1^n)$, and thus also removed from the original path p_1.

We now prove P by showing inductively that $\mathtt{Seq}(t_1, \cdot\cdot, t_k)$ is only contained on the path $(\ddot{q}_1^0, \cdot\cdot, \ddot{q}_1^k)$ where $k = 1, \cdot\cdot, n$ (call it Q). For the base case with $k = 1$, Q holds trivially because it performs the path isolation. For the inductive case, the inductive hypothesis assumes that Q holds for $k = m$ $(m \geq 1)$, and we show that Q also holds for $k = m + 1$. Before performing the path isolation, we have $(\ddot{q}_1^m, q_1^{m+1}) \in E_1$ with $t_m \in W((\ddot{q}_1^m, q_1^{m+1}))$ as well as $(q_1^m, q_1^{m+1}) \in E_1$ with $t_m \in W((q_1^m, q_1^{m+1}))$. It then performs the path isolation by copying q_1^{m+1} to $\ddot{q}_1^{m+1}$ and updating the labels on $(\ddot{q}_1^m, q_1^{m+1})$ and $(\ddot{q}_1^m, \ddot{q}_1^{m+1})$ such that we have $t_{m+1} \in W((\ddot{q}_1^m, \ddot{q}_1^{m+1}))$ and $t_{m+1} \notin W((\ddot{q}_1^m, q_1^{m+1}))$. Combined with the inductive hypothesis that $\mathtt{Seq}(t_1, \cdot\cdot, t_m)$ is only contained on path $(\ddot{q}_1^0, \cdot\cdot, \ddot{q}_1^m)$, we have $\mathtt{Seq}(t_1, \cdot\cdot, t_{m+1})$ is only contained on path $(\ddot{q}_1^0, \cdot\cdot, \ddot{q}_1^{m+1})$.

- We prove this statement by showing that whenever it removes a path $p = (q_2^0, \cdot\cdot, q_2^n) \in \mathcal{D}_2$ from paths in $\mathcal{D}_1$, the algorithm does not remove token sequences other than those on path p. This is valid since the algorithm always performs the path isolation to isolate the path with only the tokens on $(q_2^{i-1}, q_2^i), i = 1, \cdot\cdot, n$. Then making an end node a non-ending node removes only the token sequences on the path p in $\mathcal{D}_2$. □

Lemma 1. *The* $\mathtt{GenDAG}$ *algorithm is sound and complete, i.e., for a predicate* $\mathtt{Pred}$ *and a string* $s = \sigma(v)$*, it learns a DAG* $\mathcal{D}$ *such that i)* $\forall$ *token sequence* $ts \in \llbracket \mathcal{D} \rrbracket$ *we have* $\llbracket \mathtt{Pred}(v, ts) \rrbracket\ \sigma = \mathtt{True}$*, and ii) if there exists a token sequence* ts *in the DSL for a predicate* $\mathtt{Pred}$ *such that* $\llbracket \mathtt{Pred}(v, ts) \rrbracket\ \sigma = \mathtt{True}$*, then we have* $ts \in \llbracket \mathcal{D} \rrbracket$.

Proof. Since the $\mathtt{GenDAG}$ algorithm constructs all tokens that can match each substring of s, and the DAG data structure represents all possible concatenations (sequences) of those tokens, this lemma follows. □

Theorem 3. *The* $\mathtt{LearnTokenSeqs}_{\mathtt{Pred}}$ *algorithm is sound and complete for the non-disjunctive predicate expression fragment in the DSL.*

Proof. This theorem follows according to Lemma 1, Theorem 1 and Theorem 2. □

Lemma 2. *If the input DAG list $\widetilde{\mathcal{D}}$ of the* `MergeDAGs` *algorithm in Figure 18 is not empty, then the output DAG list $\widetilde{\mathcal{D}}_{res}$ is also not empty.*

Proof. This lemma follows since the `MergeDAGs` algorithm applies $\bigotimes$ only if the resulting DAG is not empty. □

Theorem 4. *The $\mathtt{LearnDisjExprs}_{\mathtt{Pred}}$ algorithm is sound and complete for all predicate expressions in the DSL.*

Proof. This theorem follows according to Lemma 1, Theorem 2, and Lemma 2. □

Theorem 5. *The $\mathtt{LearnDisjExprsInc}_{\mathtt{Pred}}$ algorithm is sound and complete for all predicate expressions in the DSL.*

Proof. This theorem follows according to Lemma 1 and Theorem 2. □

Theorem 6. *The* `LearnFilter` *algorithm is sound and complete for the* `Filter` *expression in the DSL.*

Proof. This theorem follows according to Theorem 5 and the fact that the `LearnFilter` algorithm always maintains the full DAG list for all previous examples. □