BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations

Rishabh Singh
Microsoft Research, Redmond, USA
risin@microsoft.com

ABSTRACT

The recent Programming By Example (PBE) techniques such as FLASHFILL have shown great promise for enabling end-users to perform data transformation tasks using input-output examples. Since examples are inherently an under-specification, there are typically a large number of hypotheses conforming to the examples, and the PBE techniques suffer from scalability issues for finding the intended program amongst the large space.

We present a semi-supervised learning technique to significantly reduce this ambiguity by using the logical information present in the input data to guide the synthesis algorithm. We develop a data structure `InputDataGraph` to succinctly represent a large set of logical patterns that are shared across the input data, and use this graph to efficiently learn substring expressions in a new PBE system BLINKFILL. We evaluate BLINKFILL on 207 real-world benchmarks and show that BLINKFILL is significantly faster (on average 41x) and requires fewer input-output examples (1.27 vs 1.53) to learn the desired transformations in comparison to FLASHFILL.

1. INTRODUCTION

The IT revolution has resulted in massive digitization of data and in making this data accessible to millions of users. Despite significant advances in technologies for helping users perform data analysis, the process of transforming and cleaning data before any useful analysis is still challenging and time consuming. Some studies have reported that this step of data cleaning and reshaping (also called *wrangling*) can take up to 80% of the data analysts' time [4]. These data analysts have myriad diverse backgrounds and lack programming knowledge to automate data wrangling steps [6]. Simpler specification mechanisms such as examples [1, 7] and predictive interaction [10] are recently becoming more popular to cater to the needs of these users.

An important subset of the data wrangling problem consists of regular expression based string transformation tasks. FLASHFILL [7, 8], a recently introduced feature in Microsoft Excel 2013, is a Programming By Example (PBE) system for helping end-users perform such string transformations using examples. The key idea in FLASHFILL is to learn programs in a Domain-specific language (DSL) that is expressive enough to encode the majority of real-world tasks, but also restricted enough for efficient learning. Since examples are often an under-specification of the intended task, there are typically a large number of programs in the DSL that conform to the examples. FLASHFILL uses version-space algebra (VSA) [12, 13] to succinctly represent this large set of consistent programs and then uses ranking to identify the most likely program [27]. Since FLASHFILL only uses the input-output examples and ignores the other spreadsheet data items, it needs to learn and maintain a large set of programs that are consistent with the examples but may not perform the desired transformation on other data items. This, in turn, results in scalability issues of the synthesis algorithm on longer strings and also enforces larger restrictions on the transformations supported in the DSL (e.g. FLASHFILL only supports a finite set of pre-defined regular expression tokens).

In this paper, we present BLINKFILL, a PBE system that efficiently learns string transformations in spreadsheets from input-output examples by leveraging the logical information present in the spreadsheet data. We show that this semi-supervised technique can significantly reduce the inherent ambiguity in PBE, which in turn results in an efficient learning algorithm. Moreover, learning from other spreadsheet data also enables a new class of transformations that use regular expressions based on arbitrary constant strings.

There are two key challenges in learning from spreadsheet data: 1) how to efficiently compute the set of all logical structures (and sub-structures) that are consistent with a set of column values, and 2) how to use the logical structures in learning string transformation programs. BLINKFILL constructs a data structure `InputDataGraph` that succinctly encodes the set of all logical structures that are shared across a set of column values. It then uses the sub-paths in the graph data structure to learn and disambiguate between a large number of DSL expressions that are consistent with a given set of input-output examples. The DSL for string transformation tasks in BLINKFILL is similar to that of FLASHFILL at the top-level, but consists of new substring extraction expressions based on the `InputDataGraph` nodes. We present a sound and complete synthesis algorithm to learn the set of all expressions in the DSL that conform to a set of input-output examples, and a ranking algorithm to select the most likely program amongst them. Unlike FLASHFILL, the DSL

for BLINKFILL does not support conditionals and loops, but supports a richer class of substring extraction tasks since it can learn arbitrary regular expression tokens based on the spreadsheet data. The substring extraction and merging tasks supported by BLINKFILL account for the majority (more than 88%) of the real-word FLASHFILL tasks obtained from the Excel product team.

The idea of using the data items other than input-output examples to learn consistent programs is inspired from the work on Potter's wheel [22]. Potter's wheel is a precursor to Trifacta's pioneering Data Wrangler system [11] for interactive data wrangling. For column split transformations, Potter's wheel learns the most suitable logical structure consistent with the set of column strings based on the Minimum Description Length metric [23]. Our technique of semi-supervised learning in BLINKFILL is different from Potter's wheel in three key ways. First, BLINKFILL learns the complete set of logical structures (and sub-structures) that are consistent with a given set of column values instead of learning only one structure as there might be multiple consistent sub-structures. Second, the learnt structures are represented succinctly using a VSA so that they can be integrated in VSA-based program synthesis techniques such as FLASHFILL. Finally, the class of string transformations supported by BLINKFILL (concatenation of substrings and constant strings) is richer than the string transformations supported by Potter's wheel.

We evaluate BLINKFILL on 207 benchmarks [27] obtained from the Excel team and online help forums. In comparison to FLASHFILL, BLINKFILL takes significantly less time to learn the desired transformations and requires fewer input-output examples (in spite of FLASHFILL using a highly-tuned machine learning based ranking technique). BLINKFILL is on average $41\times$ (median $11\times$) faster than FLASHFILL. It requires on average 1.27 examples per benchmark as opposed to 1.53 examples needed by FLASHFILL. This paper makes the following key contributions:

- We present a new data structure `InputDataGraph` to succinctly represent the set of all logical structures in a DSL that are consistent with a given set of strings, and present an efficient algorithm to learn the structure.

- We present a new PBE system BLINKFILL, whose DSL consists of concatenated expressions of constant string expressions and substring expressions (based on the `InputDataGraph`). We present a sound and complete synthesis algorithm to learn expressions in the DSL.

- We evaluate BLINKFILL on 207 real-world benchmarks obtained from the Excel product team and online help forums. BLINKFILL is significantly faster ($41\times$) and requires fewer input-output examples (1.27 vs 1.53) in comparison to FLASHFILL.

2. PRELIMINARIES

Without loss of generality, we assume that the string transformation task involves transforming a set of n input row strings $\{(v_1^1, \cdots, v_k^1), \cdots, (v_1^n, \cdots, v_k^n)\}$, and the user has provided a set of m input-output examples $\{((v_1^1, \cdots, v_k^1), o_s^1), \cdots, ((v_1^m, \cdots, v_k^m), o_s^m)\}$. A string s is considered as simply a sequence of characters, where $s[1]$ denotes the first character of the string and $\text{len}(s)$ denotes the length of the string. We use the notation $s[i..j]$ to denote the substring of s that

Token	Regex	Abbr.
ProperCase	$r_p \equiv \backslash\text{p}\{\text{Lu}\}(\backslash\text{p}\{\text{Ll}\})+$	p
CAPS	$r_C \equiv \backslash\text{p}\{\text{Lu}\}+$	C
lowercase	$r_l \equiv \backslash\text{p}\{\text{Ll}\}+$	l
Digits	$\backslash\text{d}+$	d
Alphabets	$r_\alpha \equiv [\backslash\text{p}\{\text{Lu}\}\backslash\text{p}\{\text{Ll}\}]+$	α
Alphanumeric	$[\backslash\text{p}\{\text{Lu}\}\backslash\text{p}\{\text{Ll}\}0\text{-}9]+$	αn
Whitespace	$r_{ws} \equiv \backslash\text{p}\{\text{Zs}\}+$	ws
StartT	$\wedge$	$\wedge$
EndT	$\$$	$\$$
ProperCaseWSpaces	$r_p(r_{ws}r_p)^*$	ps
CAPSWSpaces	$r_C(r_{ws}r_C)^*$	Cs
lowercaseWSpaces	$r_l(r_{ws}r_l)^*$	ls
AlphabetsWSpaces	$r_\alpha(r_{ws}r_\alpha)^*$	αs

Table 1: The set of base regular expression based tokens supported by BlinkFill.

starts at index i and ends at index j, where $j \geq i$ and length of the substring is $j - i + 1$.

Tokens: BLINKFILL supports two kinds of token patterns at the base level of the DSL: (i) regular expression tokens and (ii) constant string tokens. The regular expression tokens match a string with a predefined regular expression pattern. There are 13 such tokens supported by BLINKFILL, which are shown in Table 1. The constant string tokens match a string with the corresponding constant string pattern and are learnt automatically by BLINKFILL during the synthesis process.

DEFINITION 1 (TOKEN MATCH). *Let n be the number of matches of the pattern defined by a token τ in a given string s. We define a token match (τ, k) as the kth $((n+k)^{th}$ if $k < 0$) match of token τ in s. We denote the set of all token matches as $\tau \vDash s$, such that $\text{size}(\tau \vDash s) = n$, $(\tau, k) \equiv (\tau \vDash s)[k]$ if $k > 0$, and $(\tau, k) \equiv (\tau \vDash s)[n + k + 1]$ if $k < 0$. We denote the start and end indices of a token match as $(\tau \vDash s)[k] \hookrightarrow$ and $(\tau \vDash s)[k] \hookleftarrow$ respectively.*

For example, given a string $s = $ `"Mumbai, India"`, the token match (`C,1`) denotes the 1st match of the CAPS token (matching the substring `"M"`), where $\text{size}(\text{C} \vDash s) = 2$, $(\text{C} \vDash s)[1] \hookrightarrow= 1$, $(\text{C} \vDash s)[1] \hookleftarrow= 1$. The token match (`C,-1`) (matching the substring `"I"`) is defined by $(\text{C} \vDash s)[2] \hookrightarrow= 8$, $(\text{C} \vDash s)[2] \hookleftarrow= 8$. Similarly, the token match (`p,2`) matches the substring `"India"`, (`l,1`) matches `"umbai"`, the constant token match (`"M",1`) matches `"M"`, and the constant token match (`", "`, 1) matches the substring `", "`.

Version space algebra: Version-space Algebra (VSA) was first introduced by Mitchell [19] in the context of machine learning and was later used in Programming by Examples/Demonstration systems such as SmartEdit [12] and FLASHFILL [7]. The key idea in VSA is to succinctly represent an exponential number of programs in polynomial space and can be intuitively viewed as a directed graph with three types of nodes: 1) leaf nodes with direct set of programs, 2) intermediate union nodes representing a set-union of its children VSAs, and 3) intermediate join nodes with k children VSAs annotated with a k-ary function F such that it represents all resulting application of F to the cross-product of children values. More details about the VSA formalization and applications can be found in [21, 27].

3. MOTIVATING EXAMPLES

In this section, we present a few motivating scenarios that demonstrate the usefulness of semi-supervised learning for string transformation tasks. BLINKFILL efficiently learns the set of all logical structures consistent with the input strings and uses it to learn the desired transformation from only 1 or 2 input-output examples for each of these scenarios.

Example 1. An Excel user had a list of city and country names in a column, and needed to extract the country name. The user provided an input-output example (emphasized in bold) to express the intent as shown in Figure 1.

	Input v_1	Output
1	Mumbai, India	**India**
2	Los Angeles, United States of America	
3	Newark, United States	
4	New York, United States of America	
5	Wellington, New Zealand	
6	New Delhi, India	

Figure 1: Extracting country from input column.

Given the first input-output example `"Mumbai, India"` → `"India"`, there are a large number of possible logical expressions to extract the substring `"India"`. The substring expression in the DSL that extracts such substrings from the input strings is defined using two position expressions: one position expression for the left index of the substring and the other for the right index. For example, some possible logics to identify the left index of the substring `"India"` in the input string `"Mumbai, India"` are: i) start of 2nd `Alphabets` token, ii) start of 2nd `Alphanumeric`, iii) end of 1st `Whitespace`, iv) end of 1st comma followed by whitespace, etc. There are more than 10^3 different logics in our DSL that can identify this position. Similarly, there are more than 10^3 different logics for identifying the right index of the substring. Given just the input-output example, it is challenging to select the desired expression from more than 10^6 different choices of conforming logics.

BLINKFILL uses semi-supervised learning to identify that the most desirable logic for left index of substring `"India"` is the end of 1st `ConstantStr(", ")` (comma followed by a whitespace), whereas the most desirable logic for the right index is the end of -1st `lowercase` (where negative values denote match occurrences from the end). BLINKFILL prefers token sequences that have larger contexts around them in input data as they are more likely to correspond to the desired logic. For this example, it finds that the left index logic is preceded by the context `StartT ∘ 1ˢᵗ ProperCaseWSpaces`, and the right index logic is followed by the context `EndT`.

Example 2. An Excel user wanted to abbreviate a list of names to the corresponding initials of First and Last names as shown in Figure 2. The presence of optional middle names in different formats made it challenging for the user.

This example requires extraction of multiple substrings. Given an input-output example `"Brandon Henry Saunders"` → `"B.S."`, BLINKFILL needs to learn substring expressions to extract the substrings `"B"` and `"S"`. For the substring `"B"`, there are again many different possible logics for the

	Input v_1	Output
1	Brandon Henry Saunders	**B.S.**
2	William Lee	
3	Dafna Q. Chen	
4	Danelle D. Saunders	
5	Emilio William Concepcion	

Figure 2: Abbreviating names to initials.

left index such as start of 1st `CAPS`, start of 1st `ProperCase`, end of `StartT`, etc. Machine learning based techniques can reasonably identify that 1st `CAPS` is a more commonly-used expression globally and can rank them higher than other choices. But for the left index of the second substring `"S"`, identifying the correct logic is quite challenging as there are many equally likely hypotheses such as end of -1st `Whitespace`, end of 2nd `Whitespace`, start 3rd `CAPS`, end of -1st `ProperCase ∘ Whitespace`, etc. BLINKFILL uses the logical patterns shared by the input column strings to learn the following index expressions:

- For substring `"B"`, left: start of 1st `ProperCaseWSpaces` (end of `StartT` as context), right: end of 1st `CAPS` (end of `StartT` and start of 1st `lowercase` as contexts)

- For substring `"S"`, left: end of -1st `Whitespace` (start of -1st `ProperCase ∘ EndT` as context), right: start of -1st `lowercase` (start of `EndT` as context)

Example 3. A user posted the following spreadsheet on an Excel help-forum consisting of a list of medical billing codes, where some codes had a missing `"]"` at the end. The user wanted to clean the data by adding the missing `"]"` only for the strings where it was missing, and didn't want to duplicate `"]"` for the strings where it was already present.

	Input v_1	Output
1	[CPT-00350	**[CPT-00350]**
2	[CPT-00340	
3	[CPT-11536]	
4	[CPT-115]	

Figure 3: Adding "]" to codes with a missing "]".

This is an interesting example as it requires FLASHFILL to learn a conditional program to insert `"]"` to only strings that do not already have `"]"` at the end. After the first example, FLASHFILL learns the simplest program to insert `"]"` at the end of every input. The user needs to provide an additional example corresponding to the distinct behavior to help FLASHFILL learn the conditional program. On the other hand, BLINKFILL learns the desired transformation from the first example itself. Instead of selecting the substring between the `StartT` and `EndT` (i.e. the complete input), it learns that the most common logical pattern shared by the input strings is: `StartT ∘ 1ˢᵗ ConstantStr("[CPT-") ∘ 1ˢᵗ Digits`. It then learns the program that concatenates the substring between end of `StartT` and end of 1st `Digits` with the constant string `"]"`. For inputs without `"]"` at the end, the substring expression extracts the complete substring. For inputs ending with `"]"`, the substring expression extracts only the substring before the last `"]"`.

Example 4. A user posted the following spreadsheet on StackOverflow, where the user wanted to extract the information (of varying length) that was present between two strings `"nextData"` and `"moreInfo"` as shown in Figure 4.

	Input v_1	Output
1	nextData 12 Street moreInfo 35	**12 Street**
2	nextData Main moreInfo 36	**Main**
3	nextData Albany Street moreInfo 37	
4	nextData 134 Green Street moreInfo 39	

Figure 4: Extracting message between the constant strings `"nextData"` and `"moreInfo"`.

The transformation in this example cannot be learnt by FLASHFILL since it is limited by a finite predefined list of regular expression tokens and does not consider constant string based tokens such as `"nextData"`. BLINKFILL, however, is not limited by a finite list of tokens, and considers different possible logical structures in the input strings including all possible combinations of constant string tokens and regular expression tokens. It learns the substring expression to extract the desired substring whose left and right indices are defined by the logics end of 1^{st}ConstantStr(`" nextData "`) and start of 1^{st}ConstantStr(`" moreInfo "`), with StartT and -1^{st}Digits $\circ$ EndT as contexts respectively.

4. OVERVIEW OF THE APPROACH

We first define the abstract semi-supervised PBE problem. Let D denote the set of data items, $I = \{(i_1, o_1), \cdots, (i_n, o_n)\}$ denote the set of n input-output examples, and L denote the DSL that defines the possible space of programs. The traditional PBE techniques solve the following problem: $\exists P \in L \ \forall (i, o) \in I : P(i) = o$, i.e. find a program P in DSL L that is consistent with all input-output examples. The semi-supervised approach also takes into account the dataset D for learning the program and solves the following constraint: $\exists P \in L \ \forall (i, o) \in I : P(i) = o \wedge \text{SubExpr}(P) \subseteq G(D, L)$, where $G(D, L)$ denotes a data structure that represents a set of sub-expressions in L and $\text{SubExpr}(P)$ denotes the set of sub-expressions of P. For a sound and complete synthesis algorithm, we require that the data structure $G(D, L)$ consists of all sub-expressions in L that are consistent with each data item in the dataset D.

An overview of our instantiation of the semi-supervised PBE approach in BLINKFILL is shown in Figure 5. The synthesis algorithm takes two kinds of inputs: 1) the traditional input-output examples given by users in the form of a set of tuples of spreadsheet rows and the corresponding outputs, and 2) the `InputDataGraph`. The `GenInputGraph` module uses the spreadsheet data to construct the `InputDataGraph` that succinctly represents the set of all logical patterns that are shared across the spreadsheet data. The synthesis algorithm then uses the `InputDataGraph` to efficiently learn a program in the DSL that is consistent with the input-output examples, and executes it on the spreadsheet data to compute the outputs for the remaining spreadsheet rows.

5. INPUT DATA GRAPHS

Given a set of strings in a spreadsheet column, our goal is to learn the set of all logical patterns (in a DSL) that are
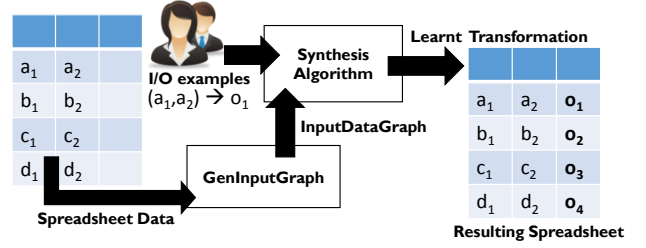

Figure 5: An overview of our approach in BlinkFill.

consistent with any substring of these strings. This is challenging because of two reasons: 1) there are a huge number of token sequences that are consistent with the quadratic number of substrings of a single string, and 2) there are exponentially many possible alignments of the strings in a column. A naïve approach would first enumerate exponentially many token sequences for each string in a column, and then compute all matching sub-sequences in double exponential time. We present a data structure `InputDataGraph` that succinctly represents a large number of token sequences that are consistent with a set of input strings, and can be constructed efficiently in practice.

DEFINITION 2. *An `InputDataGraph` $G = (V, E, I, L)$ is a 4-tuple where V denotes the set of nodes, E denotes the set of edges corresponding to a set of ordered node pairs, $I : V \to \{(id, idx)_i\}_i$ is a labeling function that labels each node with a set of string id and index pair (id, idx), and $L : E \to \{(\tau, k)_i\}_i$ maps each edge to a list of token matches.*

The nodes in an `InputDataGraph` correspond to different indices of a set of strings, where the indices are represented using the labeling function `I`. An edge between two nodes v_1 and v_2 in the graph represents all token matches that match the substrings corresponding to the indices of the two nodes. The token matches on an edge are represented using the edge labeling function `L`. A path in the graph between two nodes represents a sequence of token matches that match the corresponding substrings. In this manner, an `InputDataGraph` represents an exponential number of sequence of token matches succinctly using polynomial space.

Consider the input strings: {`"1 lb"`, `"23 g"`, `"4 tons"`, `"102 grams"`, `"75 kg"`}. The `InputDataGraph` G_1 for the input string `"1 lb"` is shown in Figure 6(a). We have $G_1 = (V_1, E_1, I_1, L_1)$, where $V_1 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$, and $E_1 = \{(v_i, v_j) | j > i \wedge v_i \in V_1 \wedge v_j \in V_1\}$. The node labeling function $I_1(v_i) = \{(1, i)\}$ for all $v_i \in V_1$, where 1 is the unique identifier assigned to the string `"1 lb"`. The edge labeling function L_1 for each edge is shown in the figure, e.g. $L_1(v_1, v_2) = \{(d, 1), (d, \text{-}1), (\alpha n, 1)(\alpha n, \text{-}1), (\text{"1"}, 1), (\text{"1"}, \text{-}1)\}$ and $L_1(v_1, v_3) = \{(\text{"1 "}, 1), (\text{"1 "}, -1)\}$. Similarly, the graph for the input string `"23 g"` is shown in Figure 6(b).

5.1 Generation of InputDataGraphs

Given a set of n input rows each consisting of k columns, the `GenInpDataGraph` algorithm constructs the corresponding `InputDataGraph` G as shown in Figure 7. The key idea of the algorithm is to first construct a graph for each spreadsheet column, and then return the union of these graphs as the resulting graph for the spreadsheet data. Given a column of string values, it uses the `GenGraphStr` algorithm
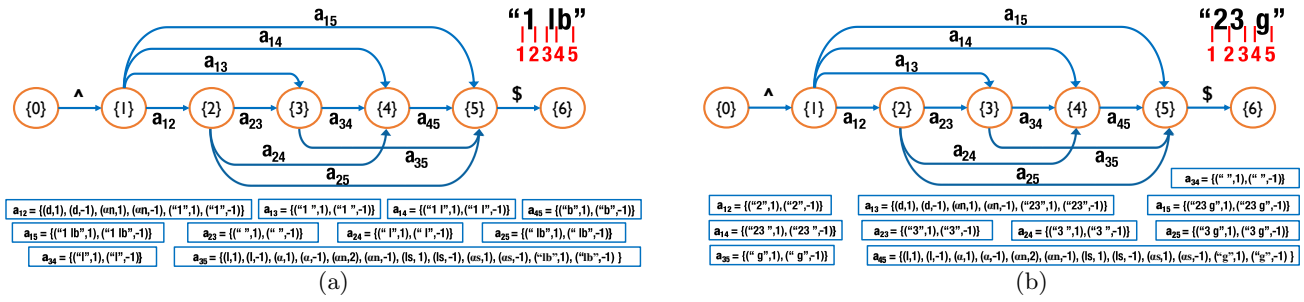
Figure 6: The InputDataGraph for the input strings (a) "1 lb" and (b) "23 g".

$a_{12} = \{(d,1), (d,-1), (\omega n,1), (\omega n,-1), (\text{"1"},1), (\text{"1"},-1)\}$ $a_{13} = \{(\text{"1"},1), (\text{"1"},-1)\}$ $a_{14} = \{(\text{" 1"},1), (\text{" 1"},-1)\}$ $a_{45} = \{(\text{"b"},1), (\text{"b"},-1)\}$

$a_{15} = \{(\text{"1 lb"},1), (\text{"1 lb"},-1)\}$ $a_{23} = \{(\text{" "},1), (\text{" "},-1)\}$ $a_{24} = \{(\text{" l"},1), (\text{" l"},-1)\}$ $a_{25} = \{(\text{" lb"},1), (\text{" lb"},-1)\}$

$a_{34} = \{(\text{"l"},1), (\text{"l"},-1)\}$ $a_{35} = \{(l,1), (l,-1), (\alpha,1), (\alpha,-1), (\omega m,2), (\omega n,-1), (ls, 1), (ls,-1), (\alpha s,1), (\alpha s,-1), (\text{"lb"},1), (\text{"lb"},-1) \}$

$a_{12} = \{(\text{"2"},1),(\text{"2"},-1)\}$ $a_{13} = \{(d,1), (d,-1), (\omega n,1), (\omega n,-1), (\text{"23"},1), (\text{"23"},-1)\}$ $a_{15} = \{(\text{"23 g"},1), (\text{"23 g"},-1)\}$

$a_{14} = \{(\text{"23 "},1), (\text{"23 "},-1)\}$ $a_{23} = \{(\text{"3"},1),(\text{"3"},-1)\}$ $a_{24} = \{(\text{"3 "},1),(\text{"3 "},-1)\}$ $a_{25} = \{(\text{"3 g"},1), (\text{"3 g"},-1)\}$

$a_{35} = \{(\text{" g"},1),(\text{" g"},-1)\}$ $a_{45} = \{(l,1), (l,-1), (\alpha,1), (\alpha,-1), (\omega m,2), (\omega n,-1), (ls, 1), (ls,-1), (\alpha s,1), (\alpha s,-1), (\text{"g"},1), (\text{"g"},-1)\}$

```
GenInpDataGraph({(v₁¹, ⋯ , vₖ¹), ⋯ , (v₁ⁿ, ⋯ , vₖⁿ)})
  for i = 1 to k:
      Gᵢ := GenGraphColumn({vᵢ¹, ⋯ , vᵢⁿ})
  return ⋃_{1≤i≤k} Gᵢ

GenGraphColumn({(s₁, ⋯ , sₙ)})
  G := GenGraphStr(s₁)
  for i = 2 to n:
      G := Intersect(G, GenGraphStr(sᵢ))
  return G
```

Figure 7: The GenInpDataGraph algorithm for constructing the input graph for a set of n input rows each consisting of k strings.

to construct the graphs for individual strings and then uses the intersection algorithm (§5.2) to intersect these graphs to compute the graph consisting of all logical patterns that are shared across the set of column strings.

The **GenGraphStr** algorithm for constructing the input graph for an input string s is shown in Figure 8. The algorithm first creates $\text{len(s)} + 1$ number of nodes for denoting different indices of the input string, and two special nodes v_0 and $v_{len(s)+2}$ for denoting the start and end tokens respectively. The labeling function I labels each node v_i with (id,i), where id denotes the unique string identifier for s. For each node pair (v_i, v_j), the algorithm adds an edge with the labeling function L such that $L(v_i, v_j)$ consists of all token matches that match the substring $s[i..(j − 1)]$. The complexity of the algorithm is $O(n^2)$ (by using hash lookup for **GetMId**), where n is the length of the input string s.

5.2 Intersection of InputDataGraphs

The intersection of two **InputDataGraphs** results in an **InputDataGraph** that consists of all patterns that are common to both graphs. Formally, the intersection of two graphs $G_1 = (V_1, E_1, I_1, L_1)$ and $G_2 = (V_2, E_2, I_2, L_2)$ is defined as $G = \text{Intersect}(G_1, G_2) = (V, E, I, L)$, where

$$V = \{(v_i, v_j) | v_i \in V_1, v_j \in V_2\}$$
$$E = \{((v_i, v_j), (v_k, v_l)) | (v_i, v_k) \in E_1, (v_j, v_l) \in E_2\}$$
$$I((v_i, v_j)) = I_1(v_i) \cup I_2(v_j), \forall v_i \in V_1, v_j \in V_2$$
$$L(((v_i, v_j), (v_k, v_l))) = \{(\tau, k) | (\tau, k) \in L_1((v_i, v_k))$$
$$\wedge (\tau, k) \in L_2((v_j, v_l))\} \forall (v_i, v_k) \in E_1, (v_j, v_l) \in E_2$$

The **Intersect** algorithm creates the Cartesian product of the nodes in the two graph V_1 and V_2 to create the nodes V for the new graph G. The node labeling function $I((v_i, v_j))$

```
GenerateInputGraph(s:  Input string)
1   V = ∅, E = ∅
2   id = string2Id[s]
3   foreach i ∈ range(0, len(s)+3):
4       V = V ∪ vᵢ
5       I(vᵢ) = { (id,i) }
6   L((v₀, v₁)) = {(^, 1)}
7   L((v_{len(s)+1}, v_{len(s)+2})) = {($, 1)}
8   foreach i ∈ range(1, len(s)+1):
9       foreach j ∈ range(i+1, len(s)+2):
10          leftIdx = i, rightIdx = j-1
11          E = E ∪ (vᵢ, vⱼ)
12          cₛ = s[leftIdx..rightIdx]
13          L((vᵢ, vⱼ)) = { (cₛ, GetMId(cₛ, s, i)) }
14          foreach τ ∈ 𝒯 ∧ Match(τ, cₛ):
15              L((vᵢ, vⱼ)) = L((vᵢ, vⱼ)) ∪ (τ, GetMId(τ, s, i))
16  return (V, E, I, L)
```

Figure 8: The generateInputGraph algorithm for constructing the input graph for an input string s.

takes the union of the labels of the corresponding nodes $I(v_i)$ and $I(v_j)$. For each edge $(v_i, v_k) \in E_1$ and edge $(v_j, v_l) \in E_2$, the algorithm adds an edge $((v_i, v_j), (v_k, v_l))$ to the new set of edges E. The edge labeling function $L(((v_i, v_j), (v_k, v_l)))$ labels each edge with the token set that is common to both sets $L((v_i, v_k))$ and $L((v_j, v_l))$. The worst-case complexity of the algorithm is $O(n^4)$, where n is the length of the input strings. However, in practice, we do not see a quadratic blowup in the graph size after the intersection algorithm as shown in §8.3. The key reason behind this is the fact that even though the two initial **InputDataGraphs** are complete with $O(n^2)$ edges, a lot of these edges do not remain after the intersection operation because of the strong constraint that token match (τ, k) should be the same for both the intersecting edges. For example, the token match ("a", 1) (matching the first occurrence of constant string "a") can not match with the token match ("a", 2) (the second occurrence of "a"). As a result, the **InputDataGraph** obtained after intersection becomes very sparse in the number of edges.

The **InputDataGraph** for the intersection of graphs for the strings "1 lb" and "23 g" is shown in Figure 9. The intersected graph consists of 6 nodes and 5 edges. Note that the node labels in the figure only show the string indices and not the string ids for brevity. The **InputDataGraph** obtained after intersecting the graphs for the set of input strings {"1 lb", "23 g", "4 tons", "102 grams", "75 kg"} also consists of same number of nodes and edges. The **InputDataGraph**
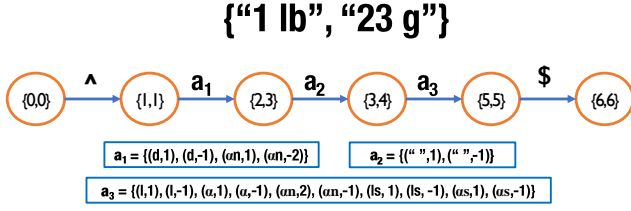
{"1 lb", "23 g"}

Figure 9: The intersection of `InputDataGraphs` for strings "1 lb" and "23 g".

for the six input strings in Example 1 is shown in Figure 14, which consists of 40 nodes and 32 edges.

6. STRING TRANSFORMATIONS

In this section, we describe the String Transformation Language L_s of BLINKFILL and the `Dag` data structure used to succinctly represent a large number of L_s expressions that are consistent with a given set of input-output examples. The DSL and the data structure are similar to the corresponding DSL and data structure of FLASHFILL at the top-level, but the key difference is in the representation of position expressions in the substring expressions, which are defined using the token match expressions from the edges of the `InputDataGraph` data structure.

6.1 String Transformation Language L_s

The DSL for BLINKFILL is shown in Figure 10(a) with the key differences from FLASHFILL highlighted. The top-level string expression e is a concatenation of a finite list of substring expressions $[f_1, \cdots, f_n]$. A substring expression f can either be a constant string s (denoted by `ConstantStr(s)`) or a substring extraction expression that is defined using two position logics p_l and p_r, which correspond to the left and right indices in the input string v_i respectively. The position logic expression p is either a constant integer value k or a token match expression. The token match expression (τ, k, Dir) returns the start index ($\text{Dir} = \text{Start}$) or the end index ($\text{Dir} = \text{Start}$) of the token match (τ, k).

The semantics of L_s is shown in Figure 10(b). The environment state σ maps the input variables v_i to their corresponding string values. The semantics of a concatenate expression is to recursively evaluate each individual substring expression f_i and then concatenate them. The semantics of a constant string expression `ConstantStr(s)` is to simply return the constant string s. The semantics of a substring expression `SubStr(v_i, p_l, p_r)` is to first evaluate index positions $l = p_l$ and $r = p_r$, and then return the corresponding substring $s[l..r]$, where $s = \sigma(v_i)$. The semantics of the constant position expression `ConstantPos(k)` is to return k if $k > 0$ and $\text{len(s)} + k$ otherwise. The semantics of a token match expression (τ, k, Start) (resp. (τ, k, End)) is to return the start index (resp. end index) of the kth occurrence of the match of token τ in the string s.

A possible L_s expression to perform the desired transformation of extracting country names in Example 1 is: $e_1 \equiv \text{Concat}(\text{SubStr}(v_1, (\text{", "}, 1, \text{End}), (l, -1, \text{End})))$. The expression e_1 extracts the substring from the input string v_1 whose left index is the end position of the match of 1st constant string `", "` (comma followed by a whitespace) and the right index is the end position of the match of last lowercase

token match. For example, for the input string `"Mumbai, India"`, the position expression $(\text{", "}, 1, \text{End})$ returns the index 9 and matches the substring `", "`, whereas the position expression $(l, -1, \text{End})$ returns the index 14 and matches the substring `"ndia"`. For abbreviating the initials of first and last names in Example 2, a possible L_s expression is: $e_2 \equiv \text{Concat}(f_1, \text{ConstantStr}(\text{"."}), f_2, \text{ConstantStr}(\text{"."}))$, where $f_1 \equiv \text{SubStr}(v_1, (C, 1, \text{Start}), (C, 1, \text{End}))$ and $f_2 \equiv \text{SubStr}(v_1, (C, -1, \text{Start}), (l, -1, \text{Start}))$.
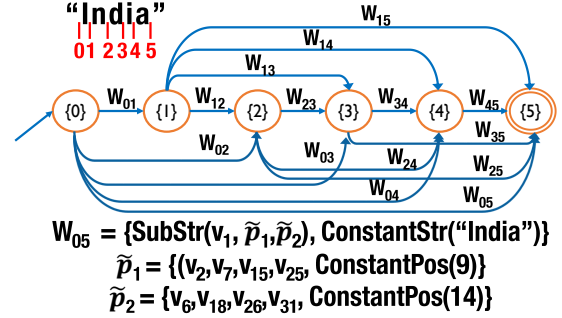
6.2 DAG Data Structure

Figure 12: The `Dag` data structure for representing a set of string expressions to generate the string "India" from an input string "Mumbai, India".

The `Dag` data structure is used to succinctly represent a large number of L_s expressions that are consistent with a given set of input-output examples. The syntax of the `Dag` data structure is shown in Figure 11(a). The set of top-level concatenate expressions are represented using paths in the `Dag` in a way similar to their `Dag` based representation in FLASHFILL. The key difference is in the way the substring expressions are represented as a pair of two independent sets of `InputDataGraph` nodes on the `Dag` edges.

A set of L_s expressions is represented using a `Dag` consisting of a set of nodes $\tilde{\eta}$, a start node η^s, a final node η^f, a set of edges $\tilde{\xi}$, and a mapping function W that maps each edge $\xi \in \tilde{\xi}$ to a set of substring expressions. The set of substring expressions f consists of either a constant string expression or a set of substring expressions $\text{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}'_k\}_k)$, where the set of position expressions $\{\tilde{p}_j\}_j$ and $\{\tilde{p}'_k\}_k$ are represented as independent sets. A set of position expressions $\tilde{p}$ consists of either a constant position or a set of `InputDataGraph` nodes $\widetilde{V}$.

The semantics of the `Dag` data structure is shown in Figure 11(b). A path from the start node η^s to the final node η^f in the `Dag` represents a $\text{Concat}(f_1, \cdots, f_n)$ expression, where the substring expressions f_i are obtained from the edges ξ_i on the path. The semantics of a set of substring expressions $\text{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}'_k\}_k)$ is to first independently evaluate the set of left and right position expressions, and then compute the set of substring expressions corresponding to the cross-product of the set of position expressions. The set of constant string and constant position expressions have expected semantics as shown in the figure. Intuitively, the semantics of a set of `InputDataGraph` nodes $\widetilde{V}$ is to return the set of token matches on the edge labels for all incoming and outgoing edges to nodes $v \in \widetilde{V}$. For each node $v \in \widetilde{V}$ and

$$
\begin{array}{rcl}
\text{String expr } e & := & \texttt{Concat}(f_1, \cdots, f_n) \\
\text{Substring expr } f & := & \texttt{ConstantStr}(s) \\
& | & \texttt{SubStr}(v_i, p_l, p_r) \\
\text{Position } p & := & (\tau, k, \texttt{Dir}) \\
& | & \texttt{ConstantPos}(k) \\
\text{Direction Dir} & := & \texttt{Start} \mid \texttt{End}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{Concat}(f_1, \cdots, f_n)]\!]_\sigma & = & \texttt{Concat}([\![f_1]\!]_\sigma, \cdots, [\![f_n]\!]_\sigma) \\
[\![\texttt{ConstantStr}(s)]\!]_\sigma & = & s \\
[\![\texttt{SubStr}(v_i, p_l, p_r)]\!]_\sigma & = & s[[\![p_l]\!]_s..[\![p_r]\!]_s], \text{where } s = \sigma(v_i) \\
[\![\texttt{ConstantPos}(k)]\!]_s & = & k > 0?k : \texttt{len}(s) + k \\
[\![(\tau, k, \texttt{Start})]\!]_s & = & k > 0?(\tau \vDash s)[k] \hookrightarrow : (\tau \vDash s)[\texttt{len}(s) + k] \hookrightarrow \\
[\![(\tau, k, \texttt{End})]\!]_s & = & k > 0?(\tau \vDash s)[k] \hookleftarrow : (\tau \vDash s)[\texttt{len}(s) + k] \hookleftarrow
\end{array}
$$

(a)   (b)

Figure 10: The (a) syntax and (b) semantics of the string transformation language L_s of BlinkFill.

$$
\begin{array}{rcl}
\tilde{e} & := & \texttt{Dag}(\tilde{\eta}, \eta^s, \eta^f, \tilde{\xi}, W) \\
\tilde{f} & := & \texttt{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}'_k\}_k) \\
& | & \texttt{ConstantStr}(s) \\
\tilde{p} & := & \texttt{ConstantPos}(k) \\
& | & \widetilde{V}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{Dag}(\tilde{\eta}, \eta^s, \eta^f, \tilde{\xi}, W)]\!] & = & \{\texttt{Concat}(f_1, \cdots, f_n) | f_i \in [\![W(\xi_i)]\!] \\
& & \xi_1 \rightsquigarrow \xi_n \text{corresponds to path between } \eta^s \text{ and } \eta^f\} \\
[\![\texttt{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}'_k\}_k)]\!] & = & \{\texttt{SubStr}(v_i, p_l, p_r) | p_l \in [\![\tilde{p}_j]\!], p_r \in [\![\tilde{p}'_k]\!]\} \\
[\![\texttt{ConstantStr}(s)]\!] & = & \{\texttt{ConstantStr}(s)\} \\
[\![\texttt{ConstantPos}(k)]\!] & = & \{\texttt{ConstantPos}(k)\} \\
[\![\widetilde{V}]\!] & = & \{(\tau, k, \texttt{Start}) | v \in \widetilde{V}, \exists v_i \in \texttt{V} : (v, v_i) \in \texttt{E}, (\tau, k) \in \texttt{L}((v, v_i))\} \\
& & \cup \{(\tau, k, \texttt{End}) | v \in \widetilde{V}, \exists v_i \in \texttt{V} : (v_i, v) \in \texttt{E}, (\tau, k) \in \texttt{L}((v_i, v))\}
\end{array}
$$

(a)   (b)

Figure 11: The (a) syntax and (b) semantics of the `Dag` data structure used to succinctly represent a large number of L_s expressions.

outgoing edge $(v, v_i) \in \texttt{E}$, the resulting set consists of a token match expression $(\tau, k, \texttt{Start})$, where $(\tau, k) \in \texttt{L}((v, v_i))$. Similarly, it also consists of token matches $(\tau, k, \texttt{End})$ for each $v \in \widetilde{V}$, $(v_i, v) \in \texttt{E}$, $(\tau, k) \in \texttt{L}((v_i, v))$.

The `Dag` data structure that succinctly represents the set of all L_s expressions consistent with the input-output example `"Mumbai, India"` $\rightarrow$ `"India"` is shown in Figure 12, where the set of nodes $\tilde{\eta} = \{0, 1, 2, 3, 4, 5\}$, the start node $\eta^s = 0$, the final node $\eta^f = 5$, and $\tilde{\xi} = \{(i, j) | 0 \le i < j \le 5\}$. Intuitively, the `Dag` nodes correspond to the indices of the output string `"India"` and an edge from node i to node j represents the set of substring expressions that can generate the substring between indices i and j. The edge mapping function W is shown for one of the edges $(0, 5)$ (corresponding to the sub-string `"India"`), where $W((0, 5))$ consists of a set of substring expressions and a constant string expression. The set of position expressions for the left index $\tilde{p}_l$ consists of a constant position expression and 4 nodes from the `InputDataGraph` in Figure 14 : $\{v_2, v_7, v_{15}, v_{25}\}$. The node labeling function for these nodes contains the tuple $(\texttt{id}, 9)$, where `id` is the unique identifier for the input string `"Mumbai, India"` and 9 is the left index of sub-string `"India"` in the input string. Similarly, the set of position expressions for the right index also consists of 4 input graph nodes and a constant position expression.

7. SYNTHESIS ALGORITHM

The `LearnProgram` algorithm for learning an L_s expression that conforms to a given set of m input-output examples $\{((v_1^1, \cdots, v_k^1), o_s^1), \cdots, ((v_1^m, \cdots, v_k^m), o_s^m)\}$ and n in-

```
LearnProgram({(v₁¹,⋯,v_k¹),⋯,(v₁ⁿ,⋯,v_kⁿ)},
            {((v₁¹,⋯,v_k¹),o_s¹),⋯,((v₁ᵐ,⋯,v_kᵐ),o_sᵐ)})
1  G := GenInpDataGraph({(v₁¹,..,v_k¹),⋯,(v₁ⁿ,..,v_kⁿ)})
2  Dag d := GenerateDag((v₁¹,⋯,v_k¹s),o_s¹,G)
3  for i = 2 to m:
4     Dag d' := GenerateDag((v₁ⁱ,⋯,v_kⁱs),o_sⁱ,G)
5     d := Intersect(d,d')
6  return TopRankExpr(d)
```

Figure 13: The `LearnProgram` procedure for learning an expression that conforms to a set of m examples.

put rows $\{(v_1^1, \cdots, v_k^1), \cdots, (v_1^n, \cdots, v_k^n)\}$ is shown in Figure 13. The algorithm first constructs the `InputDataGraph` G corresponding to the set of n input rows. It then constructs the `Dag` d denoting the set of all consistent L_s expressions that conform to the first input-output example. Next, it iterates over other input-output examples and intersects the corresponding dags to compute the resulting dag d that consists of L_s expressions that are consistent with all m examples. Finally, the algorithm uses a ranking function to find the best expression path in the dag and returns it as the learnt program. We now describe each of the individual algorithms `GenSubStrExpr`, `GenerateDag`, and `TopRankExpr`.

7.1 Learning Substring Expressions

The `GenSubStrExpr` algorithm, shown in Figure 15, takes as input an input string `s`, two integer positions `l` and `r` corresponding to the left and right indices of the desired substring, and the `InputDataGraph` G, and returns the set of
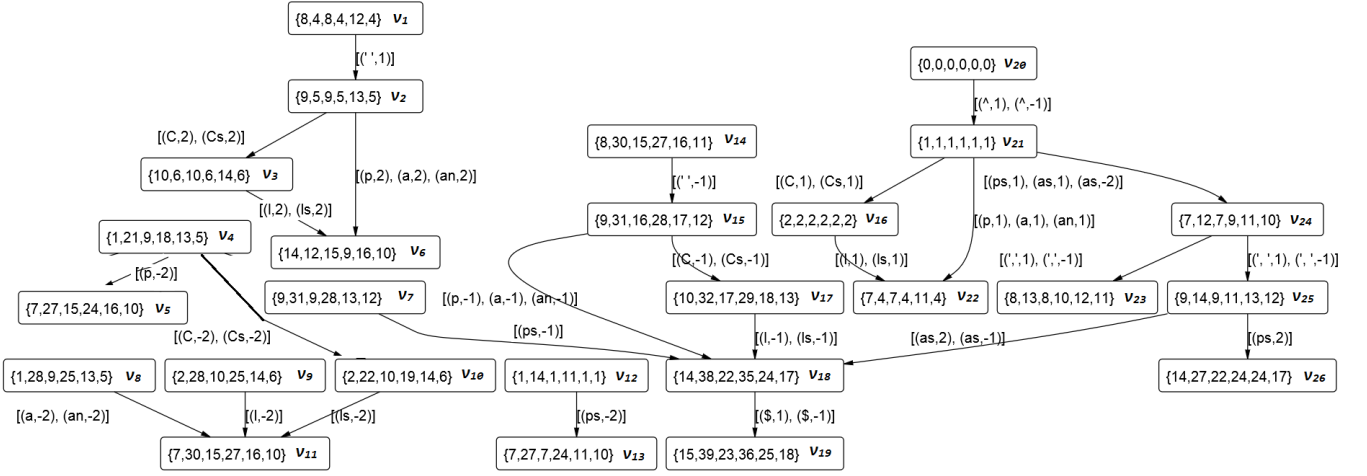
Figure 14: The InputDataGraph for the set of input strings in Example 1. Note that some constant string edges and the corresponding nodes are not shown here for brevity.

```
GenSubStrExpr(s, l, r, G)
1  id = string2Id[s]
2  Ṽ_l = ∅,  Ṽ_r = ∅
3  foreach v ∈ V(G):
4      if (id,l) ∈ I(v):   Ṽ_l = Ṽ_l ∪ {v}
5      if (id,r) ∈ I(v):   Ṽ_r = Ṽ_r ∪ {v}
6  p̃_l = Ṽ_l ∪ {ConstantPos(l)}
7  p̃_r = Ṽ_r ∪ {ConstantPos(r)}
8  return SubStr(s, p̃_l, p̃_r)
```

Figure 15: The GenSubStrExpr algorithm for generating a set of substring expressions given an input string s, the left and right indices l and r of the substring, and the InputDataGraph G.

substring expressions that can generate the corresponding substring. It searches the graph G for all nodes $v \in V(G)$ such that $(\mathtt{id}, l) \in I(v)$ to create the set of nodes $\widetilde{V}_l$ corresponding to the left index l of the substring. Similarly, the algorithm constructs the node set $\widetilde{V}_r$ corresponding to the right index r. It finally adds the constant position logics $\mathtt{ConstantPos}(l)$ and $\mathtt{ConstantPos}(r)$ to the respective set of position expressions $\tilde{p}_l$ and $\tilde{p}_r$, and returns the set of substring expressions $\mathtt{SubStr}(s, \tilde{p}_l, \tilde{p}_r)$.

An example substring extraction task from Example 1 is to extract the substring `"India"` from $s = $ `"Mumbai, India"` ($\mathtt{GenSubStrExpr}(s, 9, 14, \mathtt{G})$), where 9 and 14 respectively denote the left and right index nodes of the substring `"India"` in the input graph, and G denotes the InputDataGraph shown in Figure 14. Let `id` denote the unique identifier for the input string s. For the left index 9, the `GenSubStrExpr` algorithm adds all nodes in G to $\tilde{p}_l$ whose labeling function I contains the tuple $(\mathtt{id}, 9)$. From the figure, we observe that there are 4 such nodes in G: $\{v_2, v_7, v_{15}, v_{25}\}$. It also adds the constant position expression $\mathtt{ConstantPos}(9)$ to the set. Similarly, for the right index 14, the algorithm adds all nodes whose label contains the tuple $(\mathtt{id}, 14)$ and $\mathtt{ConstantPos}(14)$ to the set $\tilde{p}_r$. Finally, the algorithm returns the set of substring expression W_{05} as shown in Figure 12.

7.2 Learning Dag Data Structure

The `GenerateDag` algorithm for learning the `Dag` data structure is similar to the `Dag` learning algorithm of FLASHFILL. The algorithm takes as input an input row $\{v_1, \cdots, v_k\}$, an output string o_s, and an InputDataGraph G, and returns a Dag that represents all string expressions in the language L_s that can transform the input strings to the output string. The algorithm first creates $\mathtt{len}(o_s)$ number of nodes with labels $\tilde{\eta} = \{0, \cdots, \mathtt{len}(o_s)\}$, and sets the start node η^s to be the node with label 0 and the final node η^f with label $\mathtt{len}(o_s)$. It then iterates over all substrings $o_s[i..j]$ of the output string, and adds an edge (i, j) between the nodes with labels i and j. For each edge (i, j), the algorithm learns the function W that maps the edge to a constant string expression $\mathtt{ConstantStr}(o_s[i..j])$ and a set of substring expressions obtained by calling $\mathtt{GenSubStrExpr}(v_k, l, r, G)$ (for each (k, l, r) such that $v_k[l..(r - 1)] = o_s[i..j]$).

The Dag data structure learnt by `GenerateDag` algorithm for the input-output example $v_1 = $ `"Mumbai, India"` $\rightarrow$ $o_s = $ `"India"` is shown in Figure 12. The algorithm first creates $\mathtt{len}(o_s) = 6$ nodes labeled $\tilde{\eta} = \{0, \cdots, 5\}$, with the starting node $\eta^s = 0$ and final node $\eta^f = \mathtt{len}(o_s) = 5$. The algorithm then adds an edge (i, j) between nodes i and j, and assigns the labeling function $W((i, j))$ to the set of constant string expressions and substring expressions that can generate the substring $v_1[i..(j - 1)]$.

Dag Intersection: Given two dags, the intersection algorithm computes a dag that consists of expressions common to both the dags. The main idea in the intersection procedure is to compute the Cartesian product of the nodes in the two dags, such that for a node $\eta_1 \in \mathtt{Dag}_1$, and a $\eta_2 \in \mathtt{Dag}_2$, we have a node (η_1, η_2) in the resulting dag. The set of labels on the edges between these nodes is computed by intersecting the corresponding pair of edges in the two dags. For intersecting two sets of substring expressions, the corresponding sets of left and right position expressions are intersected independently. For intersecting the position expressions $\widetilde{V}_1$ and $\widetilde{V}_2$, the intersection algorithm selects the set of nodes v that are common to both $\widetilde{V}_1$ and $\widetilde{V}_2$. The worst-case complexity of the `LearnProgram` algorithm is $O((k|s|^2)^m)$, where

```
RankInpGNodes(G)
 foreach v ∈ V(G):
    v.out := 0, v.in := 0, v.score := 0
 foreach v ∈ V(G) in topological order:
    foreach (v, v_i) ∈ E(G):
        v.out := Max(v.out, v_i.out + φ_η(v, v_i))
 foreach v ∈ V(G) in reverse topological order:
    foreach (v_i, v) ∈ E(G):
        v.in := Max(v.in, v_i.in + φ_η(v_i, v))
 foreach v ∈ V(G):
    v.score := v.in + v.out
 return v with the highest v.score
```

$$\phi_\eta(v_1, v_2) := \Sigma_{\mathtt{id} \in \mathrm{I}(v_1)} |v_2[\mathtt{id}] - v_1[\mathtt{id}]|$$

Figure 16: The ranking algorithm for assigning scores to the nodes of an InputDataGraph G.

$|s|$ denotes the length of output string, k is the size of G, which is bounded by $O(|i|^{4n})$.

7.3 Ranking of Dag Expressions

We now describe the ranking strategy used by BLINKFILL to select the desirable expression amongst the large number of expressions represented by a Dag. The key idea is to first prefer token sequences that have larger contexts around them as they are more likely to correspond to the desired transformation logic. The algorithm then computes the best concatenation of substring expressions in a Dag using Dijkstra's algorithm. There are several other alternative ranking strategies. One ranking strategy is to use Occam's razor principle to prefer shortest and simplest token sequences, which has previously been shown to perform quite poorly on this dataset [27]. Another strategy is to use machine learning techniques to learn a ranking function over features of the expressions based on observed preferences. We compare our ranking strategy with this machine learning based ranking in FLASHFILL in §8.2. It is important to note that ranking only aims to reduce the number of examples needed to learn the transformation and does not have any effect on the completeness of the synthesis algorithm. Since our synthesis algorithm is complete, a user can always provide additional examples to learn the intended transformation in case ranking fails to learn it from fewer examples.

Ranking Substring Expressions We now describe the RankInpGNodes algorithm that assigns scores to nodes in an InputDataGraph G to find token matches with longest contexts. The algorithm maintains two scores for each node $v \in V(G)$: $v.in$ for incoming score and $v.out$ for the outgoing score. The $v.in$ score corresponds to the number of nodes from which there exists a path to v, whereas the $v.out$ score captures the number of nodes that are reachable from v. The algorithm also uses the node distance function ϕ_η to assign higher scores to nodes that are farther (in terms of string indices) to prefer longer token matches. Finally, the algorithm returns the node v with the highest score $v.in + v.out$. For the nodes in expressions $\tilde{p}_l$ and $\tilde{p}_r$ from Figure 12, the ranking algorithm assigns highest score to node v_{25} (corresponding to token match $(",", 1, \mathtt{End})$) and v_{18} (corresponding to token match $(\$, 1, \mathtt{Start})$) respectively.

Ranking Dag Paths The ranking algorithm first assigns a score to each individual edge of the DAG, and then uses an efficient modification of Dijkstra's shortest path algorithm for DAGs (by considering nodes in topological order) to return the maximum score path. A constant string expression $\mathtt{ConstantStr}(s)$ is assigned a low score proportional to the length of the constant string: $|s|^2 * \epsilon$, where we set $\epsilon = 0.1$ for our experiments. A substring expression corresponding to an output substring s is assigned a higher score: $|s|^2 * \kappa$, where we set $\kappa = 1.5$ for our experiments.

THEOREM 1 (SOUNDNESS). *The `LearnProgram` algorithm is sound, i.e. the program e learnt by the algorithm from a set of input-output examples $\{(\sigma_i = (v_1^i, \cdots, v_k^i), o_s^i)\}_i$ always produces the corresponding output when evaluated on the example inputs: $\forall i : [\![e]\!]_{\sigma_i} = o_s^i$.*

THEOREM 2 (COMPLETENESS). *The `LearnProgram` algorithm is complete, i.e. if there exists an L_s string expression that is consistent with the given set of input-output examples, the algorithm is guaranteed to learn the expression given sufficient input-output examples.*

8. EXPERIMENTS

We now present the experimental evaluation of BLINKFILL on 207 real-world string transformation tasks and compare it with FLASHFILL. We then present an evaluation to show that the intersection of InputDataGraphs does not lead to quadratic blowup in practice. Finally, we present the scalability of learning InputDataGraphs with increasing number of input strings. The experiments were performed on a 6-core Intel Xeon 3.5GHz processor with 32 GB RAM.

Implementation: We have implemented the inductive synthesis algorithm for the string transformation language of BLINKFILL in C# as an add-in for Microsoft Excel as well as a Web app[1]. The implementation also supports casing transformations such as lowercase to propercase etc., but we omit them from the technical section (§6) for clarity of the novel ideas in the semi-supervised learning technique.

Benchmarks: The 207 benchmarks were obtained from the Excel product team and online help forums. These benchmarks constitute a super-set of the benchmarks used for learning the ranking function in FLASHFILL [27] and come from the suite of 235 FLASHFILL benchmarks. We categorize the 235 benchmarks into 5 categories: 1) **Substring**: 88 benchmarks that require only the substring expressions, 2) **Concat**: 92 benchmarks that need concatenation of multiple substring and constant string expressions, 3) **MulCol**: 27 benchmarks that need concatenation of substrings from multiple columns, 4) **Conditionals**: 18 benchmarks that need conditionals, and 5) **Loops**: 10 benchmarks that need loops. Since BLINKFILL currently does not support conditional and loop learning, we only consider 207 benchmarks for the evaluation. Each benchmark consists of 6 to 10 input-output examples. We automate the user interaction model by incrementally providing input-output examples to the two systems until the learnt program generates the expected output for all input strings.

Baseline: We compare the performance of BLINKFILL with that of FLASHFILL. For a fair performance comparison, we remove conditional and loop learning from FLASHFILL so that the two domain-specific languages are comparable and only consists of concatenate expressions over constant string expressions and substring expressions.
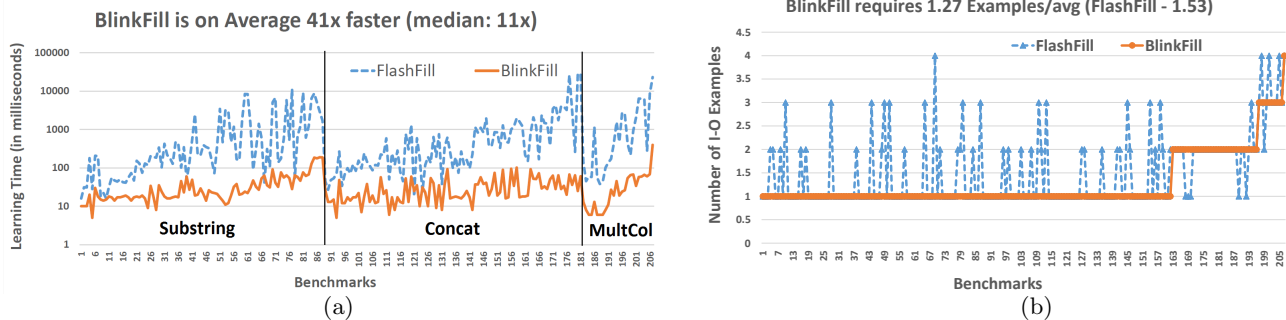
---

[1] http://cleandata.azurewebsites.net/

Figure 17: (a) The learning time for FlashFill and BlinkFill on 207 string transformation benchmarks for the three categories. **(b)** The number of examples required by FlashFill and BlinkFill for different benchmarks.

8.1 Learning Time

We first compare the learning times for BLINKFILL and FLASHFILL on the 207 benchmarks. BLINKFILL learns the desired transformations for all 207 benchmarks in only 7.32 seconds, whereas FLASHFILL requires 324.64 seconds. The learning times for individual benchmark problems for different benchmark category is shown in Figure 17(a). Within each category, we further sort the benchmarks by the average lengths of strings in the input-output examples. In general, learning times increase with increasing string lengths, but several other factors such as number of sub-expressions, number of token matches etc. also influence the learning times. As can be observed, BLINKFILL is significantly faster than FLASHFILL for all the three categories. BLINKFILL is on average $41.5\times$ faster than FLASHFILL and the median speedup is $11.25\times$. The maximum time taken by FLASHFILL on any benchmark is $28.6s$, whereas the maximum time taken by BLINKFILL on any benchmark is only $0.39s$. There are 57 benchmarks where FLASHFILL takes more than $1s$ each to learn the transformation. For these 57 challenging benchmarks, BLINKFILL is on average $124\times$ (median $68.1\times$) faster than FLASHFILL.

8.2 Ranking

BLINKFILL is not only faster than FLASHFILL, but it also learns the transformations from fewer input-output examples. We compare the ranking of the two systems by measuring the number of examples needed to learn the desired transformation, which is shown in Figure 17(b). For the 207 benchmarks, FLASHFILL needs 317 input-output examples in total, whereas BLINKFILL requires only 264 examples. On average, FLASHFILL needs 1.53 examples per benchmark, whereas BLINKFILL requires 1.27 examples. For 44 benchmarks, BLINKFILL requires fewer examples than FLASHFILL, whereas there are 6 benchmarks where FLASHFILL needs fewer examples. Overall, the ranking strategy of selecting token sequences with largest context works remarkably well. The few cases where it doesn't work as well compared to FLASHFILL are the cases where there is some unintended matching of longer contexts. For example, for the input-output example `"457 5th St S, Seattle, WA 98111"` $\rightarrow$ `"from Seattle"`, the system learns the token match (`ws`,-3) (3rd space from end) to extract the left index of the substring `" Seattle"`. This logic works for most inputs but not for an input of the form `"98743 Edwards Ave, Los Angeles, CA 78911"`, where it generates the output `"from Angeles"`. The
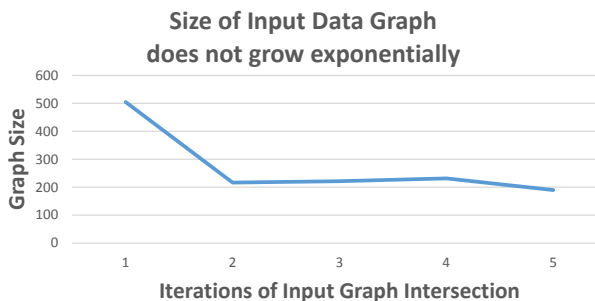
context for the token match (`ws`,-3) ($p \circ$ `","` $\circ$ `ws` $\circ$ C $\circ$ `ws` $\circ$ d) is larger than the context for the desired token match (`","`,1). A user can provide an additional example `"from Los Angeles"` in this case, and the ranking algorithm then learns the correct transformation using the 2 examples.

8.3 Intersection of InputDataGraphs

Even though the complexity of the intersection of two `InputDataGraphs` is $O(n^4)$, we never experience a quadratic blowup in graph size after intersection in practice. All benchmarks exhibit the typical variation in the size of the graph after each intersection as shown in Figure 18. The size of the graph always reduces after the first intersection, and then there is a minor variation in the size for later intersections. We hypothesize the sparsity of edges in the `InputDataGraphs` to be the reason for this phenomenon.

8.4 Scalability on Large Spreadsheets

The construction of the `InputDataGraph` on very large spreadsheets can be a bottleneck. For such large spreadsheets, we randomly sample a constant k number of input rows to compute the `InputDataGraph`. The generation times of input graph for different values of k on a large Excel spreadsheet[2] is shown in Figure 19. The `GenInpDataGraph` algorithm takes less than $1s$ to construct the input graph for $k = 1280$ input strings. However, for all real-word benchmarks, approximately 10 randomly sampled input strings suffices for learning a representative input graph.

Figure 18: The variation in size of the `InputDataGraph` after each intersection operation.

---

[2]The spreadsheet comes from the US SSA website consisting of 33044 popular baby names in 2014 together with their gender and frequency.

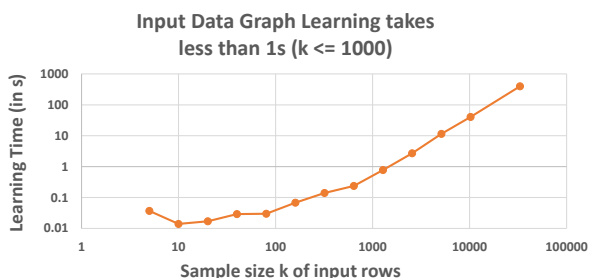**Input Data Graph Learning takes less than 1s (k <= 1000)**

Figure 19: The time taken by the `GenInpDataGraph` algorithm to compute the `InputDataGraph` for k randomly sampled input strings in a large spreadsheet.

9. RELATED WORK

Comparison with FlashFill The domain-specific language and the `Dag` data structure for BLINKFILL at top-level are similar to that of FLASHFILL, but the key difference is in the way position expressions in the substring expressions are represented and learnt using semi-supervised learning. The core technical challenge solved by BLINKFILL is in efficiently learning the set of all token sequences (and sub-sequences) that are consistent with a set of strings in a column using the `InputDataGraph`, and then using it to efficiently learn substring expressions. This results in a significant decrease in the ambiguity of possible transformations that are consistent with a set of input-output examples and makes the learning process dramatically more efficient. This is crucial for increasing the applicability of FLASHFILL for longer strings (which is currently automatically disabled by the Excel team for longer strings). Moreover, learning from data allows BLINKFILL to learn token sequences consisting of arbitrary constant string tokens as opposed to a finite pre-defined tokens supported by FLASHFILL, which has been another major shortcoming of FLASHFILL. Unlike FLASHFILL, BLINKFILL currently does not support conditionals and loops. For learning conditionals, a strategy similar to FLASHFILL can be implemented where the synthesis algorithm in BLINKFILL clusters the input strings into different `InputDataGraphs` if there is no edge in the original `InputDataGraph` that corresponds to the substring extraction task required by the input-output examples.

Version-space Algebra based PBE Systems There has been a lot of recent work on building PBE systems using the VSA methodology. The key idea in building these systems is to first design an expressive DSL that allows for decomposing the top-level specification to make the learning efficient [8]. In addition to FLASHFILL, it has been used to build PBE systems in the domains of semantic string transformations [26], data extraction from unstructured text [14], and re-structuring of semi-structured spreadsheets [2]. Recently, a generic meta-algorithm `FlashMeta` [21] has been proposed to automatically generate an efficient synthesis algorithm based on VSA from simply the description of the DSL, where the DSL is designed using a predefined set of operators. All of these systems synthesize programs by considering only the input-output examples and unlike our technique do not use the information present in other inputs in the learning algorithms.

String Transformation using Examples and Demonstrations Data Wrangler [11] is an interactive system for creating reusable data transformations such as map, joins, aggregation, sorting etc. using examples. It also automatically suggests some transformations based on the context of user interactions. The Topes system [24] allows end-users to interactively implement abstractions (called topes) for validating and transforming data in many different formats. It can also recommend some basic topes given a set of strings from a predefined set of topes based on keyword matches [25]. The analysis of user context in Wrangler and Topes' recommendation, however, is not as rich as that of BLINKFILL in finding complex logical patterns that are shared amongst a set of input strings. LAPIS [18] is a text editor that allows users to perform multiple selections using a set of positive and negative examples, and then perform simultaneous editing on the selections. It is also able to identify some outlier selections corresponding to possible incorrect generalization using machine learning [17]. Our technique can complement LAPIS in not learning such incorrect generalizations since such generalizations won't be part of large common pattern sequences in `InputDataGraph`. DataXFormer [1, 20] is an interactive data transformation and integration system that leverages Web tables and Web forms to perform syntactic and semantic data transformations. The class of transformations supported by DataXFormer are based on relational mapping of a set of strings (with some fuzzy matching capabilities), while the transformations supported by BLINKFILL involve concatenation of logical regular expression based substrings.

Record Alignment The problem of extracting relational tables from lists using record alignment is also related to our technique of learning the set of common patterns shared by a set of strings in a column. TEGRA [3] models the table extraction problem as an optimization problem to capture the conceptual goodness of an extracted table based on both syntactic and semantic information (using Web tables). ListExtract [5] is another system that greedily segments each string using various signals such as delimiters and semantic information, and then uses a majority vote to guess the correct number of segmented columns. BLINKFILL, on the other hand, learns the set of all regular expression based patterns (and sub-patterns) shared by a set of strings, and uses it to guide the synthesis algorithm for learning transformations from examples. However, the techniques from TEGRA and ListExtract can be used to enhance the semi-supervised learning in BLINKFILL by adding the semantic knowledge and by using approximate inference algorithms for increasing robustness to noise.

Ranking in Program Synthesis The problem of ambiguity arises in any inductive synthesis problem where the specification is incomplete. PROSPECTOR [15] synthesizes jungloid code fragments consisting of a chain of objects and method calls from a given input type to an output type, and uses the length of code, generality of the output type, and number of cross-package boundaries as the criterion for ranking possible fragments. LASE [16] helps developers perform systematic program changes automatically by learning from few examples. Recently, machine learning based techniques were used for learning the ranking function automatically for FLASHFILL [27]. These ranking approaches use manually defined scoring (cost) functions, the specification constraint, and some global usage information for designing the ranking functions, but do not typically exploit the information present in other inputs for a given task.

10. LIMITATIONS AND FUTURE WORK

BLINKFILL currently supports only syntactic string transformations. We plan to also support semantic string transformations by enabling DSL designers to declaratively specify the semantic knowledge [28] and by using the string correlation information from Web tables and forms [9, 20]. Although BLINKFILL is able to learn the majority of FLASHFILL real-world benchmarks, it currently does not support conditionals and loops. A strategy similar to FLASHFILL of clustering input strings into different clusters can be used for learning conditionals. The combination of approximate pattern inference techniques used in systems such as Data Wrangler [11] and TEGRA [3] with our VSA based technique can also provide a potential solution to learn conditional transformations. Other interesting extensions include learning substring expressions based on relative logic (e.g. the right position logic is not independent and depends on the left position logic) and learning probabilistic transformations to handle noise.

11. CONCLUSION

PBE techniques are starting to reach mainstream commercial markets for making programming accessible to a much wider audience. FLASHFILL is one such system for enabling Excel users to perform string transformations using examples. Since examples are an under-specification, FLASHFILL needs to learn the desired transformation from a large space of ambiguous choices. In this paper, we presented a semi-supervised learning technique to learn logical patterns present in the input data to guide the synthesis algorithm, which significantly reduces this ambiguity. We have implemented this technique in the BLINKFILL system. Our extensive evaluation shows that BLINKFILL is significantly faster and learns the desired transformation using fewer input-output examples in comparison to FLASHFILL.

12. REFERENCES

[1] Z. Abedjan, J. Morcos, M. N. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. Dataxformer: Leveraging the web for semantic transformations. In *CIDR*, 2015.

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

[3] X. Chu, Y. He, K. Chakrabarti, and K. Ganjam. TEGRA: table extraction by global record alignment. In *SIGMOD*, pages 1713–1728, 2015.

[4] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[5] H. Elmeleegy, J. Madhavan, and A. Y. Halevy. Harvesting relational tables from lists on the web. *VLDB J.*, 20(2):209–226, 2011.

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

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

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

[9] Y. He, K. Ganjam, and X. Chu. SEMA-JOIN: joining semantically-related tables using big table corpora. *PVLDB*, 8(12):1358–1369, 2015.

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

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

[12] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.

[13] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[14] V. Le and S. Gulwani. Flashextract: a framework for data extraction by examples. In *PLDI*, 2014.

[15] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[16] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[17] R. C. Miller and B. A. Myers. Outlier finding: focusing user attention on possible errors. In *UIST*, pages 81–90, 2001.

[18] R. C. Miller and B. A. Myers. LAPIS: Smart editing with text structure. In *CHI*, pages 496–497, 2002.

[19] T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2), 1982.

[20] J. Morcos, Z. Abedjan, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: An interactive data transformation tool. In *SIGMOD*, pages 883–888, 2015.

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

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

[23] J. Rissanen. Paper: Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[24] C. Scaffidi, B. A. Myers, and M. Shaw. Topes: reusable abstractions for validating data. In *ICSE*, pages 1–10, 2008.

[25] C. Scaffidi, B. A. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. In *IUI*, pages 297–306, 2009.

[26] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

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

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