Transforming Spreadsheet Data Types using Examples

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

Abstract

Cleaning spreadsheet data types is a common problem faced by millions of spreadsheet users. Data types such as date, time, name, and units are ubiquitous in spreadsheets, and cleaning transformations on these data types involve parsing and pretty printing their string representations. This presents many challenges to users because cleaning such data requires some background knowledge about the data itself and moreover this data is typically non-uniform, unstructured, and ambiguous. Spreadsheet systems and Programming Languages provide some UI-based and programmatic solutions for this problem but they are either insufficient for the user's needs or are beyond their expertise.

In this paper, we present a programming by example methodology of cleaning data types that learns the desired transformation from a few input-output examples. We propose a domain specific language with probabilistic semantics that is parameterized with declarative data type definitions. The probabilistic semantics is based on three key aspects: (i) approximate predicate matching, (ii) joint learning of data type interpretation, and (iii) weighted branches. This probabilistic semantics enables the language to handle non-uniform, unstructured, and ambiguous data. We then present a synthesis algorithm that learns the desired program in this language from a set of input-output examples. We have implemented our algorithm as an Excel add-in and present its successful evaluation on 55 benchmark problems obtained from online help forums and Excel product team.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; 1.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Human Factor

Keywords Program Synthesis, Probabilistic Synthesis, Spreadsheet Programming, Programming By Examples, Noisy Examples

1. Introduction

Nowadays more and more business organizations and users are maintaining spreadsheets that contain data collected from multiple sources. The heterogeneity of data present in these sources often requires users to format the data into a consistent format to make it usable [2, 13]. This problem of data cleaning presents many challenges for users as cleaning such data requires some semantic background knowledge about the data itself, and moreover this data is typically non-uniform[1], unstructured[2], and ambiguous[3]. In this paper, we consider an important subset of the data cleaning problem, namely *spreadsheet data type cleaning*. Some common spreadsheet data types include date, time, name, address, and phone number. These data types are also prevalent in databases and log files, and many modern programming languages provide support for parsing and formatting them.

Programming languages like C# and `SQL` provide custom format strings for formatting these data types. These format strings are quite rich and can handle a wide variety of data type representations. There are two main drawbacks of using format strings for data cleaning. First, the format strings are complicated and vary a lot across different languages, which makes it hard even for experienced developers to quickly understand and use them. Second, if the data types are present in many different formats which are not known a priori, parsing them using format strings becomes impossible as a developer can not guess all possible formats.

Spreadsheet systems like Microsoft Excel offer an alternate UI-based approach for data type formatting. It provides a collection of predefined custom formats for each data type from which users can select and visualize the resulting format. This makes life easier for end-users but this approach also falls short on two accounts. First, users often want to transform data types in formats that are not present in the finite list of predefined formats. Second, this approach only works if the spreadsheet column consists of data types in a single format. Excel fails to even recognize the data types if they are present in different formats. To perform such formatting in Excel, users currently need to write macros or VB(.NET) scripts.

We present a programming-by-example (PBE) framework of cleaning data types that learns the desired cleaning transformation automatically from a set of input-output examples, and handles non-uniform, unstructured, and ambiguous data. There are two kinds of users of our framework: 1) end-users (e.g. Excel users), and 2) data type designers (e.g. Excel product team). The end-users perform data cleaning transformations by providing only input-output examples whereas the data type designers use the framework to easily add new data type definitions. PBE has been a successful paradigm for spreadsheet transformations, e.g., FlashFill [9, 10] learns syntactic regular expression based string transformations from few input-output examples, but it fails on transformations that require semantic knowledge about the strings.

We propose a domain specific language (DSL) with probabilistic semantics, which is inspired from the work on random interpretation [8, 11]. The DSL has two novel features: (i) declarative parsing, and (ii) transformation functions. The DSL allows data type designers to declaratively specify the data type descriptions and

[1] data present in multiple formats.

[2] data not recognizable with syntactic logic based on regular expressions.

[3] data with multiple interpretations.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837668

the corresponding transformation functions associated with them. From this description, our framework automatically generates a parser and pretty printer for the data type. These parsers encode the semantic knowledge of the data types, which helps in parsing unstructured data type strings. Our framework is parameterized by data type definitions, which allows data type designers to easily add support for more data types (end-users need not worry about these descriptions). A program in the DSL consists of a set of conditional statements. Our DSL, unlike FlashFill DSL, has a probabilistic semantics, which has three key aspects: (i) *approximate predicate matching*, (ii) *joint learning* of data type interpretation, and (iii) *weighted branches*. The first aspect of approximate predicate matching in branches of a program enables the program to execute on non-uniform data by assigning real-valued interpretations to conditional predicates. The second aspect of joint learning helps in disambiguating interpretations of ambiguous data type strings using other strings present in the spreadsheet. The third aspect of weighted branches allows the program to generate a set of weighted output strings, which are used for ranking the output.

We present a synthesis algorithm for learning the desired program in this DSL efficiently from few examples. Given a set of input-output examples, the algorithm first learns the set of all programs in the DSL that conform to the individual examples. Unlike previous work [9, 25, 26] that uses intersection to compute common programs for a set of examples and heuristics to learn conditionals and ranking, our algorithm performs a weighted union of the programs to obtain the resulting program conforming to the set of examples. The intersection approach works well when the dataset is uniform and when there is a large hypothesis space (e.g. as in FlashFill), whereas weighted conditions and union work well in the cases of unstructured and non-uniform datasets and when the hypothesis space is relatively smaller. We also show that our synthesis algorithm is sound and complete for *regular data type strings* (Definition 5).

This paper makes the following key contributions:

- We present a DSL with probabilistic semantics that allows for learning transformations on non-uniform, unstructured, and ambiguous data.
- We present a sound and complete synthesis algorithm that efficiently learns the desired program in the DSL from a few input-output examples.
- We evaluate our algorithm on 55 real-world benchmark problems obtained from help forums and Excel product team. Our system requires at most 2 input-output examples to learn the desired transformation for 89% $(49/55)$ of the benchmarks.

2. Motivating Examples

In this section, we present a few examples taken from online help-forums and Excel product team to motivate our system.

EXAMPLE 1. *An Excel user copied data obtained from three office locations and as a result had dates in three different formats (US, UK, and Chinese)*[4] *as shown in Figure 1. Because of different formats, the user was struggling to write a macro for converting these dates into a consistent format.*

This data is an example of non-uniform structured data as there exist syntactical features (delimiters) in the input to distinguish them. In our system, a user only needs to provide the output for the first row ("`08/21/2010`") and the system then generates the corresponding outputs for the remaining rows (shown in bold for emphasis) by running the synthesized program over the input strings. The system learns that the user is trying to print the month field of the date followed by the day and year fields with "`/`" as the delimiter string between the fields.

[4] A simplified version of http://stackoverflow.com/questions/6642140/unifying-different-date-formats-in-excel-including-periods-duration

	Input v_1	Output
1	08/21/2010	08/21/2010
2	07/24/2010	**07/24/2010**
3	20.08.2010	**08/20/2010**
4	23.08.2010	**08/23/2010**
5	2010-06-07	**07/06/2010**
6	2010-24-08	**08/24/2010**

Figure 1. Formatting dates into a consistent `mm/dd/yyyy` format. The bold entries are generated by executing the synthesized program on the remaining input strings.

At a high level, our system learns a weighted conditional program to transform input dates of the format `mm-dd-yyy` to an output format `mm-dd-yyyy`. The system performs the correct transformation on the input string "`20.08.2010`" even though it hasn't seen an input-output example for the input format `dd.mm.yyyy` as the string has a single unambiguous interpretation. It assigns a low non-zero weight to the match of the two input parses `mm-dd-yyyy` and `dd.mm.yyyy` (using approximate predicate matching) and formats the input string to the desired output "`08/20/2010`". Similarly, the system is able to perform the desired transformation on input strings "`23.08.2010`" (row 4) and "`2010-24-08`" (row 6) using approximate predicate matching. The input date "`2010-06-07`" in row 5 illustrates another interesting aspect of our system. The date is ambiguous and has two possible parses `yyyy-mm-dd` (`7 June 2010`) and `yyyy-dd-mm` (`6 July 2010`). The synthesis algorithm assigns a higher weight to the parse `yyyy-dd-mm` than the parse `yyyy-mm-dd` because of the presence of input "`2010-24-08`" in row 6 that has a common parse `yyyy-dd-mm` (using joint learning). As a result, the synthesized program assigns a higher weight to the output string "`07/06/2010`" than the output string "`06/07/2010`".

Since the output format for the given input-output example is also ambiguous (`mm/dd/yyyy` or `mm/d/yyyy`), the synthesized program generates another output string "`07/6/2010`" but it assigns a slightly higher weight to the output string "`07/06/2010`" because of the data type constraint (encoded by the data type designer) that the day and month fields are more likely to be in similar format (2-digit format). Our system highlights output cells where the weights of corresponding output strings differ by a small threshold value, so that the user can further inspect the generated output and select an alternate output choice if it is not the desired output.

In comparison, FlashFill requires three input-output examples (one for each different date format) to learn a program of the following form:

$$\texttt{if "/"} \in v_1 \texttt{ then } (\texttt{num}_1, \text{"/"}, \texttt{num}_2, \text{"/"}, \texttt{num}_3)$$
$$\texttt{if "."} \in v_1 \texttt{ then } (\texttt{num}_2, \text{"/"}, \texttt{num}_1, \text{"/"}, \texttt{num}_3)$$
$$\texttt{if "-"} \in v_1 \texttt{ then } (\texttt{num}_3, \text{"/"}, \texttt{num}_2, \text{"/"}, \texttt{num}_1)$$

where $\texttt{num}_i$ denotes the i^{th} number token in the input string.

Since FlashFill interprets input strings as a sequence of characters, it is impossible for it to learn a transformation that involves some computation. For example, consider the case where a user wants to format the first date to a format that requires computing the corresponding day of the week: "`08/21/2010`" $\rightarrow$ "`Sat, 21st Aug 2010`". Our system can handle such transformations as it interprets the input string as a date data type (together with its corresponding day, month, and year values) and supports transformation functions such as getting day of the week from the date value.

EXAMPLE 2. *An Excel user had dates imported in various formats (mdyy,mmdyy,mddyy,mmddyy,mdyyyy,mddyyyy,mmddyyyy) as shown in Figure 2 and needed to format them in a uniform format (mmddyyyy) for exporting them to another program*[5].

	Input	Output
1	3179	03011979
2	30179	03011979
2	030179	03011979
2	311979	03011979
2	3011979	03011979
2	03011979	03011979

Figure 2. Formatting dates to `mmddyyyy` format.

An expert on the forum first replied that it is an impossible task, because there is no one way to parse dates of the form `12523` as it has two interpretations `12-5-23` or `1-25-23`, and asked the user to instead standardize the data collection method. The user replied that the data was coming from a third party and as a result can not be standardized. Finally, the expert wrote a script that returned `check` on ambiguous data types. The user tested the script on a spreadsheet of 900 entries and only `20` of them returned `check`, and this made the user ecstatic. Since we did not have access to the original user spreadsheet, we created a set of 910 random input date strings uniformly distributed in the seven different formats and tested our system on it. We found 62 of them (7%) resulted in highlighting because of multiple similarly weighted output strings. This higher number of highlighted (`check`) entries in our case is because of the fact that our generated data was uniformly distributed which was possibly not the case with the user's spreadsheet data. The input data in this example is unstructured, and can not be handled by any previous PBE work that we are aware of (including FlashFill).

EXAMPLE 3. *An Excel user was struggling with formatting names in different forms to a uniform format of initial of first name followed by the last name as shown in Figure 3. Since names are occurring with titles, pedigree, and degree, no simple macro suffices in this case.*

	Input	Output
1	Sam H. Cook	S. Cook
2	Dr. S. R. Barua PhD	**S. Barua**
3	Andrew Hudson	**A. Hudson**
4	Mr. Anand Sagar	**A. Sagar**
5	Dale McDermott Jr.	**D. McDermott**
6	Mrs. Jessie Simpson	**J. Simpson**

Figure 3. Formatting names to initial and last name.

This is another example of unstructured data where no syntactic logic can extract the initial of first name. Our framework lets data type designers define the fields, the regular expressions for matching the fields, as well as field order constraints to enable parsing of many different name formats. For the name data type, the designer describes which strings are likely to be used as titles, pedigrees, and degrees, and that middle names are more likely to occur between first and last names. From this description, the system is able to learn a program that prints a prefix of length 1 of the first name followed by a constant string ". " and the last name.

[5] http://www.excelforum.com/excel-formulas-and-functions/500884-converting-dates-to-8-digits.html

EXAMPLE 4. *An Excel user received time punch data from a client in multiple formats as shown in Figure 4. Some entries were in 12-hour format and some were in 24-hour format with different time zones. The user wanted to format the times in hh:mm:ss AM/PM EST format to conduct various calculations on it afterwards.*

	Input	Output
1	1:34:00 PM CST	02:34:00 PM EST
2	10.06 EST	**10:06:00 AM EST**
3	12:45 PST	**03:45:00 PM EST**
4	7:25:00 AM MST	**09:25:00 AM EST**

Figure 4. Formatting times in different formats and timezones to hh:mm:ss EST format.

For formatting time in different timezones, the data type designer models the hour, minutes, seconds, AM/PM, and timezone fields of the time data type. The system then learns the corresponding computation for AM/PM and time zone conversion.

3. Data Types Descriptions

We now describe our framework for data type designers to define the data type descriptions in a declarative manner. A data type description consists of: i) a set of regular expressions for recognizing its constituent fields, ii) its canonical representation, and iii) `ToCan` and `FromCan` functions that convert a data type instance to and from its canonical instance respectively, from which the framework automatically generates the corresponding parsers and pretty printers.

DEFINITION 1 (Spreadsheet Data Type). *A spreadsheet data type e is defined to be a collection of fields $\{f_1, \cdots, f_n\}$. A field f is called a source field if it can not be calculated from other fields, and is called a derived field otherwise.*

For example, the `Date` data type consists of four fields: `day`, `month`, `year`, and `dayOfWeek`, which we refer to as `d`, `m`, `y`, and `dow` respectively for brevity. The fields `day`, `month`, and `year` are source fields whereas the `dayOfWeek` field is a derived field as it can be computed from the values of the source fields.

3.1 Field Representation

A field `f` in a data type is represented using a finite set of sub-fields $\mathtt{f}_s$. The set of sub-fields for the `Date` data type is shown in Figure 5, where the month field for example is represented using the following three sub-fields:

- $\mathtt{m}_1$: single-digit format, e.g. `5` and `11`.
- $\mathtt{m}_2$: two-digit format, e.g. `08` and `12`.
- $\mathtt{m}_3$: full month name, e.g. `january` and `april`.

Each sub-field $\mathtt{f}_s$ is associated with a regular expression that defines the set of values the sub-field can take, e.g. the regular expression for the $\mathtt{m}_2$ sub-field defines the set of possible values as $\{\mathtt{01}, \cdots, \mathtt{12}\}$. A sub-field $\mathtt{f}_s$ is associated with a function $\mathtt{MatchSF} : \mathtt{f}_s \rightarrow \mathtt{double}$, which encodes the likelihood of a field `f` being in a sub-field $\mathtt{f}_s$ representation. These functions define equal likelihood values for all sub-fields by default but can be overridden by data type designers to specify additional constraints. For example, we override the likelihood values for the `y` sub-field such that $\mathtt{MatchSF}(\mathtt{y}_2) > \mathtt{MatchSF}(\mathtt{y}_1)$, which encodes the constraint that the year field is more likely to be in four-digit $\mathtt{y}_2$ sub-field representation than the two-digit $\mathtt{y}_1$ representation.

A data type is associated with two functions $\mathtt{FieldOrder} : \mathtt{f} * \mathtt{f} \rightarrow \mathtt{double}$ and $\mathtt{InitField} : \mathtt{f} \rightarrow \mathtt{double}$. The value

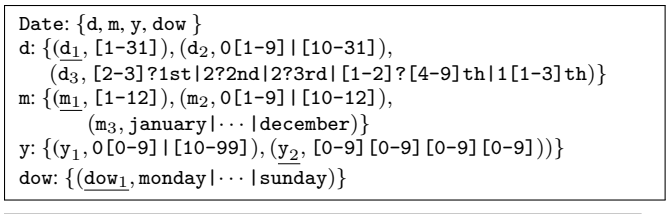


Figure 5. The definitions of fields and sub-field regular expressions for the `Date` data type.

$\texttt{FieldOrder}(\texttt{f}_{s_1}, \texttt{f}_{s_2})$ encodes the likelihood of sub-field $\texttt{f}_{s_2}$ occurring right after the sub-field $\texttt{f}_{s_1}$ in a data type string. We make a simplifying assumption that the field order likelihood depends only on the preceding field. The value $\texttt{InitField}(\texttt{f})$ encodes the likelihood of the field `f` being the first field in a data type string. Similar to the `MatchSF` function, these functions also define equal likelihood values by default and can be overridden for specifying additional constraints. For example for the `Name` data type, we use the `FieldOrder` function to encode the fact that the `middleName` field is likely to follow the `firstName` field and the `pedigree` field is likely to follow the `lastName` field.

3.2 Canonical Representation

Each field of a data type is associated with a *canonical* sub-field (underlined in the figure) that denotes the canonical representation of the field. For example, the sub-field $\texttt{m}_1$ is defined as the canonical sub-field for the `month` field.

DEFINITION 2 (Canonical Instance). *A data type instance v is said to be in its canonical form v_c if the constituent fields have values in their corresponding canonical sub-fields.*

A data type e supports three functions: $\texttt{ToCan} : v \rightarrow v_c$, $\texttt{FromCan} : v_c * \{\texttt{f}_{s_1}, \cdots, \texttt{f}_{s_n}\} \rightarrow v$, and $\texttt{isValidE} : v \rightarrow \texttt{Bool}$. The `ToCan` function converts a data type instance to its canonical form by converting each of its sub-field values to the corresponding canonical sub-field values. It also computes the derived field values for which the required source fields are available. The `FromCan` function performs the inverse operation of converting a data type instance in its canonical form to an instance where the fields take values from a given set of sub-fields. For example, the `ToCan` function for the `Date` data type converts a date instance corresponding to the string "`24th Aug 2011`" to its canonical form $v_C \equiv \texttt{Date}(\texttt{d} \mapsto 24, \texttt{m} \mapsto 8, \texttt{y} \mapsto 2011, \texttt{dow} \mapsto \texttt{wednesday})$. The `FromCan` function on the set of sub-fields $\{\texttt{d}_1, \texttt{m}_2, \texttt{y}_2\}$ converts v_C to the date instance $v_1 \equiv \texttt{Date}(\texttt{d} \mapsto 24, \texttt{m} \mapsto 08, \texttt{y} \mapsto 11)$. The Boolean function `isValidE` checks whether a data type instance is a valid data type interpretation. The `isValidE` function for the `Date` data type returns `false` for a date instance corresponding to the date string "`30 Feb 2011`".

3.3 Format Descriptors

A format descriptor assigns an interpretation to a data type string by pattern matching the string to obtain the corresponding field values.

DEFINITION 3 (Format Descriptor). *A format descriptor π is defined as a sequence of pairs $(\texttt{f}_s, \delta)$ (sub-field and delimiter pair) together with an initial delimiter string δ_0.*

Consider the format descriptor $\pi_1 \equiv (\epsilon, \langle(\texttt{d}_2, \texttt{/}), (\texttt{m}_2, \texttt{/}), (\texttt{y}_1, \epsilon)\rangle)$ of the date string "`04/05/08`". It interprets the string as the instance $v_1 \equiv \texttt{Date}(\texttt{d} \mapsto 04, \texttt{m} \mapsto 05, \texttt{y} \mapsto 08)$. The format descriptors are similar to the custom format strings used in programming languages to format common data types, e.g. the format descriptor π_1 is equivalent to the date format string "`dd/mm/yy`" in C# (.NET).

A format descriptor π is associated with a weight $w(\pi)$ that denotes its likelihood, where $w(\pi)$ is computed as:

$$w(\pi) = \texttt{InitField}(\texttt{f}_1) \times \prod_{i=1}^{i=(n-1)} \texttt{FieldOrder}(\texttt{f}_{s_i}, \texttt{f}_{s_{i+1}}) \times \prod_{i=1}^{i=(n)} \texttt{MatchSF}(\texttt{f}_{s_i}), \text{ where } \pi \equiv (\delta_0, \langle(\texttt{f}_{s_1}, \delta_1) \cdots (\texttt{f}_{s_n}, \delta_n)\rangle).$$

We now define the class of regular data type strings, which we will use afterwards for proving the completeness of the synthesis algorithm (Section 6).

DEFINITION 4 (Regular Format Descriptor). *A format descriptor π is defined to be a regular format descriptor if there exists no substring of the delimiter strings of the format descriptor that matches any sub-field regular expression of the data type.*

In other words, a regular format descriptor does not interpret a data type string in a way that results in interpreting a potential field value as a delimiter string. For example, the date string "`1 24-09-1998`" is not associated with any regular format descriptor as a substring of the initial delimiter string "`1 `" matches $\texttt{d}_1$ and $\texttt{m}_1$ sub-fields of `Date`. On the other hand, the string "`Date! 21--11/2010.`" has a regular format descriptor interpretation given by $\pi \equiv (\text{"}\texttt{Date! }\text{"}, \langle(\texttt{d}_1, \texttt{--}), (\texttt{m}_1, \texttt{/}), (\texttt{y}_2, \texttt{.})\rangle)$.

DEFINITION 5 (Regular Data Type String). *A data type string s is defined to be a regular data type string if there exists a regular format descriptor π for interpreting the string s.*

4. Parsing and Printing Data types

In this section, we describe the `getAllParses` and `printFD` functions for parsing and pretty printing data type strings. The `getAllParses` function computes the set of all parses of a given data type string using the data type description. The `printFD` functions pretty prints a data type instance to a data type string according to an output format descriptor.

4.1 Parsing Data types

The `getAllParses` function, shown in Figure 7, first computes the set of all field matches $\mathcal{M}$ for the data type string. It then constructs the parse graph $\mathcal{G}$ from the matches $\mathcal{M}$ and computes the set of format descriptors corresponding to all valid paths in $\mathcal{G}$. It finally returns the set of pairs of data type instances and format descriptors $\{(v_i, \pi_i)\}_i$. We now present a brief description of each one of these components of the `getAllParses` function.

4.1.1 Field Matches

A field match is defined as a 3-tuple $(i, j, \texttt{f}_\texttt{s})$ that corresponds to a match between the substring of data type string $s[i..j]$ and the regular expression associated with the sub-field $\texttt{f}_\texttt{s}$, where $s[i..j]$ denotes the substring $s_i \cdots s_{j-1}$. The set of all field matches in a data type string s is denoted by $\mathcal{M}(s)$ such that $\mathcal{M}(s) = \{(i, j, \texttt{f}_\texttt{s}) \mid \texttt{MatchF}(s[i..j], \texttt{f}_\texttt{s})\}$.

The `MatchF` function performs regular expression based matching and (optional) prefix matches for specified string valued fields. The matches for string valued fields are associated with two additional parameters `p` and `c` that respectively denote the length of the prefix match and the type of casing performed (`upper`, `proper`, or `iden`) to match a string with the sub-field's regular expression. For example, the data type string "`Jul`" matches the $\texttt{m}_3$ sub-field with parameters $\texttt{p} = 3$ and $\texttt{c} = \texttt{proper}$, and is denoted as $(\texttt{m}_3, 3, \texttt{proper})$. A value of ∞ for `p` means that the string matches completely with the sub-field regular expression. The set of field matches for the date string $s_1 =$ "`01/10/99`" shown in Figure 6(a) is given by:

$\mathcal{M}(s_1) = \{(1,2,\mathtt{d}_1),(3,4,\mathtt{d}_1),(3,5,\mathtt{d}_1),(6,7,\mathtt{d}_1),(7,8,\mathtt{d}_1),$
$(0,2,\mathtt{d}_2),(3,5,\mathtt{d}_2),(1,2,\mathtt{m}_1),(3,4,\mathtt{m}_1),(3,5,\mathtt{m}_1),(6,7,\mathtt{m}_1),$
$(7,8,\mathtt{m}_1),(0,2,\mathtt{m}_2),(3,5,\mathtt{m}_2),(0,2,\mathtt{y}_1),(3,5,\mathtt{y}_1),(6,8,\mathtt{y}_1)\}$.

The `ComputeMatches` function computes the set of all field matches of a data type string s. It performs matching of sub-field regular expressions with substrings of s in $O(k|s|^2)$ time, where k is the number of sub-fields in the data type and $|s|$ denotes the length of the data type string s.

4.1.2 Parse Graphs

A parse graph $\mathcal{G}$ is a directed acyclic graph (DAG) that succinctly encodes the set of all possible parses (interpretations) of a data type string s. The `ConstructParseGraph` function constructs the parse graph of s in the following manner. For each field match $(i,j,\mathtt{f_s}) \in \mathcal{M}(s)$, the graph consists of a node n with a label $\mathcal{L}(n) = (i,j,\mathtt{f_s})$. There are two additional nodes in the graph: the start node n_0 with $\mathcal{L}(n_0) = (-1,-1,\epsilon)$ and the final node n_∞ with $\mathcal{L}(n_\infty) = (\infty,\infty,\epsilon)$. There exists an edge from node n_1 to n_2 in $\mathcal{G}$ with $\mathcal{L}(n_1) = (i_1,j_1,\mathtt{f}_{\mathtt{s}_1})$ and $\mathcal{L}(n_2) = (i_2,j_2,\mathtt{f}_{\mathtt{s}_2})$ if $\forall\,(i,j,\mathtt{f_s}) \in \mathcal{M}(s)$: (i) n_2 represents one of the first field matches that occur after n_1 ($(i \geq j_1) \implies (i \geq i_2)$), and (ii) there exists no other field match that starts before j_1 and overlaps in the interval j_1 to i_2 ($(i < j_1) \implies (j \leq j_1 \vee j_1 = i_2)$). The complexity of constructing a parse graph from a set of matches of size $O(k|s|^2)$ is $O(k^2|s|^4)$.

The parse graph $\mathcal{G}$ for the date string s_1 = "`01/10/99`" is shown in Figure 6(b). There is no edge between nodes $(-1,-1,\epsilon)$ and $(1,2,\mathtt{d}_1)$ since node $(1,2,\mathtt{d}_1)$ is not amongst the set of first field matches that occur after the node $(-1,-1,\epsilon)$ (violates condition (i), e.g. node $(0,2,\mathtt{d}_2)$ occurs before). Similarly, there is no edge between nodes $(3,4,\mathtt{d}_1)$ and $(6,7,\mathtt{d}_1)$ as there exists a node $(3,5,\mathtt{d}_1)$ that overlaps in the interval $(4,6)$ (violates condition (ii)).

DEFINITION 6 (Valid Data Type Path). *We define a path $\mathtt{p}$ in the parse graph $\mathcal{G}$ to be a valid data type path if the following holds:*

1. *$\mathtt{p}$ starts from start node n_0 and ends at final node n_∞.*
2. *no field $\mathtt{f}$ is repeated in the field matches of the nodes in $\mathtt{p}$, i.e. $\forall n_1,n_2 \in \mathtt{p}:\ \mathcal{L}(n_1) = (i,j,\mathtt{f}_s), \mathcal{L}(n_2) = (i',j',\mathtt{f}'_s)$, we have $\mathtt{f} \neq \mathtt{f}'$, where $\mathtt{f}$ and $\mathtt{f}'$ denote the fields of sub-fields $\mathtt{f}_s$ and $\mathtt{f}'_s$ respectively.*

The path $\mathtt{p}_1 \equiv [(-1,-1,\epsilon) \rightarrow (0,2,\mathtt{d}_2) \rightarrow (3,5,\mathtt{m}_1) \rightarrow (6,8,\mathtt{y}_1) \rightarrow (\infty,\infty,\epsilon)]$ in the parse graph in Figure 6(b) represents a valid date path. On the other hand, the path $\mathtt{p}_2 \equiv [(-1,-1,\epsilon) \rightarrow (0,2,\mathtt{d}_2) \rightarrow (3,5,\mathtt{d}_1) \rightarrow (6,8,\mathtt{y}_1) \rightarrow (\infty,\infty,\epsilon)]$ does not represent a valid date path as the field `d` occurs twice on the path. The `GetValidPaths` function computes the set of all valid data type paths in a parse graph $\mathcal{G}$ by performing a `DFS` traversal on the graph while checking for the valid path conditions on-the-fly.

4.1.3 Format Descriptors from Valid Data Type Paths

A valid data type path $\mathtt{p} \equiv [n_0 \rightarrow \cdots (i_k,j_k,\mathtt{f}_{\mathtt{s}_k}) \rightarrow \cdots n_\infty]$ for a data type string s is converted to its corresponding format descriptor $\pi_\mathtt{p} = (\delta_0, \langle(\mathtt{f}_{\mathtt{s}_k},\delta_k)\rangle_k)$ using the `ConvertFD` function in the following way. The function sets the initial delimiter string δ_0 to the substring of s before the start index of the first field match on the path, i.e. $\delta_0 = s[0..i_1]$. For each field match $(i_k,j_k,\mathtt{f}_{\mathtt{s}_k})$ on the path $\mathtt{p}$ ($1 \leq k < m$), it adds $(\mathtt{f}_{\mathtt{s}_k},\delta_k)$ to $\pi_\mathtt{p}$, where $\delta_k = s[j_k..i_{k+1}]$. It finally sets the delimiter string of the final delimiter string δ_m of $\pi_\mathtt{p}$ to the substring $s[j_m..\mathtt{len}(s)]$. Before returning the format descriptor, the function also checks if the entity instance $v_\mathtt{p}$ corresponding to the format descriptor is a valid data type interpretation by applying the `isValidE` function.

```
getAllParses(s:string) : {(v_i, π_i)}_i
 M := ComputeMatches(s)
 G := ConstructParseGraph(M)
 P := GetValidPaths(G)
 parses := ∅
 foreach path p ∈ P :
    π_p := ConvertFD(p)
    parses := parses ∪ (GetInstance(π_p), π_p)
 return parses
```

Figure 7. The `getAllParses` function to compute the set of all parses of a data type string s.

```
printFD(v, π = (δ_0, ⟨(f_{s_i}, δ_i)⟩_i)): string
 outString := δ_0
 foreach (f_{s_i}, δ_i) ∈ π :
    if f_i ∈ v :
      outString += Format(fieldVal, f_{s_i}) + δ_i
    else return ϵ
 return outString
```

Figure 8. The `printFD` function for pretty printing a data type instance v in terms of π.

THEOREM 1. *The `getAllParses` function in Figure 7 is complete for regular format descriptors, i.e. it returns the set of all regular format descriptors of a data type string.*

Proof Sketch: The `ComputeMatches` function finds all possible regular expression matches in the string and the `GetValidPaths` function finds all valid format descriptors. These functions do not lose any match/path for regular format descriptors by definition. So, we now need to show that the `ConstructParseGraph` function constructs a parse graph that does not miss any edge belonging to a regular format descriptor. Let us consider three nodes n_1, n_2, and n_3 in the parse graph $\mathcal{G}$ with labels $\mathcal{L}(n_1) = \{i_1,j_1,f_{s_1}\}$, $\mathcal{L}(n_2) = \{i_2,j_2,f_{s_2}\}$, and $\mathcal{L}(n_3) = \{i_3,j_3,f_{s_3}\}$. There are two cases when there is not an edge between nodes n_1 and n_2. The first case is when node n_2 is not the first field match occurring after n_1, i.e. $i_3 \geq j_1 \wedge i_3 < i_2$. There would be no incoming edge to n_2 in the graph if $i_2 < j_3$, but this is fine because having such an edge would result in $s[i_3..i_2]$ to be a delimiter string which is also part of a field match (n_3). The second case when there is no edge from n_1 to n_2 is when the node n_3 overlaps in the interval j_1 to i_2, i.e. $i_3 < j_1 \wedge j_3 > j_1$. This case is also fine because such an edge would result in $s[j_1..j_3]$ to be a delimiter string that is also a part of a field match (n_3). Therefore, all edges that correspond to a regular format descriptor are contained in the parse graph $\mathcal{G}$. □

4.2 Pretty Printing Data Type Instances

The `printFD` function takes a data type instance v and an output format descriptor π as input, and returns the string obtained by pretty printing the instance v in accordance with π as shown in Figure 8. The function first adds the initial delimiter string δ_0 to the output string. It then iterates over the sequence of field-delimiter pairs $(\mathtt{f}_{\mathtt{s}_i},\delta_i) \in \pi$, and computes the field values to be printed in the output string for each pair. If the field $\mathtt{f}_i$ is present in instance v, it first formats the field value in the $\mathtt{f}_{\mathtt{s}_i}$ sub-field representation using the `Format` function, and then appends the formatted field value with the delimiter string δ_i to the output string. For example, the `printFD` function pretty prints a date instance $v \equiv \mathtt{Date}(\mathtt{d} \mapsto 5, \mathtt{m} \mapsto 8, \mathtt{y} \mapsto 2011)$ using $\pi \equiv (\epsilon, \langle((\mathtt{m}_3,3,\mathtt{proper}),\text{" "}),(\mathtt{d}_1,\text{" "}),(\mathtt{y}_2,\epsilon)\rangle)$ to the output string "`Aug 5 2011`".

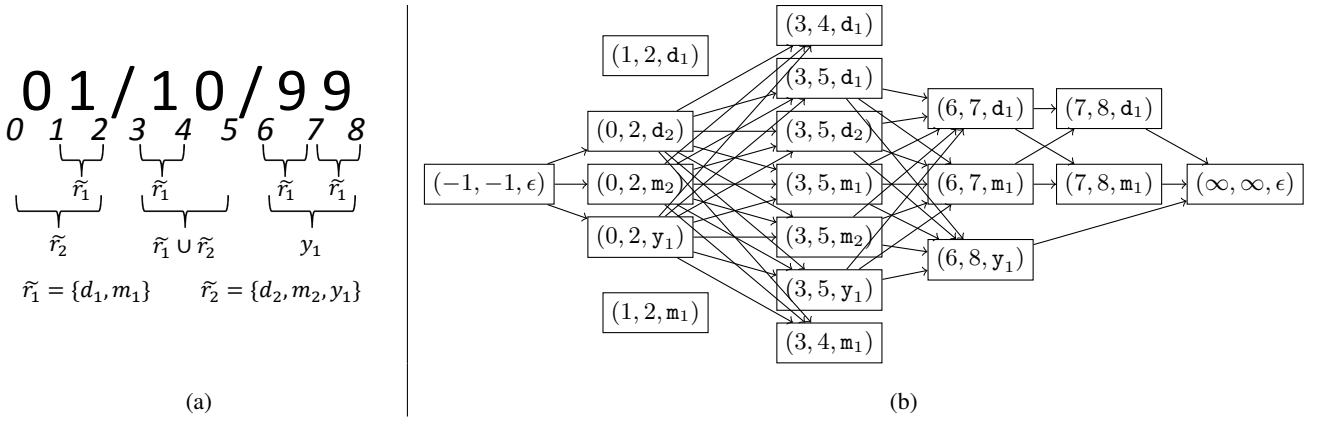


Figure 6. (a) The field matches and (b) the parse graph $\mathcal{G}$ for the date string "`01/10/99`".

5. Data Type Transformation Language

We first present a simple proposal for a baseline language $\mathcal{L}_b$ that can perform data type transformations. Since the data type strings are often present in multiple formats, a program in this language consists of a sequence of `case` statements (one for each format) that are of the form `case` $\pi_{\texttt{in}} \rightarrow \pi_{\texttt{out}}$, where $\pi_{\texttt{in}}$ corresponds to the input format descriptor and $\pi_{\texttt{out}}$ corresponds to the output format descriptor of the input-output examples. The semantics of this program is to compare the format descriptor of an input data type string with $\pi_{\texttt{in}}$, and if the input format descriptors match, then pretty print the data type using the $\pi_{\texttt{out}}$ format descriptor. We can also learn programs in this language with a simplification of the synthesis algorithm that we present in Section 6.

Since the comparison of input format descriptors is Boolean in this language, the comparison will fail for input data type strings that are in a format different than the ones present in the case statements. Furthermore, these data type strings are inherently ambiguous and a single format can itself have multiple interpretations. This non-uniformity and ambiguity would require users to provide an input-output example for each data type format and each interpretation to learn such a program. Another issue with this language is that whenever the output string is ambiguous, programs learnt in this language would produce multiple output strings for new inputs without any semantics of which output string to show to the user.

We extend the Boolean semantics of this proposal to a probabilistic semantics such that a program in the resulting data type transformation language $\mathcal{L}_e$ generates a set of weighted output strings. The language allows for real-valued interpretations of similarity of two format descriptors (`Approximate Matching`), which enables the program to generate output for input data type strings that are in a different format than the ones present in the case statements. To handle ambiguity for the data type strings, we add a dictionary of input format descriptors that helps in disambiguation based on other inputs present in the spreadsheet (`Joint Learning`). Finally, each conditional branch is associated with a weight that is used for ranking the generated output strings. This probabilistic semantics of our language enables the learning of the desired transformations from very few input-output examples.

5.1 Syntax

The syntax of language $\mathcal{L}_e$ is shown in Figure 9. A program $\mathcal{P}$ consists of a tuple $(\mathtt{D}, \Gamma)$, where $\mathtt{D}$ denotes a dictionary that stores the frequency counts of all format descriptors of the input strings present in the spreadsheet and Γ denotes a reformat program that consists of a set of reformat expressions ρ. The dictionary $\mathtt{D}$ is used to disambiguate parses of ambiguous data type strings using other input strings present in the spreadsheet, whereas a reformat program formats an input data type string to a set of weighted output strings. A reformat expression ρ is similar to a case statement of $\mathcal{L}_b$ and is denoted by a tuple $(\pi_{\texttt{in}}, \pi_{\texttt{out}}, w)$, where $\pi_{\texttt{in}}$ is used to parse input data type strings, $\pi_{\texttt{out}}$ is used to pretty print the parsed data type instance and w denotes the likelihood of the reformat expression. A format descriptor π consists of an initial delimiter string and a sequence of pairs of sub-field and delimiters, as described in Section 3.

Program $\mathcal{P}$	:=	$(\mathtt{D}, \Gamma)$
Fmt. Desc. Dictionary $\mathtt{D}$	:=	$\{(\pi_i, \nu_i)\}_i$
Reformat Prog. Γ	:=	$\{\rho_i\}_i$
Reformat Expr. ρ	:=	$(\pi_{\texttt{in}}, \pi_{\texttt{out}}, w)$
Fmt. Desc. π	:=	$(\delta_0, \langle(\mathtt{f}_i, \delta_i)\rangle_i)$

Figure 9. Syntax of data type transformation language $\mathcal{L}_e$.

EXAMPLE 5. *The transformation in Example 1 is given by:* $\mathcal{P}_1 \equiv (\mathtt{D}, \Gamma)$*, where* $\mathtt{D} = \{(\pi_1, \frac{2}{12}), (\pi_2, \frac{2}{12}), (\pi_3, \frac{2}{12}), (\pi_4, \frac{2}{12}), (\pi_5, \frac{1}{12}), (\pi_6, \frac{2}{12}), (\pi_7, \frac{1}{12})\}$*,* $\pi_1 \equiv (\epsilon, \langle(m_2, /), (d_1, /), (y_2, \epsilon)\rangle)$*,* $\pi_2 \equiv (\epsilon, \langle(m_2, /), (d_2, /), (y_2, \epsilon)\rangle)$*,* $\pi_3 \equiv (\epsilon, \langle(d_1, .), (m_2, .), (y_2, \epsilon)\rangle)$*,* $\pi_4 \equiv (\epsilon, \langle(d_2, .), (m_2, .), (y_2, \epsilon)\rangle)$*,* $\pi_5 \equiv (\epsilon, \langle(y_2, -), (m_2, -), (d_2, \epsilon)\rangle)$*,* $\pi_6 \equiv (\epsilon, \langle(y_2, -), (d_2, -), (m_2, \epsilon)\rangle)$*,* $\pi_7 \equiv (\epsilon, \langle(y_2, -), (d_1, -), (m_2, \epsilon)\rangle)$*,* $\Gamma \equiv \{(\pi_1, \pi_1, w_1), (\pi_2, \pi_1, w_2), (\pi_1, \pi_2, w_3), (\pi_2, \pi_2, w_4)\}$*, where weights* w_i *are computed as shown in Section 6.*

The reformat program Γ consists of reformat expressions that are combinations of format descriptors π_1 and π_2. Even though an input string such as $s_3 \equiv$ "`20.08.2010`" is in a different format than the example string "`08/21/2010`", the program still generates the correct output as the string s_3 has a single interpretation and the program assigns a low non-zero weight to the match of the two input format descriptors (using Approximate predicate matching in Section 5.2). For the ambiguous input string $s_5 \equiv$ "`2010-06-07`" that has two interpretations corresponding to the format descriptors π_5 and π_6, the dictionary assigns a higher frequency to π_6 $((\mathtt{D}[\pi_6] = \frac{2}{12}) > (\mathtt{D}[\pi_5] = \frac{1}{12}))$ based on the occurrence of another input string "`2010-24-08`" in the spreadsheet that also has the same format descriptor π_6. In this manner, program $\mathcal{P}_1$ generates the correct output string for all the input strings.

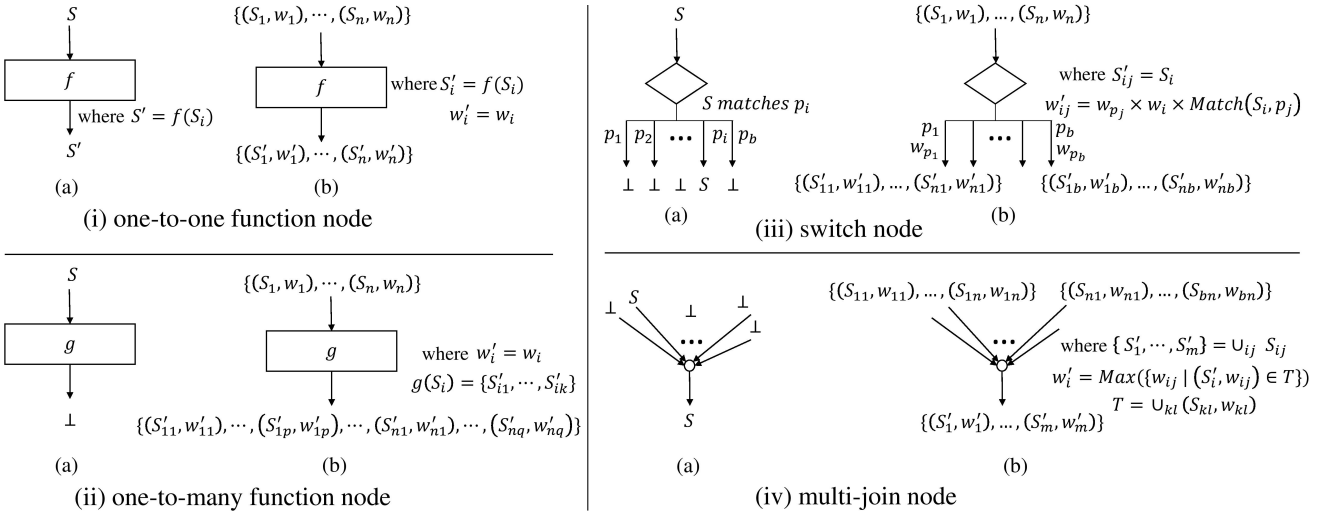


Figure 10. Abstract semantics of the data type transformation language inspired by the random interpretation framework [8, 11]. The four flowchart nodes of our language are shown together with their corresponding (a) deterministic semantics and (b) probabilistic semantics.

5.2 Semantics

The probabilistic semantics of our language is inspired by the work on random interpretation [8, 11]. We first present the abstract semantics of our data type transformation language (for simplifying the description of the key high-level ideas) and then present its concrete semantics. There are four flowchart nodes in our language: (i) one-to-one function node, (ii) one-to-many function node, (iii) switch node, and (iv) multi-join node. The four nodes are shown in Figure 10 together with their deterministic semantics (on the left labelled `(a)`) and probabilistic semantics (on the right labelled `(b)`).

In the deterministic semantics, a one-to-one function node maps an input program state S to an output program state S', whereas a one-to-many function node maps it to bottom $\perp$. In a switch node, one of the branch predicates p_i is satisfied in a state S and that branch produces the output state S, while other branches produce $\perp$. The multi-join node performs the join of a set of bottom values $\perp$ and a state S to map it to a single state S. On the other hand with the probabilistic semantics, the nodes take as input a set of weighted states $\{(S_1, w_1), \cdots, (S_n, w_n)\}$, where the tuple (S_i, w_i) denotes that state S_i is associated with weight w_i. The one-to-one function node maps the input set of weighted states to another set of weighted states $\{(f(S_1), w_1), \cdots, (f(S_n), w_n)\}$ by applying the function f to each state in the input set. The one-to-many function node maps the input set of weighted states to a set of weighted states such that for each tuple (S_i, w_i) in the input set, it produces a weighted set $\{(S'_{i1}, w_{i1}), \cdots, (S'_{ik}, w_{ik})\}$ in the resulting set, where states $\{S'_{i1}, \cdots, S'_{ik}\}$ are obtained by applying the one-to-many function g to state S_i. In a switch node, a branch i with predicate p_i and weight w_{p_i} maps the input set of weighted states to another set of weighted states of same size $\{(S_i, w'_{1i}), \cdots, (S_n, w'_{ni})\}$, where weight w'_{ji} is computed by multiplying the branch weight w_{p_i} with the input state weight w_i and with a value $\texttt{Match}(S_i, p_j)$ that denotes how closely predicate p_j is satisfied in state S_i. The multi-join node takes a set of set of weighted states as input and merges the states to produce a set of weighted states $\{(S'_1, w'_1), \cdots, (S'_m, w'_m)\}$. It first computes the union of states in the input sets to obtain $T = \cup_{kl}(S_{kl}, w_{kl})$ and then selects the maximum weight w'_i for each state S'_i in T. Note that we have chosen the `Max` function here instead of the `Add` function to filter single occurrence of high-confidence inference over multiple occurrences of low-confidence inferences.

The concrete semantics of the language $\mathcal{L}_e$ is shown in Figure 12. A program $\mathcal{P} = (\texttt{D}, \Gamma)$ on an input string s generates a set of weighted output strings $\{(s_{\texttt{out}}, w)\}$. The semantics of a set Γ of reformat expressions is similar to the semantics of a multi-join node. The set of weighted strings is obtained by selecting the maximum weight w for each output string $s_{\texttt{out}}$ from the set of weighted strings obtained from individual reformat expressions $\rho \in \Gamma$. The semantics of a reformat expression $\rho = (\pi_{\texttt{in}}, \pi_{\texttt{out}}, w)$ is similar to the semantics of a switch node. It first parses the input string using the `parseE` function to obtain a set of weighted instances V, and then pretty prints these weighted instances using the `printE` function to obtain a set of weighted output strings. The `parseE` function computes the `Match` value of how closely an input parse descriptor matches with $\pi_{\texttt{in}}$, and the `printE` function multiplies the branch weight w with the weight of the weighted instances obtained from the `parseE` function.

The $\texttt{parseE}(s, \pi, \texttt{D})$ function parses the string s in accordance with the format descriptor π. There are two key aspects of the `parseE` function. The first idea is to use Approximate Predicate Matching to assign a higher weight to those parses (format descriptors) of s that are more similar to the format descriptor π. The second idea is to use Joint Learning for disambiguating ambiguous parses by assigning a higher weight to parses that occur more frequently in the dictionary. The `parseE` function first computes the set of all format descriptors of s and the corresponding data type instances $(\{(v_i, \pi_i)\}_i)$. For each format descriptor π_i in the set, it calculates a similarity score w'_i by matching it with the format descriptor π using the `matchParse` function shown in Figure 11. The `matchParse` function returns a perfect score of 1 if the two format descriptors π_1 and π_2 are equal. Otherwise, if the sub-fields or fields of the two format descriptors are equal (ignoring the delimiter strings), it assigns a higher score to the match than the default score of ξ. The sub-fields and fields of a format descriptor are obtained using the corresponding projection operator Π. For our experiments, we chose the values $\xi = 0.0001$ and $k = 10$ based on empirical experimentation. The `parseE` function finally computes the weight w_i associated with the data type instance v_i by multiplying the closeness weight w'_i with the frequency of the format

```
matchParse(π1, π2) : double
ξ: small constant, k: scaling factor
if(π1 = π2): return 1                    // complete match
if (Π_fs(π1) = Π_fs(π2)):                // sub-fields match
   return ξ + 2 * length(Π_fs(π1)) * ξ / k
if (Π_f(π1) = Π_f(π2)):                  // fields match
   return ξ + length(Π_f(π1)) * ξ / k
return ξ
```

Figure 11. The `matchParse` function to compute the closeness score for two format descriptors π_1 and π_2.

descriptor π_i in the dictionary $\mathtt{D}[\pi_i]$, and returns the resulting set of weighted instances $\{(e_i, w_i)\}_i$.

The $\mathtt{printE}(V, \pi, w)$ function pretty prints a set of weighted data type instances V according to the format descriptor π to compute a set of weighted output strings. It formats a data type instance $(v_i, w_i) \in V$ by using the `formatE` function to obtain the output string s_i and assigns it a weight $w \times w_i$.

The `getAllFD` function, a one-to-many function, uses the `getAllParses` (Section 4) function to compute the set of format descriptors and the corresponding instances for a data type string s. It then uses the `ToCan` function to canonicalize the instances and returns the set of pairs of canonical instances and format descriptors. The `formatE` function, a one-to-one function, takes a canonical data type instance v and a format descriptor π as input, and formats the instance according to the format descriptor π using the `printFD` function (Section 4) after appropriately converting the canonical instance using the `FromCan` function.

$$
\begin{aligned}
[\![(\mathtt{D}, \Gamma)]\!]_s &= [\![\Gamma]\!]_{s,\mathtt{D}} \\
[\![\{\rho_i\}_i]\!]_{s,\mathtt{D}} &= \{(s_{\mathtt{out}}, \mathtt{Max}(\{w_i\}_i)) \mid (s_{\mathtt{out}}, w_i) \in [\![\rho_i]\!]_{s,\mathtt{D}}\} \\
[\![(\pi_{\mathtt{in}}, \pi_{\mathtt{out}}, w)]\!]_{s,\mathtt{D}} &= \mathtt{printE}(\mathtt{parseE}(\sigma(s), \pi_{\mathtt{in}}, \mathtt{D}), \pi_{\mathtt{out}}, w) \\
\mathtt{parseE}(s, \pi, \mathtt{D}) &= \{(v_i, w_i) \mid (v_i, \pi_i) \in \mathtt{getAllFD}(s), \\
&\qquad w_i' = \mathtt{matchParse}(\pi_i, \pi), \\
&\qquad w_i = w_i' \times \mathtt{D}[\pi_i]\} \\
\mathtt{printE}(V, \pi, w) &= \{(s_i, w \times w_i) \mid s_i = \mathtt{formatE}(v_i, \pi) \\
&\qquad (v_i, w_i) \in V\} \\
\mathtt{getAllFD}(s) &= \{(v_{i_c}, \pi_i) \mid v_{i_c} = \mathtt{ToCan}(v_i), \\
&\qquad (v_i, \pi_i) \in \mathtt{getAllParses}(s)\} \\
\mathtt{formatE}(v, \pi) &= \text{let } \mathtt{F} = \mathtt{subFields}(\pi) \text{ in} \\
&\quad \text{let } v' = \mathtt{FromCan}(v, \mathtt{F}) \text{ in} \\
&\quad \mathtt{printFD}(v', \pi)
\end{aligned}
$$

Figure 12. Semantics of transformation language $\mathcal{L}_e$.

The semantics of the program $\mathcal{P}_1$ in Example 5 on the input string s_2 = "`07/24/2010`" is as follows. Consider the first reformat expression $(\pi_1, \pi_1, w_1) \in \Gamma$. The `getAllFD` function on s_2 returns the set $\{(v_1, \pi_1), (v_2, \pi_2)\}$, where $v_1 \equiv v_2 \equiv \mathtt{Date}(\mathtt{d} \mapsto 24, \mathtt{m} \mapsto 7, \mathtt{y} \mapsto 2010)$ with π_1 and π_2 format descriptors as defined in Example 5. The `parseE` function computes the closeness weights w_1' and w_2' of format descriptors π_1 and π_2 respectively with the input format descriptor π_1. We have $w_1' > w_2'$ as π_1 is the same as the input format descriptor of (π_1, π_1). The function then multiplies the closeness weights with respective dictionary frequency of the format descriptors($\frac{2}{12} = \frac{1}{6}$ for both) and returns the set $V \equiv \{(v_1, \frac{w_1'}{6}), (v_2, \frac{w_2'}{6})\}$. The `printE` function then formats the instances in V according to the output format descriptor π_1 and produces the weighted set of output strings $\{(\mathtt{07/24/2010}, \frac{w_1 \times w_1'}{6}), (\mathtt{07/24/2010}, \frac{w_1 \times w_2'}{6})\}$. On the input string s_5 = "`2010-06-07`", the `getAllFD` function returns $\{(v_1, \pi_5), (v_2, \pi_6)\}$, where $v_1 \equiv \mathtt{Date}(\mathtt{d} \mapsto 7, \mathtt{m} \mapsto 6\mathtt{y} \mapsto 2010)$ and $v_2 \equiv \mathtt{Date}(\mathtt{d} \mapsto 6, \mathtt{m} \mapsto 7\mathtt{y} \mapsto 2010)$. Both format descriptors π_5 and π_6 do not match with the input parse descriptors of reformat expressions (π_1 and π_2) in Γ, and therefore get a low closeness weight $w_1' = w_2' = w'$ by the `matchParse` function. The dictionary frequency of format descriptor π_6 ($\frac{2}{12}$) is higher than that of π_5 ($\frac{1}{12}$), and therefore the `parseE` function assigns a higher weight to the date instance v_2 and returns $\{(v_1, \frac{w'}{12}), (v_2, \frac{w'}{6})\}$. The `printE` function then returns the set: $\{(\mathtt{07/6/2010}, \frac{w_1 \times w'}{6}), (\mathtt{06/7/2010}, \frac{w_1 \times w'}{12})\}$.

Conditional Formatting Programs in $\mathcal{L}_e$ The input format descriptors $\pi_{\mathtt{in}}$ in the reformat expressions implicitly encode the conditionals for transformations that require *conditional formatting*. A reformat program $\Gamma = \{(\pi_1, \pi_2, w_1), (\pi_3, \pi_4, w_2)\}$ encodes a conditional program such that the input strings whose format descriptor match closely with π_1 are formatted using the format descriptor π_2 whereas inputs whose format descriptors match closely with π_3 are formatted according to π_4.

EXAMPLE 6. *A company executive had log entries from two offices in different locations (India and US). She wanted to create filenames for each log entry such that the filename consisted of the office location followed by the date in a consistent format as shown in Figure 13.*

	Input	Output
1	09/08/2011	us_8sep11.log
2	05.03.2010	in_5mar10.log
3	11/23/2011	**us_23nov11.log**
4	09.08.2010	**in_9aug10.log**
5	14.09.2010	**in_14sep10.log**
6	06/15/2011	**us_15jun11.log**

Figure 13. Generating log filenames based on different date format (us or indian).

The desired program $\mathcal{P}_2$ for generating the log filenames can be represented in $\mathcal{L}_e$ as $\mathcal{P}_2 \equiv (\mathtt{D}, \Gamma)$, where
$\mathtt{D} \equiv \{(\pi_1, \frac{3}{12}), (\pi_2, \frac{1}{12}), (\pi_3, \frac{2}{12}), (\pi_4, \frac{3}{12}), (\pi_5, \frac{1}{12}), (\pi_6, \frac{2}{12})\}$,
$\pi_1 \equiv \{\epsilon, \langle(\mathtt{m}_2, /), (\mathtt{d}_2, /), (\mathtt{y}_2, \epsilon)\rangle\}, \pi_2 \equiv \{\epsilon, \langle(\mathtt{d}_2, /), (\mathtt{m}_2, /), (\mathtt{y}_2, \epsilon)\rangle\}$,
$\pi_3 \equiv \{\epsilon, \langle(\mathtt{m}_2, /), (\mathtt{d}_1, /), (\mathtt{y}_2, \epsilon)\rangle\}, \pi_4 \equiv \{\epsilon, \langle(\mathtt{d}_2, .), (\mathtt{m}_2, .), (\mathtt{y}_2, \epsilon)\rangle\}$,
$\pi_5 \equiv \{\epsilon, \langle(\mathtt{m}_2, .), (\mathtt{d}_2, .), (\mathtt{y}_2, \epsilon)\rangle\}, \pi_6 \equiv \{\epsilon, \langle(\mathtt{d}_1, .), (\mathtt{m}_2, .), (\mathtt{y}_2, \epsilon)\rangle\}$,
$\pi_7 \equiv \{\mathtt{us\_}, \langle(\mathtt{d}_1, \epsilon), ((\mathtt{m}_3, 3, \mathtt{iden}), \epsilon)(\mathtt{y}_1, \mathtt{.log})\rangle\}, \pi_8 \equiv \{\mathtt{in\_}, \langle(\mathtt{d}_1, \epsilon),$
$((\mathtt{m}_3, 3, \mathtt{iden}), \epsilon), (\mathtt{y}_1, \mathtt{.log})\rangle\}, \Gamma \equiv \{(\pi_1, \pi_7, w_1), (\pi_4, \pi_8, w_2)\}$.

The US date strings match the input format descriptor π_1 of Γ and get formatted using the corresponding output format descriptor π_7, whereas the Indian date strings are matched with the format descriptor π_4 and get formatted using the format descriptor π_8.

6. Synthesis Algorithm

We now present the synthesis algorithm to learn a program in $\mathcal{L}_e$ that performs the desired transformation from a given set of input-output examples. The main challenge of the algorithm is to learn a dictionary of weighted format descriptors and the set of reformat expressions. The algorithm computes the normalized frequency counts for format descriptors of the input strings to compute the dictionary. For learning the reformat expressions for an input-output example, the key idea of the algorithm is to first enumerate pairs of format descriptors of the input and output data type strings, and then compute consistent matching pairs (while

accounting for missing fields). The algorithm then takes the union of the reformat expressions from multiple input-output examples to assign higher weights to the expressions that occur more frequently.

`GenProgram` **Algorithm:** The `GenProgram` algorithm, shown in Figure 14(a), takes a set of input-output examples $\mathcal{T}$ and the spreadsheet data as inputs, and learns a program $\mathcal{P}$ that can format the input data type strings to the corresponding output strings in the set $\mathcal{T}$. The algorithm first computes the format descriptor dictionary `D` using the `GenDict` function. It then synthesizes a set of reformat expressions for each input-output example $(s_i, s_o) \in \mathcal{T}$ using the `GenRefmtExprs` function. After learning the reformat expressions, it performs their set union and adds the weights for common reformat expressions to compute the set of reformat expressions Γ for a set of examples. The weights for the reformat expressions are then normalized and bounded such that they are always between 1 and e. This approach of using union to learn programs conforming to multiple examples is essential for learning robust conditional programs over non-uniform, ambiguous, and unstructured datasets.

`GenDict` **Algorithm:** The `GenDict` function, shown in Figure 14(b) takes as input a set of spreadsheet input strings ($\mathcal{I}$) and computes a set of weighted format descriptors. The function first computes the set of format descriptors for each input string using the `getAllFD` function, and adds each format descriptor (π) to the dictionary `D` maintaining their normalized frequency counts (`D`$[\pi]$). The complexity of the `GenDict` function is $O(mk^2|s|^4)$, where m is the number of input strings in the spreadsheet. For spreadsheets with large number of input rows, sampling can be performed to efficiently learn the dictionary over a smaller set of input strings.

`GenRefmtExprs` **Algorithm:** The `GenRefmtExprs` function, shown in Figure 14(c), learns the set of reformat expressions Γ that can format the input data type string s_i to the output data type string s_o. It first computes the set of format descriptors (together with the corresponding canonical entity instances) P_i and P_o for the strings s_i and s_o respectively using the `getAllFD` function (Lines 1-2). It then iterates over the pairs of their combinations $\{(v_k, \pi_k) \in P_i \times (v_l, \pi_l) \in P_o\}$ to compute consistent pairs for the reformat expressions. Given a pair of format descriptors, the algorithm iterates over the sequence of field-delimiter pairs ($\mathtt{f_s}, \delta$) of the output format descriptor and matches the corresponding field values of the data type instances (Lines 7-17). If all the field values match, the reformat expression (π_k, π_l, w) is added to the set Γ with the weight $w = \mathtt{D}[\pi_k] \times w(\pi_k) \times w(\pi_l)$. If any field value is not consistent between the two instances v_k and v_l, the format descriptor pair (π_k, π_l) is discarded. Finally, there is an interesting case of a field present in output instance v_l that is missing from the input instance v_k. For this case, the algorithm treats the field value in the output string as a delimiter string and adds it to the last delimiter string in γ (Lines 16-17). It then resumes the iterative field value matching process. The complexity of `GenRefmtExprs` function is $O(k^3|s|^4)$.

EXAMPLE 7. *Consider the date formatting in Example 1 and its corresponding $\mathcal{L}_e$ program $\mathcal{P}_1$ in Example 5.*

The synthesis algorithm first constructs the dictionary `D` consisting of weighted format descriptors of all spreadsheet inputs using the `GenDict` function. The `getAllFD` function on the input string "`08/21/2010`" returns a set of two format descriptors $\{(v_1, \pi_1), (v_2, \pi_2)\}$, where $v_1 \equiv v_2 \equiv$ `Date(d` $\mapsto 21, $`m` $\mapsto 8, $`y` $\mapsto 2010, $`dow` $\mapsto$ `Saturday`). Similarly, the set of format descriptors for other input strings : `07/24/2010` $\rightarrow \{(v_3, \pi_1), (v_4, \pi_2)\}$, `20.08.2010` $\rightarrow \{(v_5, \pi_3), (v_6, \pi_4)\}$, `23.08.2010` $\rightarrow \{(v_7, \pi_3), (v_8, \pi_4)\}$, `2010-06-07` $\rightarrow \{(v_9, \pi_5), (v_{10}, \pi_6)\}$, `2010-24-08` $\rightarrow \{(v_{11}, \pi_6), (v_{12}, \pi_7)\}$. The 7 format descriptors occur in total 12 times. The dictionary `D` is:$\{(\pi_1, \frac{2}{12}), (\pi_2, \frac{2}{12}), (\pi_3, \frac{2}{12}), (\pi_4, \frac{2}{12}), (\pi_5, \frac{1}{12}), (\pi_6, \frac{2}{12}), (\pi_7, \frac{1}{12})\}$.

The algorithm then uses the `GenRefmtExprs` function to compute the set of reformat expressions Γ to format ($s_i \equiv$ `08/21/2010`) to ($s_o \equiv$ `08/21/2010`). The function first computes the set of format descriptors: $P_i \equiv \{(v_1, \pi_1), (v_2, \pi_2)\}$ and $P_o \equiv \{(v_{13}, \pi_1), (v_{14}, \pi_2)\}$, where $v_1 \equiv v_2 \equiv v_{13} \equiv v_{14} \equiv$ `Date(d` $\mapsto 21, $`m` $\mapsto 8, $`y` $\mapsto 2010, $`dow` $\mapsto$ `Saturday`). It then iterates over four possible combinations of the input and output format descriptors: $\{[(v_1, \pi_1), (v_{13}, \pi_1)], [(v_1, \pi_1), (v_{14}, \pi_2)], [(v_2, \pi_2), (v_{13}, \pi_1)], [(v_2, \pi_2), (v_{14}, \pi_2)]\}$. The function `MatchF` matches the corresponding field values of date instances by performing equality check on the field values. As all date instances are equal in this case, the function returns the reformat expression set Γ consisting of all 4 combinations of format descriptors $\{(\pi_1, \pi_1, w_1), (\pi_2, \pi_1, w_2), (\pi_1, \pi_2, w_3), (\pi_2, \pi_2, w_4)\}$. The algorithm then returns the program $(\mathtt{D}, \Gamma)$.

DEFINITION 7 (Inconsistent Data). *We define a set of input strings $\{i_1, \cdots i_n\}$ to be inconsistent if there exists two different inputs i_k and i_l with format descriptors π_{i_k} and π_{i_l} respectively $(\pi_{i_k} \neq \pi_{i_l})$ such that there is no common format descriptor between i_k and i_l but the sequence of delimiter strings of the two format descriptors π_{i_k} and π_{i_l} is equal.*

For example, the set of input strings $\{$`12/23/99`, `15/10/94`$\}$ is inconsistent since there exists two different format descriptors corresponding to the formats `mm/dd/yy` and `dd/mm/yy` for the two inputs respectively that have same set of delimiter strings $\{\epsilon, /, /\}$.

DEFINITION 8 (Noisy Examples). *We define a set of input-output examples $\{(i_1, o_1), \cdots, (i_n, o_n)\}$ to be noisy if there exists two examples (i_k, o_k) and (i_l, o_l) whose input format descriptors are equal ($(\pi_{i_k} = \pi_{i_l})$) but there does not exist any format descriptor that is common to both the output strings o_k and o_l.*

For example, the set of examples $\{$(`12/23/99`, `23 Dec`), (`11/19/94`, `19 November`)$\}$ is noisy as there are two different output format descriptors corresponding to the output strings `23 Dec` and `19 November` for the same input format descriptor. Similarly, the set of input-output examples $\{$(`24/7/99`, `24 July`), (`14/2/94`, `14 Feburary`)$\}$ is noisy as there is a spelling mistake in the output string for the second example `Feburary`, which leads to two different output format descriptors.

THEOREM 2 (Soundness). *The `GenProgram` algorithm is sound for a set of non-noisy input-output examples and consistent input data, i.e. the program synthesized by the algorithm is guaranteed to generate the provided output strings with highest weight when run on the corresponding input strings for consistent examples.*

Proof Sketch: Consider an input-output example (i, o) and let there be n different reformat expressions $\{\rho_1, \cdots, \rho_n\} \in \mathcal{P}$ that the `GenProgram` algorithm learns from this example. The claim is that a reformat expression $\rho = (\pi_{\mathtt{in}}, \pi_{\mathtt{out}}, w) \notin \{\rho_1, \cdots, \rho_n\}$ of program $\mathcal{P}$ whose input format descriptor $\pi_{\mathtt{in}}$ matches a format descriptor of the input string i cannot exist. If such a reformat expression existed, that would mean the provided examples are defining two different formatting transformations on the same input format, which cannot happen with a set of consistent examples. Now, when we execute the learnt program $\mathcal{P}$ on input string i, only input format descriptors of reformat expressions $\{\rho_1, \cdots, \rho_n\}$ would match with a high weight. Since each output format descriptor of the reformat expressions in this set generates the output string o (line 5 in Figure 14), the learnt program $\mathcal{P}$ always generates o as the highest weight output string when executed on the input string i. □

```
GenProgram(T ≡ {(s_{i_k}, s_{o_k})}_k, I) : P
D := GenDict(I), Γ_D := ⟨⟩, Γ := ∅
foreach (s_i, s_o) ∈ T:
    Γ_1 := GenRefmtExprs(s_i, s_o)
    foreach ρ ≡ (π_in, π_out, w_1) ∈ Γ_1:
        if Γ_D.Contains(ρ):
            Γ_D[ρ] := Γ_D[ρ] + w_1
        else:  Γ_D.Add(ρ, w_1)
foreach ρ ≡ (π_in, π_out, w_1) ∈ Γ_D.Keys():
    Γ := Γ ∪ {(π_in, π_out, e^{Γ_D[ρ] / Σ_ρ Γ_D[ρ]})}
return (D, Γ)
```

(a)

```
GenDict(I) : D
D = {}, nparses := 0
foreach string s ∈ I:
    foreach (v, π) ∈ getAllFD(s):
        if  D.Contains(π): D[π]++
        else: D.Add(π, 1)
        nparses++
if nparses > 0:
  foreach π ∈ D.Keys(): D[π] := D[π] / nparses
return D
```

(b)

```
GenRefmtExprs(s_i, s_o) : Γ
1  P_i := getAllFD(s_i)
2  P_o := getAllFD(s_o)
3  Γ := ∅
4  foreach (v_k, π_k) ∈ P_i:
5      foreach (v_l, π_l) ∈ P_o:
6          valid_π := true
7          foreach (f_s, δ) ∈ π_l:
8              γ := ⟨⟩
9              if f ∈ Fields(v_k):
10                 f'_s := EqualF(v_k[f], v_l[f], f_s)
11                 if(f'_s ≠ null):
12                     γ := γ + (f'_s, δ)
13                 else:
14                     valid_π := false
15             else:       // missing field in π_k
16                 δ' := Format(v_l[f], f_s) + δ
17                 lastDelString(γ).append(δ')
18         if(valid_π):
19             π_n := (π_l.δ_0, γ)
20             Γ := Γ ∪ (π_k, π_n, D[π_k] × w(π_k) × w(π_n))
21 return Γ
```

(c)

Figure 14. The GenProgram function takes a set $\mathcal{T}$ consisting of a set of input-ouput examples and the set of spreadsheet inputs $\mathcal{I}$ as inputs, and computes a probabilistic program $\mathcal{P}$ that can format the entity strings in the training set.

THEOREM 3 (Completeness). *The GenProgram algorithm is complete for reformat programs with regular format descriptors, i.e. if there exists a reformat program (π_{in}, π_{out}) consisting of regular format descriptors π_{in} and π_{out}, the GenProgram algorithm is guaranteed to synthesize it.*

Proof Sketch: The getAllFD function canonicalizes the instances obtained from the getAllParses function and using the result of Theorem 1, we have that the getAllFD function is complete for regular format descriptors. Since the GenProgram algorithm considers all possible combinations of parse descriptors obtained from the getAllFD function on input and output strings (lines 4-5 in Figure 14), we have that the GenProgram function is complete for reformat programs with regular format descriptors. □

7. Robust Transformations

Assigning a probabilistic semantics to $\mathcal{L}_e$ provides an additional capability of handling examples with missing information as well as noisy and inconsistent input-output examples. Traditional program synthesis approaches fail to learn a program in case of inconsistent specifications as they aim to synthesize programs that are guaranteed to satisfy the given specification. Our approach leverages the weights present in the learnt programs to assign a low likelihood to programs that correspond to inconsistent and noisy examples, so that these examples do not affect transformations on other inputs present in the spreadsheet. We give a few examples that show how our algorithm handles such cases.

EXAMPLE 8 (Missing Information). *An Excel user wanted to format dates present in two different formats `d.m` and `m-d-yyyy` into a uniform format as shown below. The user posted the following example spreadsheet on a help-forum stating that she was struggling in telling Excel to add the year `2010` to dates in which the `year` field was missing.*

	Input	Output
1	24.9	24 Sep 2010
2	6-21-2010	21 Jun 2010
3	29.1	**29 Jan 2010**
4	8-15-2010	**15 Aug 2010**
5	4-18-2010	**18 Apr 2010**
6	16.8	**16 Aug 2010**

Figure 15. Adding missing year value to a date.

The key idea in handling missing fields in input strings is to treat the missing fields as constant delimiter strings. The synthesis algorithm when synthesizing the reformat program for the input-output example ("`24.9`", "`24 Sep 2010`") finds that the `year` field of the output string (`2010`) is not present in the input string (Lines 16-17 in Figure 14) and adds the year field value to previous delimiter string to get the output format descriptor $\gamma \equiv (\epsilon, \langle(\mathtt{d}_1, \texttt{" "}), ((\mathtt{m}_3, 3, \texttt{proper}), \texttt{" 2010"})\rangle)$. It is interesting to note that even though the format descriptor γ is not regular, the synthesis algorithm still learns it. The synthesized program $\mathcal{P}_3$ is:
$\mathcal{P}_3 \equiv (\mathtt{D}, \Gamma)$, where$\mathtt{D} \equiv \{(\pi_1, \frac{3}{6}), (\pi_2, \frac{3}{6}), (\pi_3, \frac{3}{6}), (\pi_4, \frac{3}{6})\}$,
$\pi_1 \equiv (\epsilon, \langle(\mathtt{d}_1, .), (\mathtt{m}_1, \epsilon)\rangle)$, $\pi_2 \equiv (\epsilon, \langle(\mathtt{d}_2, .), (\mathtt{m}_1, \epsilon)\rangle)$,
$\pi_3 \equiv (\epsilon, \langle(\mathtt{m}_1, \texttt{-}), (\mathtt{d}_1, \texttt{-}), (\mathtt{y}_2, \epsilon)\rangle)$, $\pi_4 \equiv (\epsilon, \langle(\mathtt{m}_1, \texttt{-}), (\mathtt{d}_2, \texttt{-}), (\mathtt{y}_2, \epsilon)\rangle)$,
$\Gamma \equiv \{(\pi_1, \pi_5, w_1), (\pi_2, \pi_5, w_2), (\pi_1, \pi_6, w_3), (\pi_2, \pi_6, w_4)\}$.

EXAMPLE 9 (Noisy/Inconsistent Examples). *An excel user wanted to transform dates from `d/m/yyyy` format to `mmmm d` format as shown in Figure 16. The input-output examples in rows 1 and 4 represent the correct formatting operation. The output string in row 2 has a spelling mistake "`Feburary`" (noisy example), whereas the output string in row 3 "`21 September`" has a different format (inconsistent example).*

	Input	Output
1	24/7/2010	July 24
2	14/2/2011	Feburary 14
3	21/9/2010	21 September
4	16/4/2011	April 16
5	11/12/2010	**December 11**
6	19/6/2011	**June 19**

Figure 16. Formatting dates in a uniform format in presence of inconsistent and noisy input-output examples.

The program $\mathcal{P}_4$ learned by `GenProgram` from the 4 examples is given by: $\mathcal{P}_4 \equiv (\mathtt{D}, \Gamma)$, where $\mathtt{D} \equiv \{(\pi_1, \frac{6}{12}), (\pi_2, \frac{6}{12})\}$, $\pi_1 \equiv (\epsilon, \langle(\mathtt{d}_1, /), (\mathtt{m}_1, /), (\mathtt{y}_2, \epsilon)\rangle)$, $\pi_2 \equiv (\epsilon, \langle(\mathtt{d}_2, /), (\mathtt{m}_1, /), (\mathtt{y}_2, \epsilon)\rangle)$, $\pi_3 \equiv (\epsilon, \langle((\mathtt{m}_3, \infty, \mathtt{iden}), \texttt{" "}), (\mathtt{d}_1, \epsilon)\rangle)$, $\pi_5 \equiv (\mathtt{Feburary}, \langle((\mathtt{d}_1, \epsilon)\rangle)$, $\pi_4 \equiv (\epsilon, \langle((\mathtt{m}_3, \infty, \mathtt{iden}), \texttt{" "}), (\mathtt{d}_2, \epsilon)\rangle)$, $\pi_6 \equiv (\mathtt{Feburary}, \langle((\mathtt{d}_1, \epsilon)\rangle)$, $\pi_7 \equiv (\epsilon, \langle(\mathtt{d}_1, \epsilon), ((\mathtt{m}_3, \infty, \mathtt{iden}), \texttt{" "})\rangle)$, $\pi_8 \equiv (\epsilon, \langle(\mathtt{d}_2, \epsilon), ((\mathtt{m}_3, \infty, \mathtt{iden}), \texttt{" "})\rangle)$, $\Gamma \equiv \{(\pi_1, \pi_3, w_2), (\pi_1, \pi_4, w_2), (\pi_1, \pi_5, w_1), (\pi_1, \pi_6, w_1), (\pi_1, \pi_7, w_1), (\pi_1, \pi_8, w_1), (\pi_2, \pi_3, w_2), (\pi_2, \pi_4, w_2), (\pi_2, \pi_5, w_1), (\pi_2, \pi_6, w_1), (\pi_2, \pi_7, w_1), (\pi_2, \pi_8, w_1)\}$, assuming the weights for each reformat expression is equal for simplicity.

The correct reformat expressions with output format descriptors π_3 and π_4 have a higher weight (w_2) than the incorrect reformat expressions with output format descriptors π_5, π_6, π_7 and π_8 (w_1), as the algorithm adds up the weights of reformat expressions learned from examples in rows 1 and 4. The key hypothesis of our algorithm in cases like these is that the inconsistent and noisy examples are less frequent than the number of correct examples in the set of provided examples. Our interface runs the synthesized program on the input-output examples and compares the resulting highest weight output string with the user-provided output string in the example. If they differ, as is the case in this example with the output generated by $\mathcal{P}_4$ for rows 2 and 3 being "`February 14`" and "`September 21`" respectively, the interface highlights such cells for further user inspection and presents the highest weight output string as a suggested fix.

8. Prototype and Experiments

We have implemented our domain specific language and the synthesis algorithm in $C\#$ as an add-in for the Microsoft Excel spreadsheet system. Our user interface is similar to that of FlashFill [9, 10], where users provide input-output examples as a set of rows in an Excel table and our system then generates the output strings for the remaining input strings in the table. In this section, we present the evaluation of our algorithm on a representative set of 55 real-world benchmark problems. Our system can learn the desired transformation for all 55 benchmarks as compared to 26 benchmarks learnt by the base language and 11 benchmarks learnt by FlashFill (and its variants). Moreover, the number of examples needed to learn the desired transformation is also significantly reduced from 2.2 (for the base language) to 1.575.

8.1 Benchmarks

We obtained 55 benchmark problems from both the online help forums (24/55) and the Excel product team (31/55). The benchmarks that the Excel team provided came from internal customer surveys and piloting, while the online forum benchmarks were collected by the authors. Out of the 55 benchmark problems, 40 of them involved transformations on the date data type, whereas the remaining 15 problems involved transformations over other data types including name, phone number, time, and unit. The larger fraction of date scenarios is simply a manifestation of date being a more commonly used data type than others, and one that requires normalization into a clean format for various operational purposes.

The number of input strings in the spreadsheets varied from 6 to 35, with an exception of 910 inputs (randomly generated using a uniform distribution) for Example 2. For the benchmarks taken from the Excel forums, we used the inputs provided by users on the forum posts (either as Excel attachments or from the text of the forum conversations). For the benchmarks provided by the Excel team, we directly used the inputs in the spreadsheets.

8.2 Need for Probabilistic Semantics

We first evaluate the need for probabilistic interpretations in the data transformation language $\mathcal{L}_e$ over 40 benchmark problems from the date domain. The number of benchmark problems for which the desired transformation can be learnt by different tools is shown in Figure 17(a). `FF`, `FF-num`, and `FF-table` denote FlashFill [9], FlashFill extended with numbers [26] and FlashFill extended with tables [25] respectively. Previous FlashFill versions can learn the transformations for only 11 out of the 40 benchmarks and require more number of input-output examples. The base language proposal of data transformation language without probabilistic semantics ($\mathcal{L}_b$) can learn the desired transformations for 26 benchmarks whereas our system learns the desired transformations (in $\mathcal{L}_e$) for all 40 benchmarks.

Ranking The probabilistic interpretations in $\mathcal{L}_e$ is also important for learning the transformations from a small number of input-output examples. The number of examples required to learn the desired transformation for the benchmark problems in different configurations of $\mathcal{L}_e$ (where each feature is incrementally added) is shown in Figure 17(b). Adding weights to reformat expressions in $\mathcal{L}_b$ (`Weighted branches`) reduces the number of examples required to learn the transformation for 5 benchmark problems (on which the base language can learn the transformation) and on average requires 2.2 input-output examples per benchmark problem to learn the desired transformations. Adding the `MatchParse` function to perform real-valued input parse matches (`Approximate Predicate Matching`) reduces the number of examples required for 7 more benchmark problems to 1 from more than 3 previously. The average number of examples required per benchmark problem in this configuration is 1.825. Finally, adding the parse dictionary for disambiguation (`Joint Learning`) further reduces the number of examples required for 8 more problems and the average number of examples required goes down to 1.575.

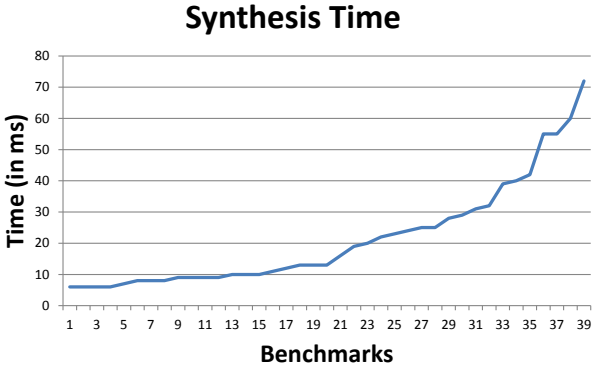
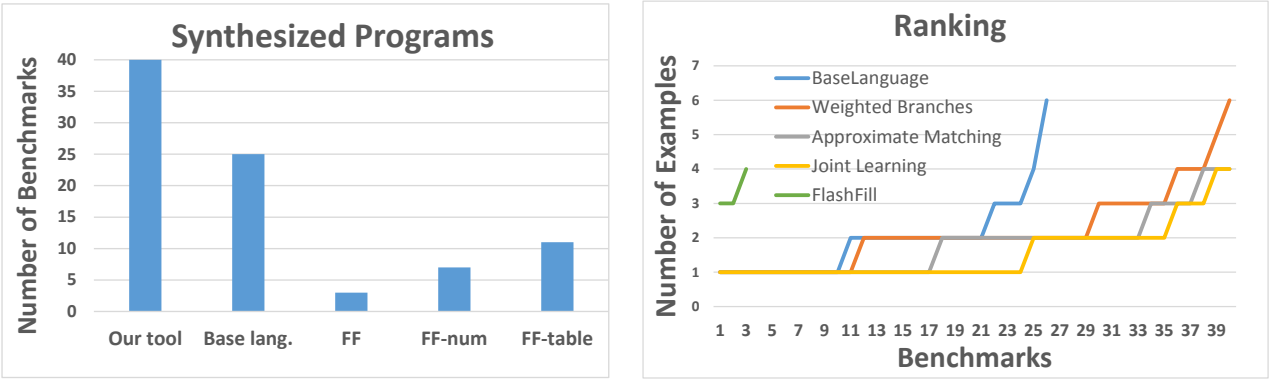


Figure 18. The synthesis time for learning the desired program for each benchmark task.



(a) (b)

Figure 17. (a) The number of benchmarks for which the desired transformation can be learnt by different tools, (b) the number of input-output examples required by different configurations of $\mathcal{L}_e$ to synthesize the desired transformation.

Performance The synthesis algorithm works in real time and takes less than 0.08 seconds each to learn the desired transformation for the 39 benchmark problems as shown in Figure 18. The algorithm takes 1.1 seconds for the benchmark in Example 2 with 910 input strings. The experiments were performed on a machine with Intel Core i5-3317U 1.7GHz CPU with 16 GB of RAM.

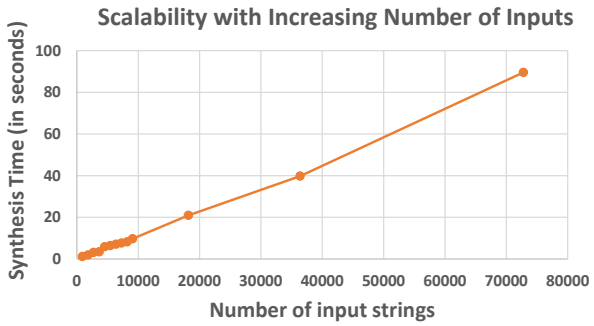


Figure 19. The synthesis times for learning the desired program for varying number of input strings in the spreadsheet.

8.3 Scalability with Increasing Number of Inputs

Since the synthesis algorithm computes the dictionary of weighted parses of the input strings, the scalability of the algorithm also depends on the number of the input strings in the spreadsheet. The learning times for the algorithm on a spreadsheet from Example 2 with increasing number of input strings is shown in Figure 19. The algorithm takes about 3 seconds for 2500 inputs and about 20 seconds for 18000 inputs. However, in practice, we can efficiently learn a good approximation of the dictionary by sampling only a few hundred inputs from large spreadsheets, which takes less than 0.5 seconds.

8.4 Other Data Types

We next present the extensibility of our data description framework by encoding the name, phone number, time, and unit data types. Since our framework is parameterized by declarative data type definitions, we were able to easily add support for these data types within few hours and in less than 200 LOC for each data type. We evaluate these data types over 15 representative benchmark problems. Our system can synthesize transformations for all 15 benchmarks, whereas previous FlashFill versions can learn the transformations for only 5 of them. The number of input-output examples required for each benchmark problem is shown in Figure 20. The time required to learn the desired transformation for each benchmark problem was at most 0.6 seconds.

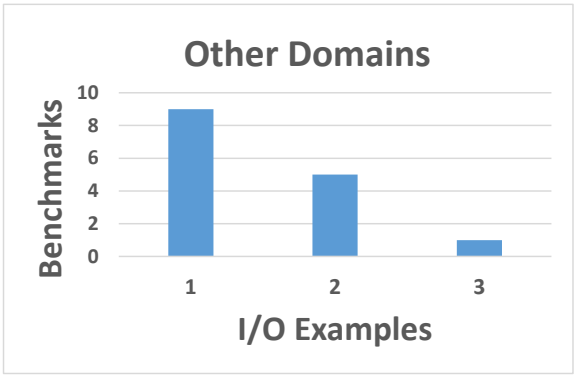


Figure 20. The number of examples required to learn the transformation for the benchmark problems over names, phone number, time, and unit data types.

9. Related Work

The most closely related work to ours is that of automated text-editing using demonstrations and examples. Most of these work treat strings as a sequence of characters without assigning any semantic interpretation to them, and therefore are limited in terms of the range of transformations and the complexity of data they can support. The work on ad-hoc data manipulation for programmers is also closely related but our work also learns the structure of output data and the relationship between the input and output structures in addition to learning the structure of input data.

Text-editing Systems using Demonstrations and Examples: Nix described a text-editing system that synthesizes *gap programs* based on examples [17]. A gap program is a collection of (pattern, replacement) pairs, where each pattern is composed of constants and variables that bind to the text in between the constants, and a replacement can be a constant string or a variable from the input pattern. SMARTedit [15] is a Programming by Demonstration (PBD) system for learning text-editing commands, where the primitive program statements include moving the cursor to a new

position and inserting/deleting text. Simultaneous editing [16] is another PBD-like system that allows the user to define a set of regions to edit and to make edits in one region, while the system makes equivalent editing in all other regions. Potter's wheel [21] lets users define different domains by providing API support and learns the transformations on strings by recognizing the structure of input and output strings using a description length metric.

These systems are more general and support arbitrary input strings but here we limit our discussion to strings that represent data types. These systems interpret strings as only a sequence of characters, and support operations such as substring and their concatenation with constant strings. This limits them to support only simple transformations on the data types. Moreover, these systems support only uniform and structured data types whereas most of the benchmark problems that we obtained from the help-forums included non-uniform and unstructured data.

FlashFill [9] is a programming-by-example system for automating string processing in spreadsheets. It synthesizes *deterministic* programs using a restricted form of regular expressions, conditionals, and loops for performing *syntactic* manipulation of strings. In contrast, we describe a probabilistic system to perform transformations on strings that represent rich data types. FlashFill, like previous systems, can perform only syntactic transformations on data types and lacks information about the data type interpretations to perform more complex transformations. It also supports transformations on non-uniform structured data types by learning a set of conditionals, but it requires a lot of input-output examples to learn such conditional programs and in some cases does not learn any conditional because of a limited language of conditionals. FlashFill's extension for handling relational tables [25] and number transformations [26] enables it to perform more complex transformations but these systems are still limited by interpreting the input data type strings as just strings. Moreover, these extensions can only handle uniform and structured data types.

Topes [23, 24] provides end-users an abstraction of their data and helps them describe constraints on how to validate the data. From the given data, it infers some basic formats such as numbers and words, and allows users to modify the format by adding more constraints or specify additional formats. With these rules, Topes can validate user's data and provide error messages regarding why validation of certain strings failed. In addition, since it has knowledge of the formats, it also provides a finite set of formats as a recommendation in which a user might want to reformat the data. This can be viewed as an abstraction for providing Excel custom format strings, but is more powerful as additional constraints can be specified. Our tool is a complete PBE system and allows users to format the data type strings in any arbitrary format without asking them to specify these formats. Topes also does not handle ambiguity present in the data type strings, which we found essential for learning desired transformations from few input-output examples.

Ad-hoc Data Manipulation for Programmers: The PADS project has enabled simplification of ad hoc data processing tasks for programmers by contributing along several dimensions: development of domain specific languages for describing text structure or data format [3, 4], learning algorithms for automatically inferring such formats [5], and a markup language to allow users to add simple annotations to enable more effective learning of text structure [34]. The learned format can then be used by programmers for documentation or implementation of custom data analysis tools. In contrast, the focus of this paper is to enable end-users (non-programmers) to perform small, often one-off, repetitive tasks on their spreadsheet data. Asking end-users to provide annotations for learning (relatively simple) text structure, and then develop custom tools to format/process the inferred structure is beyond the expertise and usability bar for these users. Hence, we are interested in automating the entire end-to-end process, which includes not only learning the text structure from the inputs, but also learning the desired transformation from the outputs.

Random Interpretation: The probabilistic semantics of our data type transformation language is inspired by the random interpretation framework [8, 11]. The key idea of random interpretation is to execute the program on a few random inputs while modifying states on the fly to satisfy branch constraints, and combining states using random weights at joins. It has been used previously for finding affine equalities [11], global value numbering, and interprocedural analysis of programs [8]. However, our probabilistic semantics maintain weighted states instead of states at each edge. We also execute all branches, but copy the states and adjust the weights based on approximate predicate matching. At join points, we also perform a weighted combination, but we use max instead of addition. Furthermore, we also deal with one-to-many probabilistic operators in our formalism.

Program Synthesis: The area of program synthesis is gaining a renewed interest [1]. It has recently been used to synthesize efficient low-level code using partial programs [29, 30] and from higher-order declarative specifications [14], automated feedback generation for introductory programming assignments [27], compiler for low-power spatial architectures [20] using solver-aided languages [31, 32], data structure manipulations using visual input-output examples [28], program refactoring using examples [22], inference of efficient synchronization in concurrent programs [33], and relational data representations [12]. A major difference in our approach as compared to these synthesis approaches is the ability to handle some inconsistencies and noise in the specification (examples) due to the probabilistic semantics of our language.

Probabilistic Programming: There has also been a lot of interest lately in the field of probabilistic programming [6, 7, 18] for specifying probability distribution over the input and program space as a first class construct in the language itself, and using efficient inference mechanisms to learn the distributions [19]. Our probabilistic DSL can also be encoded in such frameworks where we define distributions over the set of format descriptors and reformat expressions, and define appropriate probabilistic operators for approximate matching and joint learning. One challenge, however, is that these techniques typically require a lot of examples to infer desired distributions whereas our goal is to learn from very few (possibly 1) examples. Moreover, there are no good mechanisms currently in these frameworks to easily specify declarative constraints over the distributions in contrast to our framework.

10. Limitations and Future Work

One limitation of our system is that it currently supports only a single data type entity in an input string and can not handle multiple data type entities present in the same input. Moreover, our synthesis algorithm is complete for only regular data type strings. We plan on extending our algorithm for supporting multiple data entities including non-regular data type strings. Another limitation is that the likelihood values for `InitField`, `FieldOrder`, and `MatchSF` currently needs to be provided manually by the data type designers. Even though we did not find it challenging to provide these values for the individual data types (as these values can be treated as partial orders), we envision this will become more challenging for more complex data types and their combinations. We are now building a large repository of such data types from real-world spreadsheets so that it can be possible to automatically learn more precise likelihood values for these functions. Finally, we also plan to integrate our system with FlashFill so that it can learn arbitrary combination of syntactic and semantic transformations on input strings.

11. Conclusion

In this paper, we have identified an important subset of the data cleaning problem, namely data type transformations. These data types are present in many spreadsheets and databases, and transforming them presents a big challenge for both developers and end-users as they are often present in multiple formats, some of which may not be known a priori. The inherent ambiguity present in these data types presents another challenge. We present a Programming-by-Example (PBE) technology that combines ideas from inductive synthesis and random interpretation to solve some of these challenges. The previous works in PBE systems learn deterministic programs from a set of examples, while our approach learns weighted programs with probabilistic semantics that allows us to handle non-uniform, unstructured, and ambiguous data. We have implemented our algorithms as an Excel add-in and have evaluated it successfully on several real-world examples.

References

[1] 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.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[3] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI*, pages 295–304, 2005.

[4] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL*, pages 2–15, 2006.

[5] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL*, 2008.

[6] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229, 2008.

[7] A. D. Gordon, T. Graepel, N. Rolland, C. Russo, J. Borgstrom, and J. Guiver. Tabular: A schema-driven probabilistic programming language. In *POPL*, pages 321–334, 2014.

[8] S. Gulwani. *Program Analysis using Random Interpretation*. PhD thesis, EECS Dept., UC Berkeley, 2005.

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

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

[11] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *POPL*, 2005.

[12] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49, 2011.

[13] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *Computer*, 24(12):12–18, 1991.

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

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

[16] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2001.

[17] R. P. Nix. Editing by example. *TOPLAS*, 7(4):600–621, 1985.

[18] A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel. R2: an efficient MCMC sampler for probabilistic programs. In *AAAI*, pages 2476–2482, 2014.

[19] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *PLDI*, pages 208–217, 2015.

[20] 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.

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

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

[23] C. Scaffidi. Topes: Enabling end-user programmers to validate and reformat data, 2009.

[24] C. Scaffidi, B. A. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. In *IUI*, 2009.

[25] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 2012.

[26] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.

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

[28] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, pages 289–299, 2011.

[29] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

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

[31] E. Torlak and R. Bodík. Growing solver-aided languages with rosette. In *Onward*, pages 135–152, 2013.

[32] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54, 2014.

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

[34] Q. Xi and D. Walker. A context-free markup language for semi-structured text. In *PLDI*, pages 221–232, 2010.