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 \[1\] \[13\]. 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\[2\], unstructured\[3\], and ambiguous\[4\]. 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 VBA(.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.
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 \cite{9, 25, 26} that uses intersection to compute common programs for a set of examples and heuristics to learn conditions 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) 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.

At a high level, our system learns a weighted conditional program to transform input dates of the format mm/dd/yyyy to an output format mm/dd/dd-yyy. 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 input string “06/07/2010”. Since the output format for the given input-output example is also ambiguous (mm/dd/yyyy or mm/dd/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:

\[
\begin{align*}
\text{if } "/" \in v_1 \text{ then } (\text{num}_1, "/", \text{num}_2, "/", \text{num}_3) \\
\text{if } "/" \in v_1 \text{ then } (\text{num}_2, "/", \text{num}_1, "/", \text{num}_3) \\
\text{if } "-" \in v_1 \text{ then } (\text{num}_1, ",", \text{num}_2, ",", \text{num}_1)
\end{align*}
\]

where \text{num}_i denotes the \text{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” → “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.

<table>
<thead>
<tr>
<th>Input $v_1$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/21/2010</td>
<td>08/21/2010</td>
</tr>
<tr>
<td>07/24/2010</td>
<td>07/24/2010</td>
</tr>
<tr>
<td>20.08.2010</td>
<td>08/20/2010</td>
</tr>
<tr>
<td>23.08.2010</td>
<td>08/23/2010</td>
</tr>
<tr>
<td>2010-06-07</td>
<td>07/06/2010</td>
</tr>
<tr>
<td>2010-24-08</td>
<td>08/24/2010</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31/79</td>
</tr>
<tr>
<td>2</td>
<td>30/179</td>
</tr>
<tr>
<td>3</td>
<td>31/179</td>
</tr>
<tr>
<td>4</td>
<td>30/1179</td>
</tr>
</tbody>
</table>

Figure 2. Formatting dates to mdyyyy 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 12/5/23 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.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam H. Cook</td>
<td>S. Cook</td>
</tr>
<tr>
<td>Dr. S. R. Barua PhD</td>
<td>S. Barua</td>
</tr>
<tr>
<td>Andrew Hudson</td>
<td>A. Hudson</td>
</tr>
<tr>
<td>Mr. Anand Sagar</td>
<td>A. Sagar</td>
</tr>
<tr>
<td>Dale McDermott Jr.</td>
<td>D. McDermott</td>
</tr>
<tr>
<td>Mrs. Jessie Simpson</td>
<td>J. Simpson</td>
</tr>
</tbody>
</table>

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.


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.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:34:00 PM CST</td>
</tr>
<tr>
<td>2</td>
<td>10:06 EST</td>
</tr>
<tr>
<td>3</td>
<td>12:45 PST</td>
</tr>
<tr>
<td>4</td>
<td>7:25:00 AM MST</td>
</tr>
</tbody>
</table>

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 \( \{ f_1, \ldots, f_n \} \). 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 \( d\equiv \) 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 \( \mathcal{f} \). 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:

- \( m_1 \): single-digit format, e.g. 5 and 11.
- \( m_2 \): two-digit format, e.g. 08 and 12.
- \( m_3 \): full month name, e.g. january and april.

Each sub-field \( f_\mathcal{f} \) is associated with a regular expression that defines the set of values the sub-field can take, e.g. the regular expression for the \( m_2 \) sub-field defines the set of possible values as \( \{01, \ldots, 12\} \). A sub-field \( f_\mathcal{f} \) is associated with a function \( \text{MatchSF} : f_\mathcal{f} \rightarrow \text{double} \), which encodes the likelihood of a field \( f \) being in a sub-field \( f_\mathcal{f} \) 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 \( \text{MatchSF}(y_1) > \text{MatchSF}(y_2) \), which encodes the constraint that the year field is more likely to be in four-digit \( y_2 \) sub-field representation than the two-digit \( y_1 \) representation.

A data type is associated with two functions \( \text{FieldOrder} : f_1, f_2 \rightarrow \text{double} \) and \( \text{InitField} : f \rightarrow \text{double} \). The value
FromCan said to be in its canonical form sub-field for the Each field of a data type is associated with a 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 data instance corresponding to the string “24th Aug 2011” to its canonical form \( v_{UC} \equiv \text{Date}(d \rightarrow 24, m \rightarrow 8, y \rightarrow 2011, dow \rightarrow \text{wednesday}). \) The FromCan function on the set of sub-fields \( \{d_1, m_2, y_2\} \) converts \( v_{UC} \) to the date instance \( v_1 \equiv \text{Date}(d \rightarrow 24, m_2 \rightarrow 08, y_2 \rightarrow 11). \) 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”.

### 3.2 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 \( m_1 \) 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 \( c \) supports three functions: ToCan \( : v \rightarrow v_c \), FromCan \( : v_c \rightarrow \{f_{s_1}, \ldots, f_{s_n}\} \rightarrow v \), and isValidE \( : v \rightarrow \text{Bool} \). 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 data instance corresponding to the string “24th Aug 2011” to its canonical form \( v_{UC} \equiv \text{Date}(d \rightarrow 24, m \rightarrow 8, y \rightarrow 2011, dow \rightarrow \text{wednesday}). \) The FromCan function on the set of sub-fields \( \{d_1, m_2, y_2\} \) converts \( v_{UC} \) to the date instance \( v_1 \equiv \text{Date}(d \rightarrow 24, m_2 \rightarrow 08, y_2 \rightarrow 11). \) 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 \( \pi \) is defined as a sequence of pairs \( (f_s, \delta) \) (sub-field and delimit pair) together with an initial delimit string \( \delta_0 \).

Consider the format descriptor \( \pi_1 \equiv (e, ((d_2, /), (m_2, /), (y_2, e))) \) of the date string “04/05/08”. It interprets the string as the instance \( v_1 \equiv \text{Date}(d \rightarrow 04, m \rightarrow 05, y \rightarrow 08). \) The format descriptors are similar to the custom format strings used in programming languages to format common data types, e.g., the format descriptor \( \pi_1 \) is equivalent to the date format string “dd/mm/yy” in C# (.NET).

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

\[
 w(\pi) = \text{InitField}(f_1) \times \prod_{i=1}^{n} \text{FieldOrder}(f_s_i, f_{s_{i+1}}) \times \prod_{i=1}^{n} \text{MatchSF}(f_{s_i}), \text{where } \pi \equiv (\delta_0, ((f_1, \delta_1) \cdot \ldots \cdot (f_{s_n}, \delta_n))).
\]

We now define the class of regular data type strings, which we will use afterwards for proving the completeness of the synthesis algorithm (Section 4).

**Definition 4 (Regular Format Descriptor).** A format descriptor \( \pi \) is defined to be a regular format descriptor if there exists no substring of the delimit 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 delimit string. For example, the date string “24-09-1998” is not associated with any regular format descriptor as a substring of the initial delimit string “24” matches \( d_1 \) and \( m_1 \) 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{"Date!"}, \langle (d_1, \rangle, (m_1, /), (y_2, \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 \( \pi \) 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 5, first computes the set of all field matches \( M \) for the data type string. It then constructs the parse graph \( G \) from the matches \( M \) and computes the set of format descriptors corresponding to all valid paths in \( G \). It finally returns the set of pairs of data type instances and format descriptors \( \{(v_1, \pi_1)\}\). 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 \((i, j, f_s)\) that corresponds to a match between the substring of data type string \( s[i..j] \) and the regular expression associated with the sub-field \( f_s \), 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 \( M(s) \) such that \( M(s) = \{ (i, j, f_s) \mid \text{MatchF}(s[i..j], f_s) \} \). 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 \( m_1 \) sub-field with parameters \( p \equiv 3 \) and \( c \equiv \text{proper} \), and is denoted as \( m_1, 3, \text{proper} \). A value of \( \infty \) for \( p \) means that the string matches completely with the sub-field regular expression. The set of field matches for the data string \( s_1 = "01/10/999" \) shown in Figure 6 is given by:
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 $(i, j, f_s) \in \mathcal{M}(s)$, the graph consists of a node $n$ with a label $L(n) = (i, j, f_s)$. There are two additional nodes in the graph: the start node $n_0$ with $L(n_0) = (\_1, \_1, \_1)$ and the final node $n_{\infty}$ with $L(n_{\infty}) = (\_\infty, \_\infty, \_\infty)$. There exists an edge from node $n_1$ to $n_2$ in $\mathcal{G}$ with $L(n_1) = (i_1, j_1, f_{s_1})$ and $L(n_2) = (i_2, j_2, f_{s_2})$ if $\forall (i, j, f_s) \in \mathcal{M}(s)$: (i) $n_2$ represents one of the first field matches that occur after $n_1$, i.e., $(i \geq i_1) \implies (i \geq i_2)$, and (ii) there exists no other field match that starts before $j_1$ and overlaps in the interval $j_1$ to $j_2$ ($i < j_1 \implies (j \leq j_1 \vee j = j_2)$). The complexity of constructing a parse graph from a set of matches of size $O(k|s|^2)$ is $O(k|s|^3)$.

The parse graph $\mathcal{G}$ for the data string $s_1 = \{01/10/99\}$ is shown in Figure 8 (b). There is no edge between nodes $(-1, -1, \_1)$ and $(1, 2, d_1)$ since node $(1, 2, d_1)$ is not among the first field matches that occur after the node $(-1, -1, \_1)$ (violates condition (i), e.g. node $(0, 2, d_2)$ occurs before). Similarly, there is no edge between nodes $(3, 4, d_1)$ and $(6, 7, d_1)$ as there exists a node $(3, 5, d_1)$ that overlaps in the interval $(4, 6)$ (violates condition (ii)).

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

1. $p$ starts from start node $n_0$ and ends at final node $n_{\infty}$.
2. no field $f$ is repeated in the field matches of the nodes in $p$, i.e. $\forall n_1, n_2 \in p: L(n_1) = (i_1, j_1, f_{s_1})$, $L(n_2) = (i_2, j_2, f_{s_2})$, we have $f \neq f'$, where $f$ and $f'$ denote the fields of sub-field $f_{s_1}$ and $f_{s_2}$ respectively.

The path $p_1 = [(-1, -1, \_1) \rightarrow (0, 2, d_2) \rightarrow (3, 5, m_1) \rightarrow (6, 8, y_1) \rightarrow (\_\infty, \_\infty, \_\infty)]$ in the parse graph in Figure 8 (b) represents a valid data path. On the other hand, the path $p_2 = [(-1, -1, \_1) \rightarrow (0, 2, d_2) \rightarrow (3, 5, d_1) \rightarrow (6, 8, y_1) \rightarrow (\_\infty, \_\infty, \_\infty)]$ does not represent a valid data 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 $p \equiv [n_0 \rightarrow \cdots (i_k, j_k, f_{s_k}) \rightarrow \cdots n_{\infty}]$ for a data type string $s$ is converted to its corresponding format descriptor $\pi_p = (\delta_0, \{(f_{s_k}, \delta_k)\})$ using the `ConvertFD` function in the following way. The function sets the initial delimiter string $\delta_0$ 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 $(i_k, j_k, m_k)$ on the path $p$ ($1 \leq k < m_1$), it adds $(f_{s_k}, \delta_k)$ to $\pi_p$, where $\delta_k = s[j_k..i_k-1]$. Finally, it sets the delimiter string $\delta_m$ of $\pi_p$ to the substring $s[j_m..1\text{en}(s)]$. Before returning the format descriptor, the function also checks if the entity instance $\pi_p$ corresponding to the format descriptor is a valid data type interpretation by applying the `isValidE` function.

**4.2 Pretty Printing Data Type Instances**

The `printFD` function takes a data type instance $v$ and an output format descriptor $\pi$ as input, and returns the string obtained by pretty printing the instance $v$ in accordance with $\pi$ as shown in Figure 8. The function first adds the initial delimiter string $\delta_0$ to the output string. It then iterates over the sequence of field-delimiter pairs $(f_{s_k}, \delta_k) \in \pi$, and computes the field values to be printed in the output string for each pair. If the field $f_{s_k}$ is present in instance $v$, it first formats the field value in the $f_{s_k}$ sub-field representation using the `Format` function, and then appends the formatted field value with the delimiter string $\delta_k$ to the output string. For example, the `printFD` function pretty prints a date instance $v \equiv \text{Date}(d \mapsto 5, m \mapsto 8, y \mapsto 2011)$ using $\pi \equiv (\epsilon, ((\underline{mm}, 3, \text{pro})", \text{"}, (d_1", \text{"}), (y_2, \epsilon)))$ to the output string "Aug'5 2011".
5. Data Type Transformation Language

We first present a simple proposal for a baseline language \( L_b \) 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 \( \text{case } \pi_{in} \to \pi_{out} \), where \( \pi_{in} \) corresponds to the input format descriptor and \( \pi_{out} \) 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 \( \pi_{in} \), and if the input format descriptors match, then pretty print the data type using the \( \pi_{out} \) format descriptor. We can also learn programs in this language with a simplification of the synthesis algorithm that we present in Section 5.2.

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 with many 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 \( L_c \) 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 \( L_c \) is shown in Figure 9. A program \( P \) consists of a tuple \( (D, \Gamma) \), where \( D \) is a dictionary that stores the frequency counts of all format descriptors of the input strings present in the spreadsheet and \( \Gamma \) denotes a reformat program that consists of a set of reformat expressions \( \rho \). The dictionary \( D \) 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 \( \rho \) is similar to a case statement of \( L_b \) and is denoted by a tuple \( (\pi_{in}, \pi_{out}, w) \) where \( \pi_{in} \) is used to parse input data type strings, \( \pi_{out} \) is used to pretty print the parsed data type instance and \( w \) denotes the likelihood of the reformat expression. A format descriptor \( \pi \) consists of an initial delimiter string and a sequence of pairs of sub-field and delimiters, as described in Section 5.2.

\[
\begin{aligned}
\text{Program } P &:= (D, \Gamma) \\
\text{Fmt. Desc. Dictionary } D &:= \{ (\pi_{in}, \nu_{i}) \}, \\
\text{Reformat Prog. } \Gamma &:= \{ \nu_{i} \}, \\
\text{Reformat Exp. } \rho &:= (\pi_{in}, \pi_{out}, w) \\
\text{Fmt. Desc. } \pi &:= (d_{1}, m_{1})
\end{aligned}
\]

Figure 9. Syntax of data type transformation language \( L_c \).

EXAMPLE 5. The transformation in Example 7 is given by: \( P_1 \equiv (D, \Gamma) \), where \( D = \{ (\pi_1, \frac{1}{2}), (\pi_2, \frac{1}{4}), (\pi_3, \frac{1}{4}), (\pi_4, \frac{1}{4}) \} \), \( \pi_1 \equiv \{ (2002, \lambda), (2007, \lambda), (2008, \lambda), (2010, \lambda) \} \), \( \pi_2 \equiv \{ (2002, \lambda), (2007, \lambda), (2008, \lambda), (2010, \lambda) \} \)

The reformat program \( \Gamma \) consists of reformat expressions that are combinations of format descriptors \( \pi_1 \) and \( \pi_2 \). 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 \( s_1 \) 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). For the ambiguous input string \( s_4 \equiv "2010-06-07" \) that has two interpretations corresponding to the format descriptors \( \pi_3 \) and \( \pi_4 \), the dictionary assigns a higher frequency to \( \pi_3 \) based on the occurrence of another input string "2010-04-08" in the spreadsheet that also has the same format descriptor \( \pi_3 \). In this manner, program \( P_1 \) generates the correct output string for all the input strings.

\[
\begin{aligned}
\text{Program } P_1 &:= (D, \Gamma) \\
\text{Fmt. Desc. Dictionary } D &:= \{ (\pi_1, \nu_{i}) \}, \\
\text{Reformat Prog. } \Gamma &:= \{ \nu_{i} \}, \\
\text{Reformat Exp. } \rho &:= (\pi_{in}, \pi_{out}, w) \\
\text{Fmt. Desc. } \pi &:= (d_{1}, m_{1})
\end{aligned}
\]
function to filter single occurrence of high-confidence inference over multiple occurrences of low-confidence inferences.

The concrete semantics of the language $L_e$ is shown in Figure 12. A program $P = (\mathcal{D}, \Gamma)$ on an input string $s$ generates a set of weighted output strings $\{s_{out,i}, w_i\}$. The semantics of a set $\Gamma$ 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 $s_{out,i}$ from the set of weighted strings obtained from individual reformat expressions $\rho \in \Gamma$. The semantics of a reformat expression $\rho = (\pi_{1a}, \pi_{out}, w)$ 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 the weight $w$ of the weighted instances obtained from the $parseE$ function.

The $parseE(s, \pi, \mathcal{D})$ function parses the string $s$ in accordance with the format descriptor $\pi$. 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 $\pi$. 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 $\{s_i, \pi_i\}$. For each format descriptor $\pi_i$ in the set, it calculates a similarity score $w_i'$ by matching it with the format descriptor $\pi$ using the $matchParse$ function shown in Figure 11. The $matchParse$ function returns a perfect score of 1 if the two format descriptors $\pi_1$ and $\pi_2$ 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 0. The $matchParse$ function finally computes the weight $w_i$ associated with the data type instance $\pi_i$ by multiplying the closeness weight $w_i'$ with the frequency of the format sequence.
matchParse(\(\pi_1, \pi_2\)) : double
\[ \xi : \text{small constant, } k : \text{scaling factor} \]
if(\(\pi_1 = \pi_2\)): return 1 // complete match
if (\(\Pi_2(\pi_1) = \Pi_1(\pi_2)\)) : // sub-fields match
return \(\xi + 2 \times \text{length}(\Pi_2(\pi_1)) \times \xi / k\)
if (\(\Pi_1(\pi_1) = \Pi_2(\pi_2)\)) : // fields match
return \(\xi + \text{length}(\Pi_1(\pi_1)) \times \xi / k\)
return \(\xi\)

Figure 11. The matchParse function to compute the closeness score for two format descriptors \(\pi_1\) and \(\pi_2\).

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

The printE\((V, \pi, w)\) function prints a set of weighted data type instances \(V\) according to the format descriptor \(\pi\) 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 getAllFDes (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 a set of pairs of canonical instances 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.

\[
\begin{align*}
\text{matchParse}(\pi_1, \pi_2) & : \text{double} \\
\xi & : \text{small constant, } k : \text{scaling factor} \\
\text{if}(\pi_1 = \pi_2) \text{ return 1} \quad \text{// complete match} \\
\text{if } (\Pi_2(\pi_1) = \Pi_1(\pi_2)) \text{ \text{ return } } \xi + 2 \star \text{length}(\Pi_2(\pi_1)) \star \xi / k \\
\text{if } (\Pi_1(\pi_1) = \Pi_2(\pi_2)) \text{ \text{ return } } \xi + \text{length}(\Pi_1(\pi_1)) \star \xi / k \\
\text{return } \xi
\end{align*}
\]

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

The semantics of the program \(P_3\) in Example 5 on the input string \(s_2 = “07/24/2010”\) is as follows. Consider the first reformat expression \((\pi_1, \pi_2, w_1) \in \Gamma\). The getAllFD function on \(s_2\) returns the set \(\{(v_1, \pi_1), (v_2, \pi_2)\}\), where \(v_1 \equiv v_2 \equiv \text{Date}(d \mapsto 24, m \mapsto 7, y \mapsto 2010)\) with \(\pi_1\) and \(\pi_2\) format descriptors as defined in Example 5. The parseE function computes the closeness weights \(w'_{\pi_1}\) and \(w'_{\pi_2}\) of format descriptors \(\pi_1\) and \(\pi_2\) respectively with the input format descriptor \(\pi_1\). We have \(w'_{\pi_1} > w'_{\pi_2}\) as \(\pi_1\) is the same as the input format descriptor of \((\pi_1, \pi_1)\). The function then multiplies the closeness weights with respective dictionary frequency of the format descriptors \(\frac{1}{\xi} = \frac{1}{k}\) for both and returns the set \(V \equiv \{(v_1, w'_{\pi_1}), (v_2, w'_{\pi_2})\}\). The printE function then formats the instances in \(V\) according to the output format descriptor \(\pi_1\) and produces the weighted set of output strings \(\{(07/24/2010, w_1 \times w'_{\pi_1}), (07/24/2010, w_1 \times w'_{\pi_2})\}\). On the input string \(s_5 = “2010-06-07”\), the getFD function returns \(\{(v_1, \pi_3), (v_2, \pi_3)\}\), where \(v_1 \equiv \text{Date}(d \mapsto 7, m \mapsto 6, y \mapsto 2010)\) and \(v_2 \equiv \text{Date}(d \mapsto 6, m \mapsto 7, y \mapsto 2010)\). Both format descriptors \(\pi_1\) and \(\pi_2\) do not match with the input parse descriptors of reformat expressions \((\pi_1, \pi_2)\) in \(\Gamma\), and therefore get a low closeness weight \(w'_{\pi_1} < w'_{\pi_2} = w'\) by the matchParse function. The dictionary frequency of format descriptor \(\pi_1\) \(\frac{1}{\xi}\) is higher than that of \(\pi_3\) \(\frac{1}{\xi}\), and therefore the parseE function assigns a higher weight to the date instance \(v_2\) and returns \(\{(v_1, w'_{\pi_1}), (v_2, w'_{\pi_2})\}\). The printE function then returns the set: \(\{(07/6/2010, w_1 \times w'_{\pi_2}), (06/7/2010, w_1 \times w'_{\pi_1})\}\).

Conditional Formatting Programs in \(\mathcal{L}_e\). The input format descriptors \(\pi_{in}\) in the reformat expressions implicitly encode the conditions 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 \(\pi_1\) are formatted using the format descriptor \(\pi_2\) whereas inputs whose format descriptors match closely with \(\pi_3\) are formatted according to \(\pi_4\).

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.

![Figure 13. Generating log filenames based on different date format (us or indian).](image-url)

The desired program \(P_2\) for generating the log filenames can be represented in \(\mathcal{L}_e\) as \(P_2 = (D, \Gamma)\), where \(D = \{(\pi_1, \frac{1}{\xi}), (\pi_2, \frac{1}{\xi}), (\pi_3, \frac{1}{\xi}), (\pi_4, \frac{1}{\xi}), (\pi_5, \frac{1}{\xi}), (\pi_6, \frac{1}{\xi})\}\), \(\pi_1 \equiv (\text{Date}(d \mapsto 24, m \mapsto 7, y \mapsto 2010)), \pi_2 \equiv (\text{Date}(d \mapsto 6, m \mapsto 7, y \mapsto 2010)), \pi_3 \equiv (\text{Date}(d \mapsto 7, m \mapsto 6, y \mapsto 2010)), \pi_4 \equiv (\text{Date}(d \mapsto 6, m \mapsto 7, y \mapsto 2010)), \pi_5 \equiv (\text{Date}(d \mapsto 7, m \mapsto 6, y \mapsto 2010)), \pi_6 \equiv (\text{Date}(d \mapsto 24, m \mapsto 7, y \mapsto 2010))\).

The US date strings match the input format descriptor \(\pi_1\) of \(\Gamma\) and get formatted using the corresponding output format descriptor \(\pi_7\), whereas the Indian date strings are matched with the format descriptor \(\pi_2\) and get formatted using the format descriptor \(\pi_8\).

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 13(a), takes a set of input-output examples \( T \) and the spreadsheet data as inputs, and learns a program \( P \) that can format the input data type strings to the corresponding output strings in the set \( T \). 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 \( (s_i, s_o) \in T \) 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 \( \Gamma \) 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 13(b), takes as input a set of spreadsheet input strings \( T \) 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 \((\pi)\) 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 13(c), learns the set of reformat expressions \( \Gamma \) 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 \((\pi)\) (together with the corresponding canonical entity instances) \( P_1 \) and \( P_2 \) for the strings \( s_i \) and \( s_o \) respectively using the getAllFD function (Lines 2-3). It then iterates over the pairs of their combinations \((vi, \pi)\) \( \in P_1 \times (v_\pi, \pi) \) \( \in P_2 \) to compute consistent pairs for the reformat expressions. Given a pair of format descriptors, the algorithm iterates over the sequence of field-delimiter pairs \((\delta, \epsilon)\) of the output format descriptor and matches the corresponding field values of the data type instances (Lines 12-17). If all the field values match, the reformat expression \((\pi_i, \pi_o)\) is added to the set \( \Gamma \) with the weight \( w = D[\pi_i] \times w(\pi_o) \times w(\pi) \). If any field value is not consistent between the two instances \( v_i \) and \( v_o \), the format descriptor pair \((\pi_i, \pi_o)\) is discarded. Finally, there is an interesting case of a field present in output instance \( v_i \) that is missing from the input instance \( v_o \). 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 \( \gamma \) (Lines 16-17). It then returns the iterative field value matching process. The complexity of GenRefmtExprs function is \( O(k|\pi|^5) \).

**EXAMPLE 7. Consider the date formatting in Example 1 and its corresponding \( L_\text{c} \) program \( P \) 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 getDict function on the input string “08/21/2010” returns a set of two format descriptors \( (\pi_1, \pi_2) \), where \( v_1 \equiv v_2 \equiv \text{Date}(d \mapsto 21, m \mapsto 8, y \mapsto 2010, d\text{ow} \mapsto \text{Saturday}) \). Similarly, the set of format descriptors for other input strings: 07/24/2010 \( \rightarrow \{v_1, \pi_1, v_2, \pi_2\} \), 08.08.2010 \( \rightarrow \{v_1, \pi_1, v_3, \pi_3\} \), 23.08.2010 \( \rightarrow \{v_1, \pi_2, (v_3, \pi_3)\} \), 10-06-07 \( \rightarrow \{v_1, \pi_5, \pi_6\} \), 2010-04-28 \( \rightarrow \{v_1, \pi_5, v_2, \pi_7\} \). The 7 format descriptors occur in total 12 times. The dictionary \( D \) is: \( \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7\} \).

The algorithm then uses the GenRefmtExprs function to compute the set of reformat expressions \( \Gamma \) to format \((s_i \equiv 08/21/2010) \) to \( (s_o \equiv 08/21/2010) \). The function first computes the set of format descriptors: \( P_1 \equiv \{v_1, \pi_1, v_2, \pi_2\} \) and \( P_2 \equiv \{v_1, \pi_1, v_3, \pi_3, v_4, \pi_4, (v_5, \pi_5), \text{Date}(d \mapsto 21, m \mapsto 8, y \mapsto 2010, d\text{ow} \mapsto \text{Saturday}) \). It then iterates over four possible combinations of the input and output format descriptors: \( (v_1, \pi_1, \pi_3, \pi_4) \), \( (v_1, \pi_1, \pi_3, \pi_4, v_2, \pi_2) \), \( (v_1, \pi_1, \pi_5, \pi_6) \), \( (v_1, \pi_1, \pi_5, \pi_6, v_2, \pi_2) \). 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 \( \Gamma \) consisting of all 4 combinations of format descriptors \( (\pi_1, \pi_1, \pi_3, \pi_4) \), \( (\pi_1, \pi_3, \pi_4, v_2, \pi_2) \), \( (\pi_1, \pi_5, \pi_6) \), \( (\pi_1, \pi_5, \pi_6, v_2, \pi_2) \). The algorithm then returns the program \((D, \Gamma)\).

**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 \( \pi_{i_k} \) and \( \pi_{i_l} \) respectively \( s_i \neq s_o \) 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 \( \pi_{i_k} \) and \( \pi_{i_l} \) 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 \( /, / \).

**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 \( \{(24/7/99, 24 July), (14/2/94, 14 February)\} \) is noisy as there is a spelling mistake in the output string for the second example February, 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 \( \{p_1, \cdots, p_n\} \subset P \) that the GenProgram algorithm learns from this example. The claim is that a reformat expression \( p \equiv (\pi_{i_1}, \pi_{i_k}, o) \notin \{p_1, \cdots, p_n\} \) of program \( P \) whose input format descriptor \( \pi_{i_k} \) 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 \( P \) on input string \( i \), only input format descriptors of reformat expressions \( \{p_1, \cdots, p_n\} \) 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 13), the learnt program \( P \) always generates \( o \) as the highest weight output string when executed on the input string \( i \).
Assigning a probabilistic semantics to $L_\mathcal{P}$.

Robust Transformations

There exists a reformat program $\text{GenProgram}$ for reformat programs with regular format descriptors, i.e. if $(\text{GenProgram}(s_i, s_o))$ in Figure 14, we have that the $\text{GenProgram}$ function is guaranteed to synthesize it.

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

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

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

**Example 8 (Missing Information).** An Excel user wanted to format dates present in two different formats $d.m-yyyy$ 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.

**Table:**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 24.9</td>
<td>24 Sep 2010</td>
</tr>
<tr>
<td>2 6-21-2010</td>
<td>21 Jun 2010</td>
</tr>
<tr>
<td>3 29.1</td>
<td>29 Jan 2010</td>
</tr>
<tr>
<td>4 8-15-2010</td>
<td>15 Aug 2010</td>
</tr>
<tr>
<td>5 4-18-2010</td>
<td>18 Apr 2010</td>
</tr>
<tr>
<td>6 16.8</td>
<td>16 Aug 2010</td>
</tr>
</tbody>
</table>

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. It is interesting to note that even though the format descriptor $\gamma$ is not regular, the synthesis algorithm still learns it. The synthesized program $P_3$ is:

**Example 9 (Noisy/Inconsistent Examples).** An excel user wanted to transform dates from $d/m/yyyy$ format to $mm/dd/yyyy$ as shown in Figure 15. The input-output examples in rows 1 and 4 represent the correct formatting operation. The output string in row 2 has a spelling mistake “February” (noisy example), whereas the output string in row 3 “21 September” has a different format (inconsistent example).
The program \( P_1 \) learned by GenProgram from the 4 examples is given by: \( P_1 \equiv (D, \Gamma) \), where \( D \equiv \{ (\pi_1, d_1), (\pi_2, d_2) \} \), \( \pi_1 \equiv (e, ((d_1, l), (m_1, l), (y_2, e))) \), \( \pi_2 \equiv (e, ((d_2, l), (m_1, l), (y_2, e))) \), \( \pi_3 \equiv (e, (((m_3, \infty, \text{iden}), "\text{"}), (d_1, e))) \), \( \pi_4 \equiv (e, (((m_3, \infty, \text{iden}), "\text{"}), (d_2, e))) \), \( \pi_5 \equiv (\text{February}, (((d_1, e))) \), \( \pi_6 \equiv (\text{February}, (((d_1, e))) \), \( \pi_7 \equiv (e, ((d_1, e)), ((m_3, \infty, \text{iden}), "\text{"})) \), \( \pi_8 \equiv (e, (d_2, e)), ((m_3, \infty, \text{iden}), "\text{"})) \), \( \pi_9 \equiv (e, ((d_2, e)), ((m_3, \infty, \text{iden}), "\text{"})) \), \( \pi_{10} \equiv ((\pi_1, \pi_3), w_2), (\pi_1, \pi_4, w_2), (\pi_1, \pi_5, w_1), (\pi_1, \pi_6, w_1), (\pi_1, \pi_7, w_1), (\pi_1, \pi_8, w_1), (\pi_2, \pi_3, w_2), (\pi_2, \pi_4, w_2), (\pi_2, \pi_5, w_3), (\pi_2, \pi_6, w_3), (\pi_2, \pi_7, w_3), (\pi_2, \pi_8, w_3), (\pi_2, \pi_9, w_3)) \), assuming the weights for each reformat expression is equal for simplicity.

The correct reformat expressions with output format descriptors \( \pi_3 \) and \( \pi_4 \) have a higher weight \( w_2 \) than the incorrect reformat expressions with output format descriptors \( \pi_5 \), \( \pi_6 \), \( \pi_7 \) and \( \pi_8 \) \( w_1 \), 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 \( P_1 \) 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 \( L_e \) 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 (\( L_0 \)) can learn the desired transformations for 26 benchmarks whereas our system learns the desired transformations (in \( L_e \)) for all 40 benchmarks.

**Ranking** The probabilistic interpretations in \( L_e \) 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 \( L_e \) (where each feature is incrementally added) is shown in Figure 17(b). Adding weights to reformat expressions in \( L_0 \) (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) further 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.

![Synthesis Time](image)

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

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 \[23\] and number transformations \[29\] 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 \[13\]. 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 \[7\]. 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 \[41, 42\], 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 \[6, 11, 13\] 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