Combining Programming-by-Example with Transformation Discovery from large Databases

Aslihan Özmen[1], Mahdi Esmailoghli[2], Ziawasch Abedjan[3]

Abstract: Data transformation discovery is one of the most tedious tasks in data preparation. In particular, the generation of transformation programs for semantic transformations is tricky because additional sources for look-up operations are necessary. Current systems for semantic transformation discovery face two major problems: either they follow a program synthesis approach that only scales to a small set of input tables, or they rely on extraction of transformation functions from large corpora, which requires the identification of exact transformations in those resources and is prone to noisy data. In this paper, we try to combine approaches to benefit from large corpora and the sophistication of program synthesis. To do so, we devise a retrieval and pruning strategy ensemble that extracts the most relevant tables for a given transformation task. The extracted resources can then be processed by a program synthesis engine to generate more accurate transformation results than state-of-the-art.

1 Introduction

In the era of big data, various large datasets are generated from different sources and stored in various forms. Integration and preparation of raw data from diverse sources with different schema is an important step in every data-driven analysis tasks. The preparation steps include cleaning tasks, such as normalization, entity resolution, and data transformation. In this paper, we address the vital task of data transformation discovery, which refers to the task of generating a transformation function that systematically converts values of columns from one representation to another [Ab15; Ab16; GHS12; Ji17; Mo15; Ro17; Si16].

A data transformation task might be either syntactic or semantic. Syntactic transformation tasks require syntactic manipulations on the input value via a program based on a formula or, a regular expression. A typical example is a date conversion from "XX-XX-XXXX"to "XX/XX/XXXX". Unlike the syntactic manipulation, semantic transformation cannot be performed with a formula or program and the input value only. Semantic transformations requires more context to identify an implicit relationship between the input and output values. For example, there is no formula that can calculate the airport code for a given city name. Practically one needs a lookup operation on external resources. Transformation discovery is hence a tedious task that requires domain knowledge, programming expertise, and access to external datasets. Therefore, research tried to come up with approaches to facilitate this process.

[1] TU Berlin, aslihan.ozmen@thoughtworks.com
[2] Leibniz Universität Hannover, esmailoghli@dbs.uni-hannover.de
[3] Leibniz Universität Hannover abedjan@dbs.uni-hannover.de

doi:10.18420/btw2021-16

There are two general directions of research for data transformation discovery. One line of research is focused on programming-by-example (PBE) approaches that learn a program to syntactically manipulate an input value and search for semantic relationships in additionally provided support tables [GHS12; Ji17; Ro17; Si16]. While PBE is highly innovative and effective on spreadsheet scale, it is bound to a small set of given and related look-up tables and cannot serve newly incoming transformation tasks. A different approach is taken by the DataXFormer system [Ab16; Mo15] and TransformDatabyExample (TDE) [He18], where transformations are to be retrieved from large repositories of tables or functions. Here the given examples are used to swift through large repositories to find potential tables or functions that implement the desired hidden transformation relationship. These approaches have currently the limitation that they rely on the existence of at least one resource that fully implements the whole relationship. Consider the example depicted in Table 1. The user wants to map addresses such as "13701 Riverside Drive Pittsburgh" to the corresponding state abbreviation "PA" and at this end, the user provides a set of examples as depicted in table (a). Related tables in general purpose repositories are however unlikely to contain a resource that explicitly shows this relationship. More likely a table lists mappings of more general concepts, such as city to state as depicted in table (b). DataXFormer and TDE would both not be able to use this related table because they follow a very coarse-granular look-up approach that checks the exact match of input and output examples. Due to the large amount of candidate tables, trying to search for all possible substrings of input and output examples will hurt the performance of the transformation discovery significantly. Furthermore, it might lead to extraction of many irrelevant tables.

In this paper, we tackle exactly this problem. We want to merge the benefits of transformation discovery from large repositories and the PBE approach to harvest more accurate transformation functions. Since applying PBE on a large corpus of data is infeasible, we need to define pruning techniques that effectively limit the set of relevant tables. This problem is hard, as we do not know upfront which substrings inside the values of a table might be relevant for a transformation task at hand.

Tab. 1: Nested Syntactic transformation with lookup operation

(a) Example pair and input values

Input	Output
13701 Riverside Drive Pittsburgh	PA
27700 Medical Center Road Scarborough	**?**
4270 North Blackstone Avenue Tucson	**?**

(b) Related table

Town	State	County
Pittsburgh	PA	Allegheny
Claremore	OK	Rogers
Scarborough	ME	Cumberland
Tucson	AZ	Pima
Indianapolis	IN	Marion

To this end, we propose PROTEUS that extends the DataXFormer system [Ab15; Ab16; Mo15] to be able to detect more complex transformations. PROTEUS is able to detect transformations that do not explicitly appear in the given data source. We achieve this by relaxing the exact matching approach of the DataXFormer system. We propose a multi-step filtering strategy that successively prunes irrelevant tables and cells before program synthesis

is applied. Finally, we enable parallel processing of the PBE-component to increase the scalability of our proposed system. In summary, our contributions in this paper are:

- We enhance the data transformation tool DataXFormer [Ab15; Ab16; Mo15] with the PBE framework. Therefore, our proposed system is able to detect semantically and syntactically more complex transformations.
- To make PBE feasible on millions of tables, we propose a set of pruning rules for filtering and reducing the search space to use only web tables and table entries that are relevant for the transformation task.
- By considering transformation steps per example-pair independently, we are able to process the PBE-based transformation detection task in parallel, so we are able to process the same task in the scale of millions of web tables.

2 Related Work

Several lines of research attempt to solve the task of the data transformation discovery. DataXFormer [Ab15; Ab16; Mo15] serves as the foundation of our approach. It leverages different resource types, such as Web tables [LB17; Ya12], Web forms, knowledge bases, and expert sourcing, for example-based transformation discovery. For this purpose, they proposed an inverted index for fast retrieval of relevant tables and a web form wrapper generator. The main limitation of DataXFormer is that it cannot support transformations that do not exactly match individual resource entries, i.e., tuple values of a web table.

Another line of research concerns the discovery of transformations from smaller data structures [GHS12; Ji17; SG12; Si16]. These approaches are referred to as programming-by-example (PBE) techniques, which synthesize transformation programs using input and output examples and a set of transformation operators on a small domain of data. Consequently, in case of bigger scale use cases such as web tables, they lead to real-time performance issues. In this paper, we try to combine the benefit of both lines of research. Our approach improves on DataXFormer by synthesizing transformation functions require the combination of multiple tables, by separating the retrieval process from the transformation generation process. We solve the scalability issue of PBE by defining effective pruning rules and filtering strategies.

Other than the scalability problem, systems such as REFAZER [Ro17] suffer from generalizability problem. REFAZER only uses specialized Domain Specific Language to generate codes for syntactic transformations. This system is only able to transform repetitive code transformations by learning the observations but it is not general enough to apply the transformations on pure text with unpredictable domain. Similar to Foofah, REFAZER does not apply semantic transformations at all.

Finally, there is a line of research on interactive transformation script generation [HHK15; Ji19; Ka11]. The main focus of this line of research is on interactive transformation

generation and user-support during the transformation task, which is complementary to the focus of transformation discovery for semantic transformations.

3 System Overview

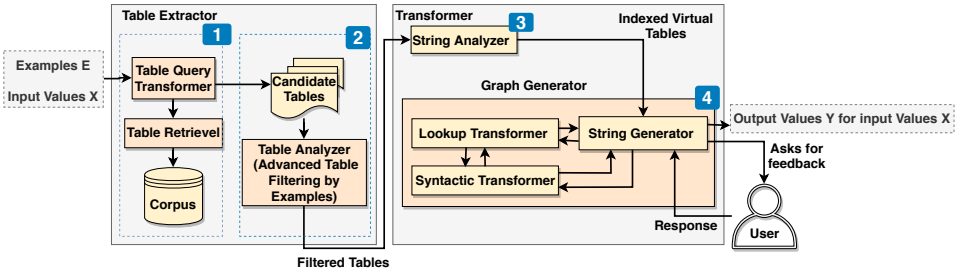

Fig. 1: System overview of PROTEUS

Figure 1 shows the overall PROTEUS architecture. PROTEUS receives a set of values $X = \{x_i | 1 \leq i \leq n\}$ that are desired to be transformed and a set of example pairs $E = \{(x_i, y_i) | x_i \in X, 1 \leq i \leq m\}$ as inputs. In the end, the system outputs the discovered transformation results Y for the remaining input values X as $\{(x_i, y_i)\}$, where y_i is the detected transformation of x_i. PROTEUS consists of two main components: *Web Table Extractor* and *Transformer*. *Web Table Extractor* is responsible for detecting the most related web tables from the table corpus and the *Transformer* component detects and generates the transformations for values in X.

In the first step, we dynamically generate queries to extract candidate tables from the corpus. The tables are indexed via an inverted index as proposed in prior work [Ab16]. The index maps tokens to tables and vice versa. Unlike DataXFormer, we do not only extract tables with exact matches but relax the extraction query to accommodate more resources and postpone the refinement to the PBE engine in *Transformer*. Because of the relaxation technique, many of the tables might be irrelevant for the transformation task at hand. Therefore, in the second step, we filter candidate tables that are unlikely to support the transformation task. In the third step, we send the relevant candidate tables to the *Transformer* component. *Transformer* first applies finer-granular pruning rules and scoring functions to remove potential noisy, irrelevant, and duplicate records from the extracted tables. For this purpose it uses a row-level filtering approach to detect the most relevant entries inside each selected external table. In the fourth step, we leverage the PBE framework to generate syntactic manipulation programs for the relevant tables. The programs can be represented as graphs of operations that connect input values to the output values of the provided examples in E. The common path between all these graphs is the answer for the transformation task.

The graph generation for each example pair can be performed in parallel, which allows us to scale the process for more extracted tables. If there is not a common path, the user can

choose the right graph among two disjoint graphs. In the end, PROTEUS applies the final chosen graph as the desired program to generate the output for the remaining input values.

4 Web Table Extractor

The *Web Table Extractor* takes example pairs *(E)* and returns transformation-related web tables. It consists of the two sub-components: *Table Query Transformer* and *Table Analyzer*.

The **Table Query Transformer** is implemented as an extension of the query generator of DataXFormer. The original DataXFormer query generator creates a single SQL query with two `IN` operators, one for the input values and one for the output values to identify tables that contain at least τ example pairs. To obtain more data, we relax this query twofold. First, PROTEUS tokenizes every input and output value from the example pairs and uses each generated token independently for table retrieval. For instance, if the given example-pair is "13701 Riverside Drive Pittsburgh → PA", it generates queries to find tables that contain at least one token from the input (here: "13701","Riverside","Drive", "Pittsburgh") and one token from the output (here: "PA"). If no table is found, the system generates a new query to find tables that contain any of the tokens as a substring of at least one entry. This is done leveraging the `LIKE` predicate in SQL. The first query often returns a large number of web tables, in which case we drop the time consuming query with the `LIKE` predicate.

A straightforward approach to detect the desired transformation would be to evaluate every single retrieved table from the *Table Query Transformer* sub-component. This approach is time-consuming and error-prone because of the large number of irrelevant tables. Therefore, **Table Analyzer** further reduces the search space by filtering the irrelevant tables. First, it determines whether at least the row alignment for input/output pairs in each table is correct [Ab16]. Input and output pairs in each example should appear in the same rows. While DataXFormer checks the row alignment for exact input/output values, we check the row alignment of partial tokens and substrings of the input/output values. Note that this alignment is necessary for the program generation later in order to obtain consistent program paths that connect the input value to the output value. Furthermore, we also want to get rid of tables where the tokens are randomly aligned. Especially in long column values the occurrence of multiple tokens is likely. Thus we also drop tables where the Jaccard similarity of the aligned rows and the corresponding original input examples is below 50%.

The remaining selected tables that pass the alignment test, will be rated based on their relatedness to the transformation task. Depending on the number of matches and the strength of the matches, tables differ in the degree of relatedness to the transformation task. Similar to DataXFormer, PROTEUS leverages a *refine* technique based on expectation - maximization (EM) to update the scores of tables and found alignments. However, our approach differs in two ways. First, we also accommodate the fact that not every token is equally important and further, we are only interested in obtaining the scores while DataXFormer uses the final scores to choose the transformation. In the expectation step the scores of the tables are

updated and in the maximization step the scores of the instances. In each step the scores of the other step is used to update the scores. This process converges as soon as the the updates are below a very low threshold ϵ. The process starts with scores that reflect the ratio of existing example pairs inside the extracted tables. However, as we are considering partial values, i.e., tokens, we also consider the fact that not every token is equally important. For instance, if there is a match for value "of" in table T_1 and another match for value "New York City" in table T_2. It is desirable to give more weight to the table that contains "New York City" because it is more specific and thus more likely to support our specific task. To reflect this property in the EM formula, we use the Inverse Document Frequency(IDF). Higher IDF score means that the token is more specific. Therefore in the EM model, we multiply the score of each occurring example with its IDF. Finally, *Table Analyzer* sends the tables along their calculated scores to the *Transformer* component.

5 Transformer

The *Transformer* component has two sub-components: *String Analyzer* and *Graph Generator*.

String Analyzer takes the tables from the *Web Table Extractor* sorted by their relatedness scores and finds the most relevant table entries (rows) for each input value. Each input value can be matched with more than one entry inside a table. Therefore, *String Analyzer* reduces them to the most promising match to unburden the PBE step from generating programs for the matches that are less likely to be a candidate for the transformation.

To find the most related entry to the input value of an example pair, we leverage a score based on the Longest Common Substring (LCS) [AO11]. This is consistent with most PBE algorithms, which use common string patterns between two strings to detect the transformations. We calculate the LCS score between each input value from the example pairs and all the matched entries in the related tables. The table entry with the maximum LCS score is selected to be used in the program generation phase. If there are two table entries with the same LCS score, the edit distance, a.k.a. Levenshtein distance [Le66] will break the tie. The entry with lower edit distance will be picked as more related.

Consider the example in Table 2. The first two example pairs in the left table are provided by the user who wants to transform the address of the last two inputs to their corresponding state codes. The table on the right reflects a candidate web table, which contains aligned input/output tokens of our examples (the result of the steps before). In this candidate table, the first row and the value "Redmond" has the highest LCS with the given input example "1906 Jackson Way Redmond". The same way, we compute the LCS for the second example pair. In this particular case, there are two entries with the same LCS, the third and the fifth rows contain both entries with the LCS length of 8. Here, the similarity check would be the tie breaker, which chooses the entry in the third row. The String Analyzer outputs the top related tables and the injective mapping of example pairs to rows in each table.

Tab. 2: Simple LCS and similarity check example in *Transformer*

(a) Input values (X) including the example pairs (E)

Input	Output
1906 Jackson Way **Redmond**	**WA**
1703 Red Creek Road **Richmond**	**VA**
101 Sundown Blvd **Sacramento**	**?**
510 Green Lake Road **Beaumont**	**?**

(b) Related web table

C1	C2
Redmond	**WA**, Washington
Redm. branch is open till 24/02	**WA**, Washington
Richmond	**VA**, Virginia
Beaumont	**TX**, Texas
Richmond branch is open only till 03/03	**VA**, Virginia
Sacramento	**CA**, California

Graph Generator discovers the programs that convert an input value to a provided output transformation following the PBE approach [GHS12]. This sub-component receives the final set of tables, top related entries, and the corresponding example pairs and then generates the corresponding Directed Acyclic Graph (DAG) for each example pair. In this graph each node represents a transformation state of a value and each edge corresponds to a program that changed the value of its source node to the value in the target node. The PBE approach combines programs for syntactic manipulations, which are concatenations of regular expressions based on sub-strings and table mappings. For instance, a syntactic manipulation would be a program that converts "Bob.Franklin@tu-berlin.de" to "Bob Franklin" by learning to remove the substring after "@" and replace the first dot symbol by a space. Depending on the provided resources, several possible paths can be generated to map an input example to its output value.

Considering our running example, the graph for the first example pair shown in Figure 2 is generated using the first row in the web table, because based on the LCS score, the first row in the web table is the closest to the example pair. Edges with the labels of $\texttt{Prog}_i$ represent the generated code snippets and η_j represents the value in the j^{th} node. Here, $\texttt{Prog}_1$ represents the program to extract the first alphabetic string after the last space from the input value. The next program encodes the mapping from "Redmond" to "WA, Washington". And finally, $\texttt{Prog}_3$ extracts the first alpha-numerical substring from "WA, Washington".

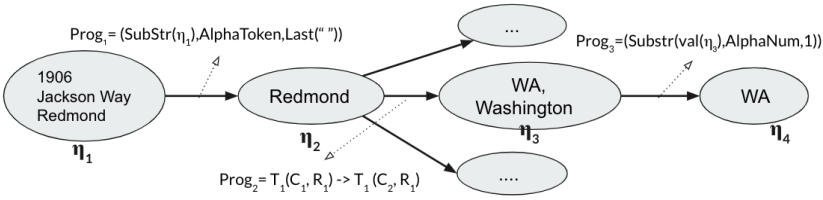

Fig. 2: Generated graph for the first example pair in *Transformer*

Graph Generator greedily starts with the examples inside the table with the highest relatedness score and starts to generate program graphs for them. Whenever it generates the

graphs of the next example pair it starts to find the intersection of the two sets of graphs and this way gradually identifies the graph that covers all example pairs.

In the end, the intersection of all paths will be the program that can be used to transform the remaining inputs. Once all the input values are transformed, all tables have been processed, or the result of graph intersection is empty, the algorithm stops. The latter suggests that no transformation could be found that covers all example pairs but there are still other tables that might contain a fitting graph path. In such cases, PROTEUS can return possible graph candidates and ask the user to choose the correct one. PROTEUS can then proceed with the chosen graph to find a match for remaining input values in the remaining tables. Figure 3 shows the application of graph intersection on our running example.

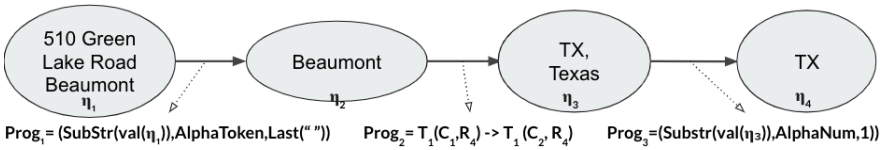

Fig. 3: Path after the result of the final intersection

We generate the graphs independently from each other so that we are able to run each graph generation and graph intersection in parallel. For instance, while we simultaneously generate graphs for first and second example pairs, once we start intersecting them, we generate the third example pair at the same time and so on. Defining the transformation tasks as sub-independent processes enables us to perform transformations in parallel.

6 Evaluation

In this section, we show the efficiency and effectiveness of our transformation discovery system and compare it to the state-of-the-art.

6.1 Data and experimental setup

We use the Dresden Web Table Corpus[4] [Eb15] as the table corpus to extract transformation-related tables. It contains 145 million web tables from about 4 billion web pages. We evaluate our approach by running 20 semantic data transformation tasks. Each transformation task has 3 unique example pairs. Four of these transformation tasks (first two rows in Table 3) are created manually to ensure that these tasks require syntactic manipulations before and after lookup operations e.g., *"Country is Ukraine → City name: Kiev"*. The rest of the tasks are public benchmarks found in Bing Search Engine query logs and asked by Data Scientists and BI Analysts in StackOverflow. Table 3 shows all the transformation tasks used in this paper. We compare our system to four state-of-the-art approaches: DataXFormer [Ab15;

[4] https://wwwdb.inf.tu-dresden.de/misc/dwtc/

Tab. 3: Transformation tasks used in this paper

Input	Output	Input	Output
Country Names	Capitals	Country Names with Attributes	Capitals
Element names	Boiling point	Country names	Denonyms
Color Number	Color code	Regular time	Military Time
Hijri	Gregorian Caleandar	Company Address	State
MB	GB	yyyymmdd	Datetime
Number	Numeric Padding	String	Camelize Casing
Regular Format	ISBN Format	Datetime	Month
Cookies	Domain name	Month number	Month name
CUSIP	Ticker	Product	Company
MEME	Filename	Time span	hrs mins secs

Ab16; Mo15], Gulwani's approach [GHS12], FlashFill [Gu16], and Foofah [Ji17]. We ran our experiments on a machine with 2.9 GHz Intel Dual Core CPU and 8 GB RAM. Codes are available in our GitHub repository[5].

6.2 Results

Tab. 4: Coverage, Precision, and Recall for 20 tasks.

Systems	#Coverage	Precision	Recall
PROTEUS	**95% (19/20)**	**90%**	**78%**
FlashFill	50% (10/20)	50%	50%
Gulwani	50% (10/20)	50%	50%
DataXFormer	30% (6/20)	24%	17%
Foofah	5% (1/20)	5%	5%

Coverage. We define coverage as the ratio of the number of transformation tasks where the system returns a correct transformation output for at least one of the input values. As depicted in Table 4, PROTEUS achieves considerably higher coverage than the other baselines. It achieves 95% coverage, which means that our system generates output for 19 out of the 20 transformation tasks. High coverage conveys the fact that PROTEUS is more robust to the noisy data which is common in web tables. Noisy and erroneous data results in lower exact match rate and in the end, it will lead to less related tables and lower coverage rate. Foofah, due to the lack of ability in Regex matching, DataXFormer because of only using exact match, Flashfill because of the lack of lookup operation and being limited to only program generation, and Gulwani's system due to the fact that it requires clean and user-defined tables for the transformation tasks, have low coverage. The only task that was not covered by PROTEUS was "hijri to the gregorian calendar". This conversion is difficult based on static web tables. Consider the following input/output example pair given by the user:

"11 Shawwal 1430" → "Wednesday 30 September 2009 C.E"

PROTEUS cannot find web tables that contain the values *"11 → 30"* and *"1430 → 2009"*.

Effectiveness. As shown in Table 4, PROTEUS outperforms other systems in terms of Precision and recall. Leveraging token-based matching that retrieves wider range of candidate tables from the corpus improves the recall. More candidate tables increase the

[5] https://github.com/aslihanozmen/Proteus

chance to find the desired transformation. On the other hand, the LCS-based entry filtering strategy allows the system to pick the most fitting table entries to the input values and drop the irrelevant ones. Therefore, the final transformations are only generated using the most promising candidates improving the precision. Gulwani's approach because of being limited to the provided tables, and FlashFill and Foofah because of their limitation to syntactic transformations achieve lower precision and recall.

Runtime. As shown in Figure 4, PROTEUS is faster and more scalable than DataXFormer and Gulwani's approach. We excluded FlashFill and Foofah from the runtime evaluation because they only cover syntactic manipulations based on the input values. PROTEUS is fast due to its pruning rules, and parallelization of the graph processing. It is also faster than DataXFormer because before any look up operations, PROTEUS checks whether syntactic manipulations of the input values alone can perform the transformation for all the input values. In these five cases it refrained from looking for more complicated semantic transformations. These five tasks are "MB → GB", "Cookies → Domain name", "Regular format → ISBN format", "Number → Numeric padding", and "String → Camelize casing".

String Analyzer. Our LCS-based entry scoring technique that only keeps the top related entry, reduces the number of generated *Progs* more than 50 times compared to the raw PBE approach. This reduction is due to the elimination of the irrelevant table entries before the program generation phase. As an example, for the transformation task "CUSIP" → "Ticker", without using our LCS-based pruning, the first two generated graphs for the first two examples contain overall 44, 075 *Progs* (edges). Intersecting these two graphs leads to a graph with 9, 276 *Progs*. Applying the LCS pruning, the total number of *Progs* for the first two generated graphs decreases to 750 and their intersection to 8 *Progs*.

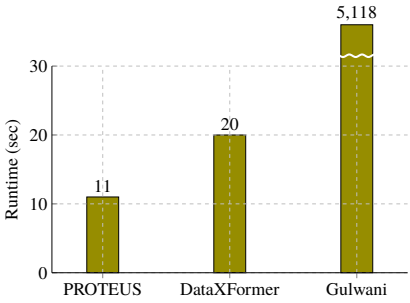

Fig. 4: Runtime comparison

7 Conclusion

In this paper, we proposed a system to combine transformation discovery with programming by example. The general idea was to first relax the transformation discovery systems to obtain more partially relevant resources and then filter irrelevant resources with PBE. Our experiments show that the approach is covering more transformation tasks than state-of-the-art. Unlike the state-of-the-art, PROTEUS has higher robustness to noisy data that is very common in web tables. In future, we would like to make the filtering steps more dynamic to avoid heuristic-based thresholds.

Acknowledgements. This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445.

References

[Ab15] Abedjan, Z.; Morcos, J.; Gubanov, M. N.; Ilyas, I. F.; Stonebraker, M.; Papotti, P.; Ouzzani, M.: Dataxformer: Leveraging the Web for Semantic Transformations. In: CIDR. 2015.

[Ab16] Abedjan, Z.; Morcos, J.; Ilyas, I. F.; Ouzzani, M.; Papotti, P.; Stonebraker, M.: DataXFormer: A robust transformation discovery system. In: ICDE. Pp. 1134–1145, 2016, URL: https://doi.org/10.1109/ICDE.2016.7498319.

[AO11] Arnold, M.; Ohlebusch, E.: Linear Time Algorithms for Generalizations of the Longest Common Substring Problem. Algorithmica 60/4, pp. 806–818, 2011, URL: https://doi.org/10.1007/s00453-009-9369-1.

[Eb15] Eberius, J.; Braunschweig, K.; Hentsch, M.; Thiele, M.; Ahmadov, A.; Lehner, W.: Building the Dresden Web Table Corpus: A Classification Approach. In: BDC, 2015. IEEE Computer Society, pp. 41–50, 2015, URL: https://doi.org/10.1109/BDC.2015.30.

[GHS12] Gulwani, S.; Harris, W. R.; Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM 55/8, pp. 97–105, 2012, URL: https://doi.org/10.1145/2240236.2240260.

[Gu16] Gulwani, S.: Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In: Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706. Springer-Verlag, Berlin, Heidelberg, pp. 9–14, 2016, ISBN: 978-3-319-40228-4, URL: https://doi.org/10.1007/978-3-319-40229-1_2, visited on: 03/10/2019.

[He18] He, Y.; Chu, X.; Ganjam, K.; Zheng, Y.; Narasayya, V. R.; Chaudhuri, S.: Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. Proc. VLDB Endow. 11/10, pp. 1165–1177, 2018, URL: http://www.vldb.org/pvldb/vol11/p1165-he.pdf.

[HHK15] Heer, J.; Hellerstein, J. M.; Kandel, S.: Predictive Interaction for Data Transformation. In: CIDR. 2015, URL: http://cidrdb.org/cidr2015/Papers/CIDR15%5C_Paper27.pdf.

[Ji17] Jin, Z.; Anderson, M. R.; Cafarella, M. J.; Jagadish, H. V.: Foofah: Transforming Data By Example. In: SIGMOD. ACM, pp. 683–698, 2017, URL: https://doi.org/10.1145/3035918.3064034.

[Ji19] Jin, Z.; Cafarella, M. J.; Jagadish, H. V.; Kandel, S.; Minar, M.; Hellerstein, J. M.: CLX: Towards verifiable PBE data transformation. In: EDBT. Pp. 265–276, 2019.

[Ka11] Kandel, S.; Paepcke, A.; Hellerstein, J. M.; Heer, J.: Wrangler: interactive visual specification of data transformation scripts. In: CHI. Pp. 3363–3372, 2011.

[LB17] Lehmberg, O.; Bizer, C.: Stitching Web Tables for Improving Matching Quality. PVLDB 10/11, pp. 1502–1513, 2017.

[Le66] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. Vol. 10. 8, pp. 707–710, 1966.

[Mo15] Morcos, J.; Abedjan, Z.; Ilyas, I. F.; Ouzzani, M.; Papotti, P.; Stonebraker, M.: DataXFormer: An Interactive Data Transformation Tool. In: SIGMOD. ACM, pp. 883–888, 2015, URL: https://doi.org/10.1145/2723372.2735366.

[Ro17] Rolim, R.; Soares, G.; D'Antoni, L.; Polozov, O.; Gulwani, S.; Gheyi, R.; Suzuki, R.; Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of ICSE. IEEE, pp. 404–415, 2017, URL: https://doi.org/10.1109/ICSE.2017.44.

[SG12] Singh, R.; Gulwani, S.: Learning Semantic String Transformations from Examples. CoRR abs/1204.6079/, 2012, arXiv: 1204.6079, URL: http://arxiv.org/abs/1204.6079.

[Si16] Singh, R.: BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations. Proc. VLDB Endow. 9/10, pp. 816–827, 2016, URL: http://www.vldb.org/pvldb/vol9/p816-singh.pdf.

[Ya12] Yakout, M.; Ganjam, K.; Chakrabarti, K.; Chaudhuri, S.: Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In: SIGMOD. Pp. 97–108, 2012.