Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations

Yeye He[1], Xu Chu[2]*, Kris Ganjam[1], Yudian Zheng[3]†, Vivek Narasayya[1], Surajit Chaudhuri[1]

[1]Microsoft Research, Redmond, USA
[2]Georgia Institute of Technology, Atlanta, USA
[3]Twitter Inc., San Francisco, USA

[1]{yeyehe, krisgan, viveknar, surajitc}@microsoft.com

[2]xu.chu@cc.gatech.edu

[3]yudianz@twitter.com

ABSTRACT

Today, business analysts and data scientists increasingly need to clean, standardize and transform diverse data sets, such as name, address, date time, and phone number, before they can perform analysis. This process of data transformation is an important part of data preparation, and is known to be difficult and time-consuming for end-users.

Traditionally, developers have dealt with these longstanding transformation problems using custom code libraries. They have built vast varieties of custom logic for name parsing and address standardization, etc., and shared their source code in places like GitHub. Data transformation would be a lot easier for end-users if they can discover and reuse such existing transformation logic.

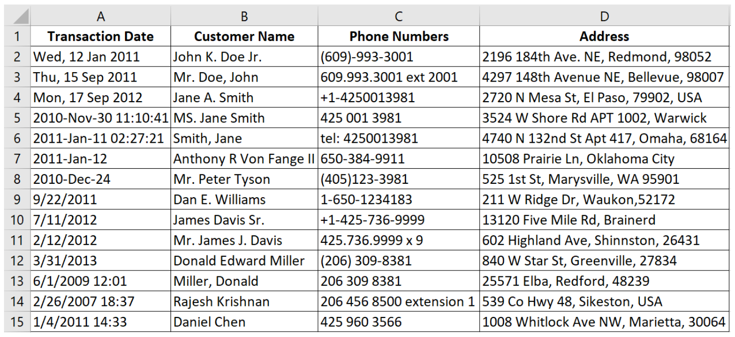
We developed *Transform-Data-by-Example* (*TDE*), which works like a search engine for data transformations. *TDE* "indexes" vast varieties of transformation logic in source code, DLLs, web services and mapping tables, so that users only need to provide a few input/output examples to demonstrate a desired transformation, and *TDE* can interactively find relevant functions to synthesize new programs consistent with all examples. Using an index of 50K functions crawled from GitHub and Stackoverflow, *TDE* can already handle many common transformations not currently supported by existing systems. On a benchmark with over 200 transformation tasks, *TDE* generates correct transformations for 72% tasks, which is considerably better than other systems evaluated. A beta version of *TDE* for Microsoft Excel is available via Office store[1]. Part of the *TDE* technology also ships in Microsoft Power BI.

*Work done at Microsoft Research.

†Work done at Microsoft Research.

[1]`https://aka.ms/transform-data-by-example-download`

Proceedings of the VLDB Endowment, Vol. 11, No. 10
Copyright 2018 VLDB Endowment 2150-8097/18/06... $ 10.00.
DOI: https://doi.org/10.14778/3231751.3231766



	A	B	C	D
1	**Transaction Date**	**Customer Name**	**Phone Numbers**	**Address**
2	Wed, 12 Jan 2011	John K. Doe Jr.	(609)-993-3001	2196 184th Ave. NE, Redmond, 98052
3	Thu, 15 Sep 2011	Mr. Doe, John	609.993.3001 ext 2001	4297 148th Avenue NE, Bellevue, 98007
4	Mon, 17 Sep 2012	Jane A. Smith	+1-4250013981	2720 N Mesa St, El Paso, 79902, USA
5	2010-Nov-30 11:10:41	MS. Jane Smith	425 001 3981	3524 W Shore Rd APT 1002, Warwick
6	2011-Jan-11 02:27:21	Smith, Jane	tel: 4250013981	4740 N 132nd St Apt 417, Omaha, 68164
7	2011-Jan-12	Anthony R Von Fange II	650-384-9911	10508 Prairie Ln, Oklahoma City
8	2010-Dec-24	Mr. Peter Tyson	(405)123-3981	525 1st St, Marysville, WA 95901
9	9/22/2011	Dan E. Williams	1-650-1234183	211 W Ridge Dr, Waukon,52172
10	7/11/2012	James Davis Sr.	+1-425-736-9999	13120 Five Mile Rd, Brainerd
11	2/12/2012	Mr. James J. Davis	425.736.9999 x 9	602 Highland Ave, Shinnston, 26431
12	3/31/2013	Donald Edward Miller	(206) 309-8381	840 W Star St, Greenville, 27834
13	6/1/2009 12:01	Miller, Donald	206 309 8381	25571 Elba, Redford, 48239
14	2/26/2007 18:37	Rajesh Krishnan	206 456 8500 extension 1	539 Co Hwy 48, Sikeston, USA
15	1/4/2011 14:33	Daniel Chen	425 960 3566	1008 Whitlock Ave NW, Marietta, 30064

Figure 1: A sales dataset with heterogeneous values.

PVLDB Reference Format:
Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narrasayya and Surajit Chaudhuri. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *PVLDB*, 11(10): 1165-1177, 2018.
DOI: https://doi.org/10.14778/3231751.3231766

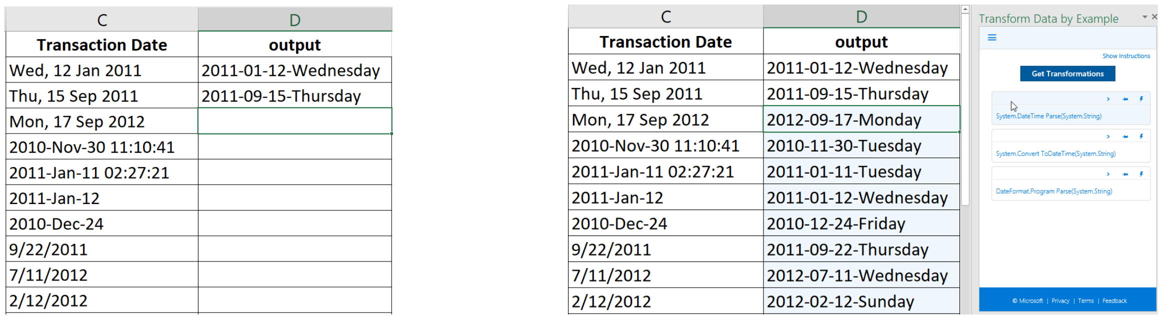
1. INTRODUCTION

Today, end users such as business analysts and data scientists increasingly need to perform analysis using diverse data sets. However, raw data collected from different sources often need to be *prepared* (e.g., cleaned, transformed, and joined) before analysis can be performed. This is difficult and time-consuming for end-users – studies suggest that analysts spend up to 80% of time on data preparation [13].

In light of the difficulties, there is a recent trend in the industry called *self-service data preparation* [15]. Gartner estimates this market to be worth over $1 billion by 2019 [15]. In this work we consider *self-service data transformation*, which is a major component of data preparation. Specifically, we focus on *row-to-row transformations*, where the input is a row of string values and the output is also a string[2]. We illustrate such transformations with an example.

Figure 1 gives an example data set with sales transactions of customers from different sales channels, with transaction dates, customer names, etc. Note that values from same columns are highly heterogeneous, which often happen if data come from different sources, or if they are manually

[2]Sorting a *set* of values, for instance, is not a row-to-row transformation.



C	D
Transaction Date	**output**
Wed, 12 Jan 2011	2011-01-12-Wednesday
Thu, 15 Sep 2011	2011-09-15-Thursday
Mon, 17 Sep 2012	
2010-Nov-30 11:10:41	
2011-Jan-11 02:27:21	
2011-Jan-12	
2010-Dec-24	
9/22/2011	
7/11/2012	
2/12/2012	

C	D
Transaction Date	**output**
Wed, 12 Jan 2011	2011-01-12-Wednesday
Thu, 15 Sep 2011	2011-09-15-Thursday
Mon, 17 Sep 2012	2012-09-17-Monday
2010-Nov-30 11:10:41	2010-11-30-Tuesday
2011-Jan-11 02:27:21	2011-01-11-Tuesday
2011-Jan-12	2011-01-12-Wednesday
2010-Dec-24	2010-12-24-Friday
9/22/2011	2011-09-22-Thursday
7/11/2012	2012-07-11-Wednesday
2/12/2012	2012-02-12-Sunday

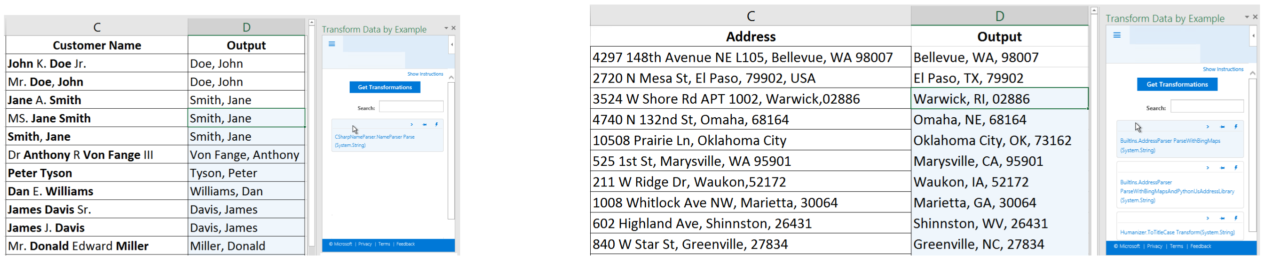
Transform Data by Example
Show Instructions
Get Transformations
© Microsoft | Privacy | Terms | Feedback

System.DateTime Parse(System.String)

System.Convert ToDateTime(System.String)

DateFormat.Program Parse(System.String)

Figure 2: *TDE* transformation for date-time. (Left): user provides two desired output examples in column-D, for the input data in column-C. (Right): After clicking on the "Get Transformation" button, *TDE* synthesizes programs consistent with the given examples, and return them as a ranked list within a few seconds. Hovering over the first program (using System.DateTime.Parse) gives a preview of all results (shaded in green).



C	D
Customer Name	**Output**
John K. **Doe** Jr.	Doe, John
Mr. **Doe, John**	Doe, John
Jane A. **Smith**	Smith, Jane
MS. **Jane Smith**	Smith, Jane
Smith, Jane	Smith, Jane
Dr **Anthony** R **Von Fange** III	Von Fange, Anthony
Peter Tyson	Tyson, Peter
Dan E. **Williams**	Williams, Dan
James Davis Sr.	Davis, James
James J. **Davis**	Davis, James
Mr. **Donald** Edward **Miller**	Miller, Donald

Transform Data by Example
Show Instructions
Get Transformations
Search:
© Microsoft | Privacy | Terms | Feedback

CSharpNameParser.NameParser Parse (System.String)

C	D
Address	**Output**
4297 148th Avenue NE L105, Bellevue, WA 98007	Bellevue, WA, 98007
2720 N Mesa St, El Paso, 79902, USA	El Paso, TX, 79902
3524 W Shore Rd APT 1002, Warwick,02886	Warwick, RI, 02886
4740 N 132nd St, Omaha, 68164	Omaha, NE, 68164
10508 Prairie Ln, Oklahoma City	Oklahoma City, OK, 73162
525 1st St, Marysville, WA 95901	Marysville, CA, 95901
211 W Ridge Dr, Waukon,52172	Waukon, IA, 52172
1008 Whitlock Ave NW, Marietta, 30064	Marietta, GA, 30064
602 Highland Ave, Shinnston, 26431	Shinnston, WV, 26431
840 W Star St, Greenville, 27834	Greenville, NC, 27834

Transform Data by Example
Show Instructions
Get Transformations
Search:
© Microsoft | Privacy | Terms | Feedback

BuiltIns.AddressParser ParseWithBingMaps (System.String)

BuiltIns.AddressParser ParseWithBingMapsAndPythonUsAddressLibrary (System.String)

Humanizer.ToTitleCase Transform(System.String)

Figure 3: (Left): transformation for names. The first three values in column-D are provided as output examples. The desired first-names and last-names are marked in bold for ease of reading. A composed program using library CSharpNameParser from GitHub is returned. (Right): transformations for addresses. The first two values are provided as output examples to produce city, state, and zip-code. Note that some of these info are missing from the input. A program invoking Bing Maps API is returned as the top result.

entered. For instance, in the first column dates are represented in many different formats. In the second column, some customer names have first-name followed by last-name, while others are last-name followed by comma, then first-name, with various salutations (Mr., Dr., etc.) and suffixes (II, Jr., etc.). Phone numbers are also inconsistent, with various international calling codes (+1) and extensions (ext 2001), etc. Addresses in the last column are also not clean, often with missing state and zip-code information.

This data in Figure 1 is clearly not ready for analysis yet – an analyst wanting to figure out which day-of-the-week has the most sales, for instance, cannot find it out by executing a SQL query: the date column needs to be transformed to day-of-the-week first, which however is non-trivial even for programmers. Similarly the analyst may want to analyze sales by area-code (which can be extracted from phone-numbers), or by zip-code (from addresses), both of which again require non-trivial data transformations.

In a separate scenario, suppose one would like identified possible duplicate customer records in Figure 1, by first standardizing customer names into a format with only last and first names (e.g., both the first two records will convert into "Doe, John"). This again requires complex transformations.

Data transformation is clearly difficult. However, our observation is that these domain-specific transformation problems like name parsing and address standardization are really not new – for decades developers have built custom code libraries to solve them in a variety of domains, and shared their code in places like GitHub. In a recent crawl, we obtained over 1.8M functions extracted from code libraries crawled from GitHub, and over 2M code snippets from StackOverflow, some of which specifically written to handle data transformations in a variety of domains.

Transform-Data-by-Example. The overarching goal of the project is to build a search engine for end-users to easily reuse code for transformations from existing sources. Specifically, we adopt the by-example paradigm and build a production-quality system called *Transform-Data-by-Example* (*TDE*). The front-end of *TDE* is an Excel add-in, currently in beta and available from Office Store [7]. From the Excel add-in, users can find transformations by providing a few input/output examples. In Figure 2(left), a user provides two output examples to specify the desired output. Once she clicks on the "Get Suggestions" button, the front-end talks to the *TDE* back-end service running on Microsoft Azure cloud, which searches over thousands of indexed functions, to on-the-fly *synthesize* new programs consistent with all examples. In the right part of Figure 2, a ranked list of programs are returned based on *program complexity*. The top-ranked program uses the `System.DateTime.Parse()` function from the .Net system library to generate correct output for all input. Figure 3 shows additional examples for transforming names and addresses using the data in Figure 1.

TDE has a number of unique features, which we believe are important first steps towards realizing *self-service data transformation*.

• *Search-by-Example. TDE* works like a search engine, which allows end-users to search transformations by just a few examples, a paradigm known as program-by-example (PBE) [23] that was also used by FlashFill [16] for data transformation with much success. Compared to existing PBE systems such as FlashFill that compose a small number of string primitives predefined in a Domain Specific Language (DSL), *TDE* synthesizes programs from a much larger search space (tens of thousands of functions). We develop novel algorithms to make it possible at an interactive speed.

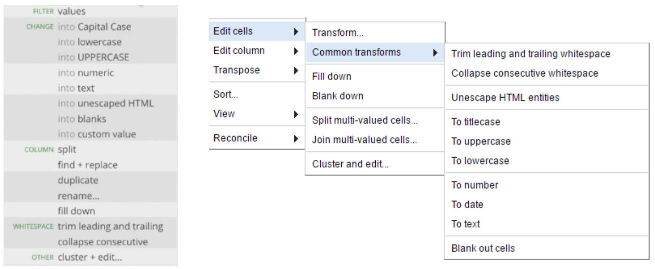


Figure 4: Example menu-based transformation. (Left): Paxata. (Right): OpenRefine.

• *Program Synthesis on-the-fly.* Since existing functions in code libraries rarely produce the exact output required by users, *TDE* automatically synthesize new programs, by *composing* functions, web services, mapping-tables [28], and syntactic string transformations from a DSL, all within a few seconds. Expert-users have the option to inspect the synthesized programs to ensure correctness.

• *Head-domain Support.* We build an instance of *TDE* that indexes over 50K functions from GitHub and 16K mapping tables, which can already handle many head domains (e.g., date-time, person-name, phone-number, us-address, url, unit-conversion, etc.) that are not supported by any existing system.

• *Extensibility.* Although *TDE* can already handle many important domains, there will be diverse application domains where *TDE* will not support out of the box, as it would not have encountered and crawled functions in those domains. *TDE* is designed to be *extensible* – users can simply point *TDE* to their domain-specific or enterprise-specific source code, DLLs, web services, and mapping tables, the transformation logic in these resources will be automatically extracted, and made immediately search-able. The way *TDE* works is just like a search engine "indexing" new documents.

To sum up, we have built a self-service, by-example data transformation engine *TDE*, leveraging unique technologies such as function analysis and ranking. We will first give an overview of existing approaches to data transformation.

2. REVIEW OF EXISTING SOLUTIONS

There have been significant activities in the industry in the space of self-service data preparation in recent years. A Gartner report [25] reviews 36 relevant systems from startups (e.g., Paxata [4] and Trifacta [8]), as well as established vendors (e.g., Informatica Rev [2]). We discuss a few representative approaches here to set our discussion in context.

Menu-based transformation. Most existing data preparation systems have menus for commonly used transformations. Users are expected to find appropriate transformations from the menus. Figure 4 shows the menus of Paxata [4] and OpenRefine [3] for instance. As can be seen, only a limited number of simple transformations are supported.

Language-based transformation. Existing systems also define their own "transformation languages" that users can learn to write. For example, Trifacta [8] defines a language called Trifacta Wrangle language, and OpenRefine [3] has its own OpenRefine Expression Language. Like other domain specific languages (DSL), these tend to have steep learning curves and limited expressiveness [20, 23].

Transformation by input. Trifacta adopts an interesting variant of PBE in which users can select portions of input data, and the system will suggest possible transformations (termed predictive interaction [18]). Similarly, Informatica Rev [2] and Talend [6] can also suggest operations based on selected input. However, input alone often cannot fully specify the desired transformation – for example, for a datetime input, users' intent can be anything from adding X number of hours (for timezone conversion), to changing into one of many possible formats. The input-only approaches have no clue which output is intended and often produce long lists of irrelevant suggestions.

Example-driven search using string primitives. Although program-by-example (PBE) is a known paradigm in the programming language literature for decades [23], FlashFill [16] pioneered its use for tabular data transformation. In FlashFill-like systems [20, 22, 26], users provide input/output examples, and the system uses simple *string primitives* to find consistent programs. While the PBE paradigm significantly improves ease-of-use (which *TDE* also builds upon), the expressiveness of existing systems is limited with no support of complex transformations requiring domain-specific knowledge (e.g., Figure 2 and Figure 3).

Example-driven search with search engines. DataXFormer [9] uses search engines to find relevant web tables and web forms, from which desired output values can be extracted (note that unlike PBE that only use examples, column headers are required here as input to build keyword queries). While web forms/tables are important resources, DataXFormer lacks ability to synthesize results to handle complex transformations.

Type-based transformation. Systems such as Informatica Rev [2] can suggest common type-specific transformations based on data types; so can research prototype AutoType [30], which like *TDE* also leverages open-source code. In comparison, *TDE* does not require a priori specifications of data types; instead it leverages unique fuzzy function ranking to match relevant functions against tasks, which makes it much more flexible and broadly applicable.

3. SYSTEM ARCHITECTURE

A high-level overview of *TDE* architecture is shown in Figure 5. Like a typical search engine, there are two main phases. First, in an *offline phase*, *TDE* collects, analyzes, restructures and indexes useful resources from (1) code libraries, (2) web service APIs, and (3) mapping tables. Next, in the *online phase*, when a user submits a transformation task in Excel, *TDE* uses the index built offline to quickly find relevant transformation logic, which are then used to synthesize programs matching all input/output examples.

The offline phase. In this phase, we collect around 12K C# repositories from GitHub, and extract a total of 1.8M functions. We also index all functions in the .Net system library, and 16K mapping relationships from a variant of [28]. Note that although we currently focus on C# code, the code-analysis techniques here can be easily applied to other languages like Java and Python.

Given the large repository of code, a key technical challenge is to "understand" these functions, so that at run time we can quickly find ones relevant to a new transformation task. We perform extensive offline analysis of functions, using code-analysis as well as distribution-analysis from executions, so that we can build input/output profiles for these functions. Section 4 gives details of this step.

Mapping relationships (e.g., "Washington" to "WA"; or "SFO" to "San Francisco International Airport") work like dictionary lookup and are important resources that can complement code. We start with a large crawl of HTML tables

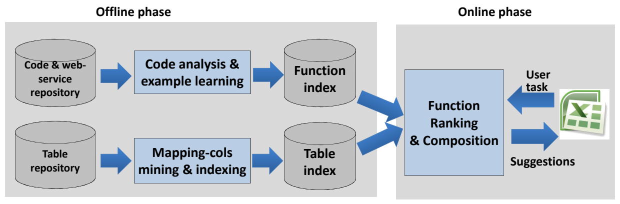

Offline phase
Online phase
Code & web-service repository
Code analysis & example learning
Function index
Table repository
Mapping-cols mining & indexing
Table index
Function Ranking & Composition
User task
Suggestions

Figure 5: Architecture of *TDE* back-end.

from Bing's index [12], and derive mapping-relationships from these tables [28]. These mappings are used holistically with code to synthesize programs (Section 7).

For web services, we use service URLs identified from Bing's index, as well as ones manually curated (e.g., Bing Maps API [1]). These services are necessary for a class of complex transformations that require extensive external resources for which standalone code libraries are often insufficient (e.g., produce missing zip-code for addresses in Figure 3). Web service APIs are handled by *TDE* just like functions but are invoked slightly differently (Section 6).

The online phase. In this phase, given a user transformation task, we need to (1) quickly identify functions, web services and tables that may be relevant; and (2) synthesize new programs that leverage a combination of these resources. This process needs to be highly efficient to ensure interactivity (i.e., in a few seconds). We design novel ranking and synthesis algorithms that can efficiently compose transformation logic into complex programs (Section 5).

We note that end-users may not be able to inspect programs for correctness, and instead often rely on inspecting a small set of output for quality checks. Additional techniques to profile output results, and alert users of potential errors (e.g., [19]), are interesting directions of research for not only TDE but also the PBE paradigm in general.

4. OFFLINE FUNCTION INDEXING

Given a large repository of code, a key technical challenge is to figure out what functions may be relevant for a given transformation task, because blindly testing all functions crawled will be impossibly slow. We perform extensive analysis of black-box functions to profile and index "typical" input parameters that each function can take. We will describe two main groups of techniques: (1) code-based analysis, and (2) entropy-based distribution analysis.

4.1 Code Analysis Techniques

For code analysis, we use the Roslyn compilation framework [5] to compile source code and generate *abstract syntax tree (AST)*[3], which is an intermediate representation widely used by the PL community. We use Roslyn to programmatically compile all code projects from GitHub (using .sln files); and StackOverflow snippets (we create a new project for each code snippet, appropriately padded with common dependencies.) in order to generate AST.

4.1.1 Static Analysis & Code Restructuring

To figure out what input values each function can typical take as input (so that at runtime we can quickly find relevant functions), one approach is to use static analysis techniques such as *constant-propagation*. For example, given the code snippet in Figure 6, it is not hard to figure out that value "`Tuesday, August 25, 2015`" is a valid parameter for `System.DateTime.Parse()`, since it propagates down and

[3]This is generally applicable to other programming languages.

```
using System;
// ...
static bool processDateTime()
{
    string date = @"Tuesday, August 25, 2015";
    DateTime d1 = DateTime.Parse(date);
    int year1 = d1.Year;
    bool isLeapYear = DateTime.IsLeapYear(year1);
    return isLeapYear;
}
```

Figure 6: An example function from StackOverflow.

```
static bool processDateTime_1(string para_1)
{
    string date = para_1;
    DateTime d1 = DateTime.Parse(date);
    int year1 = d1.Year;
    bool isLeapYear = DateTime.IsLeapYear(year1);
    return isLeapYear;
}
```

Figure 7: A restructured function from Figure 6.

becomes a parameter of that function. We populate this pair of function/parameter-value in a function-to-example index, conceptually shown as a table in Figure 8. Note that by using AST, names of functions are fully resolved. Also only one input parameter is populated in the table since this function is unary. Applying this technique to all code projects crawled allows us to identify other parameter values used to invoke `System.DateTime.Parse()`, shown in the second and third row of the table.

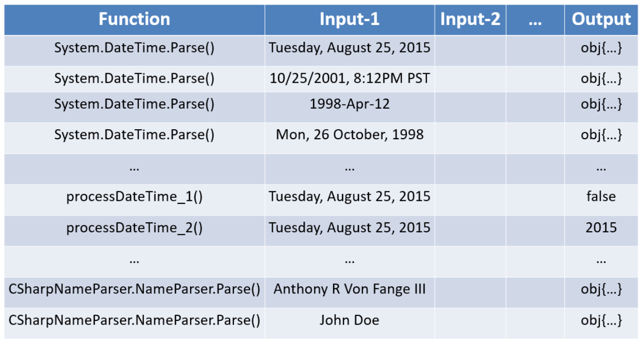
The approach of constant propagation is particularly effective for commonly used libraries (e.g., .Net standard functions), as well as custom code with unit-tests. Unit tests are a common engineering practice to ensure functions work as expected, and their parameters are often hard-coded constants, which this approach can leverage to collect input parameter appropriate for each function.

Code restructuring. We observe that crawled code can often be restructured to create a rich variety of new functions useful for data transformations. Specifically, we leverage AST to perform two types of restructuring. The first is known as *constant lifting*. For example, the function in Figure 6 is a parameter-less function. We can lift the constant value to the top as a parameter and create a new function in Figure 7 (using AST and Roselyn). This new function can now be invoked and we know the string "`Tuesday, August 25, 2015`" is a good parameter, which we populate as a new entry `processDateTime_1()` in Figure 8.

Observing that intermediate states of a complex function can often be useful, our second restructuring approach is to decompose functions into smaller units using dependency analysis. For the function in Figure 7, we can create a new function `processDateTime_2()` that returns the intermediate variable "`int year1`" by removing all trailing statements, as well as statements before it that the return statement does not depend on (which in this case is empty). This newly created function can again be populated in Figure 8. This restructuring approach creates rich functions from monolithic ones that can be useful for transformations.

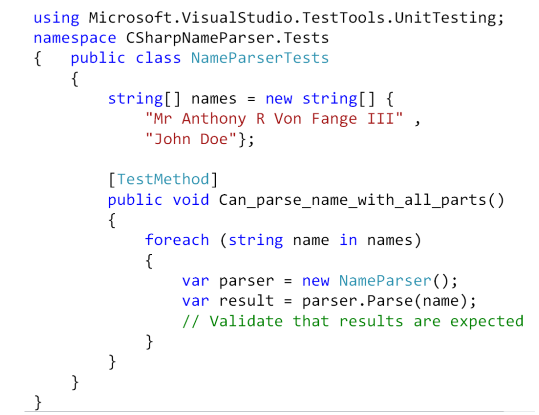
4.1.2 Dynamic Execution-based Analysis

In addition to static analysis, there are many cases where parameter values are only known during execution. Figure 9 shows a simple example, which is a unit test for function `CSharpNameParser.Parse()` on GitHub. Simple static analysis techniques like constant propagation would not be able to identify parameters for this function.



Function	Input-1	Input-2	…	Output
System.DateTime.Parse()	Tuesday, August 25, 2015			obj{…}
System.DateTime.Parse()	10/25/2001, 8:12PM PST			obj{…}
System.DateTime.Parse()	1998-Apr-12			obj{…}
System.DateTime.Parse()	Mon, 26 October, 1998			obj{…}
…	…			…
processDateTime_1()	Tuesday, August 25, 2015			false
processDateTime_2()	Tuesday, August 25, 2015			2015
…	…			…
CSharpNameParser.NameParser.Parse()	Anthony R Von Fange III			obj{…}
CSharpNameParser.NameParser.Parse()	John Doe			obj{…}

Figure 8: An example function-to-example index.



```
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace CSharpNameParser.Tests
{   public class NameParserTests
    {
        string[] names = new string[] {
            "Mr Anthony R Von Fange III" ,
            "John Doe"};

        [TestMethod]
        public void Can_parse_name_with_all_parts()
        {
            foreach (string name in names)
            {
                var parser = new NameParser();
                var result = parser.Parse(name);
                // Validate that results are expected
            }
        }
    }
}
```

Figure 9: An example function for dynamic analysis from CSharpNameParser on GitHub.

We instead use dynamic execution to profile parameter passed into each function during execution. Specifically, we leverage the AST to modify each function f, by inserting a "printf" statement right at the beginning of f so that whenever f is invoked its parameters will be recorded.

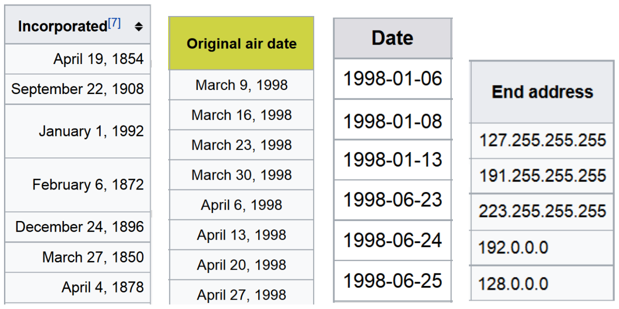
We then compile the modified source code, and use a number of ways to find functions we can execute, in the hope that they will in turn invoke other functions in the project to cover a substantial part of the code, and thus producing useful parameters for many functions. For instance, we programmatically execute all `Main()` functions; all static functions with no parameters; parameter-less functions in a class with constructors that we can invoke; as well as unit-test functions in popular testing frameworks (XUnit, NUnit, VS Unit Testing, etc.).

In Figure 9, by programmatically invoking all unit-tests, we can obtain two additional parameters for the function `CSharpNameParser.NameParser.Parse()`, shown in Figure 8.

Using both static and dynamic code analysis techniques discussed above, we are able to extract parameter examples for 214K functions out of 1.8M candidates (12%). After additional validation (e.g., checking whether a function can be compiled and programmatically invoked, whether a function has too many parameters, etc.), we arrive at around 45K invocable functions that are associated with parameters.

4.2 Entropy-based Distribution Analysis

The aforementioned techniques are *in-vivo* since they leverage existing data (both statically encoded and dynamically generated) in code. While useful, their coverage can be limited by the way code is written. For example, if a large code base comes with no unit tests or constants, then code-analysis is unlikely to handle it well. We propose a novel



Incorporated[7]
April 19, 1854
September 22, 1908
January 1, 1992
February 6, 1872
December 24, 1896
March 27, 1850
April 4, 1878

Original air date
March 9, 1998
March 16, 1998
March 23, 1998
March 30, 1998
April 6, 1998
April 13, 1998
April 20, 1998
April 27, 1998

Date
1998-01-06
1998-01-08
1998-01-13
1998-06-23
1998-06-24
1998-06-25

End address
127.255.255.255
191.255.255.255
223.255.255.255
192.0.0.0
128.0.0.0

Figure 10: Example Wikipedia table columns.

in-vitro method that leverages information theory to understand black-box functions, which to our knowledge is new in the database and programming language literature.

The high level idea is that we first generate a rich set of "representative" examples, denoted by $\mathbf{E}$, that cover a wide range of data types of interest (e.g., date-time, names, ip addresses, phone, locations, urls, etc.). We then pass each value in $\mathbf{E}$ as parameters to each function f, and by observing how f behaves in terms of its output distribution, we can infer "good" parameters suitable for f (for example, if f returns null for all of $\mathbf{E}$ except a small subset representing phone-numbers, then we know these are likely good parameters for f). We will describe these two steps in turn.

4.2.1 *Generating representative examples*

First, we would like to generate a large variety of parameter values $\mathbf{E}$. For this we resort to over 11 million relational table columns extracted from Wikipedia. With simple pruning such as removing long text-columns, we obtain structured values covering rich types of data. Figure 10 shows a few example table columns. A naive approach is to use all these values as parameters to execute all candidate functions, but the challenge is efficiency – the cross-product of the two would translate to trillions of function executions, which is prohibitively expensive.

Our observation here is that we only need to find "representative" values to "cover" the space of data, without needing to repeatedly test very similar values. For instance, in Figure 10, values in same columns follow similar patterns, and it is sufficient to pick one representative value from each column. Furthermore, for columns with very similar patterns like the first two columns in Figure 10, we can again pick only one value to "represent" both. On the other hand, for values of sufficiently different patterns, such as the first and the third column, we need to make sure both types of values are selected to cover different date formats.

This calls for a way to quantify "distance" between the patterns of two strings, and intuitively we want low pattern-distance (high similarity) between values like `April 19, 1854` and `September 22, 1908` despite a high edit-distance between them. We introduce a pattern-level distance measure inspired by sequence alignment [14]. Our first observation is that punctuation often serves as structural delimiters for rich data types, so we generate punctuation-induced sequences by "segmenting" strings using punctuation.

Definition 1. Given a string v, define s_v be its punctuation-induced sequence, where each element is either a single punctuation, or a maximal substring with no punctuation.

EXAMPLE 1. *The punctuation-induced sequence for* `April 19, 1854` *is* { "`April`", " ", "`19`", "`,`", " ", "`1854`" }; *while* *for string* `September 22, 1908`*, it is* { *"*`September`*", " ", "*`22`*", "*`,`*", " ", "*`1908`*"* }.

Given two sequences s_u and s_v, we define their pattern-level distance inspired by sequence alignment. Recall that in general sequence alignment [14], the distance between two sequences s_u and s_v is computed by padding both sequences with special gap-elements denoted as '-', to produce sequences $\overline{s}_u$ and $\overline{s}_v$ with the same number of elements so that they can be "aligned", and the sum of their element-wise distance $\sum_{i \in [|\overline{s}_u|]} d\left(\overline{s}_u[i], \overline{s}_v[i]\right)$ is minimized over all such $\overline{s}_u$ and $\overline{s}_v$ (where $\overline{s}[i]$ denotes the i-th element of sequence $\overline{s}$).

This alignment-based distance framework is general and has been applied to different scenarios, such as for gene sequences where element-wise distance are defined between pairs of symobols in {`A, T, C, G, -`}; as well as for the edit-distance between strings where a unit cost is used between different characters. We adopt this sequence-alignment framework to define our pattern-distance, but define different element-wise distances based on character classes. Specifically, we consider three classes of characters in the English alphabet: the letters, the digits, and the punctuation $\mathcal{P}$ (which is defined to include everything else). We define distance between a pair of elements as follows.

Definition 2. The distance $d(e, e')$ between two elements e, e' is defined as:

$$d\left(e, e'\right) = \begin{cases} 1, & \text{if } e = \text{'-' or } e' = \text{'-'} \\ 0, & \text{if } e, e' \in \mathcal{P},\, e = e' \\ 1, & \text{if } e, e' \in \mathcal{P},\, e \neq e' \\ 1, & \text{if } e \in \mathcal{P},\, e' \notin \mathcal{P} \\ 1 - \frac{\min(n_l(e), n_l(e')) + \min(n_d(e), n_d(e'))}{\max(n(e), n(e'))}, & \text{if } e, e' \notin \mathcal{P} \end{cases} \tag{1}$$

Where '-' denotes the special gap character inserted; $n(e)$, $n_l(e)$, and $n_d(e)$ denote the total number of characters, letters and digits in e, respectively, for which $n(e) = n_l(e) + n_d(e)$ always holds when $e \notin \mathcal{P}$.

These five cases in the definition can be explained as follows: (1) if e or e' is the special gap element '-' then their distance is 1 (similar to other sequence-alignment distances); (2) if e and e' are the same punctuation then their distance is clearly 0; (3) if e and e' are different punctuation their distance is 1 (because different punctuation are often very different structural delimiters); (4) if e is punctuation and e' is not their distance is 1 (one is data content and the other is a delimiter); (5) if neither e nor e' is punctuation, then intuitively their "pattern similarity" can be defined as the number of shared characters in the same class (letters and digits), divided by the max length of the two strings, which is in the range of [0, 1]. We define their distance simply as 1 minus this similarity score.

The definition above ensures element-wise distance $d\left(e, e'\right) \in [0, 1]$. We can then define *pattern distance* of two sequences as their average element-wise distances, similar to other sequence alignment distances.

Definition 3. The pattern distance $d(s_u, s_v)$ between sequence s_u and s_v, is defined as

$$d(s_u, s_v) = \min_{\overline{s}_u, \overline{s}_v} avg_{i \in [|\overline{s}_u|]} d\left(\overline{s}_u[i], \overline{s}_v[i]\right) \tag{2}$$

where $\overline{s}_u$ and $\overline{s}_v$ are sequences padded from s_u and s_v, with the same number of elements.

This pattern-distance can be computed similar to sequence alignment using dynamic programming [14].

Example 2. *The pattern-distance between* `April 19, 1854` *and* `September 22, 1908`*, can be computed from the best alignment between sequences* { *"*`April`*", " ", "*`19`*", "*`,`*", " ", "*`1854`*"* }*, and* { *"*`September`*", " ", "*`22`*", "*`,`*", " ", "*`1908`*"* }*. The trivial alignment without inserting '-' produces the lowest distance with the following 6 element-wise distances:* $1 - \frac{\min(5,9)+\min(0,0)}{\max(5,9)} = 0.44$ *(between "*`April`*" and "*`September`*"), 0, 0, 0, 0, 0. The overall sequence distance is thus* $\frac{0.44}{6} = 0.075$*, which is small, indicating that these two strings are similar patterns.*

Similarly, it can be verified that for value `April 19, 1854` *(sequence* { *"*`April`*", " ", "*`19`*", "*`,`*", " ", "*`1854`*"* }*) and* `1998/01/06` *(sequence* { *"*`1998`*", "*`/`*", "*`01`*", "*`/`*", "*`06`*"* }*), the best alignment has these element-wise distances:* $1 - \frac{\min(4,0)+\min(0,5)}{\max(4,5)} = 1$ *(between "*`1998`*" and "*`April`*"); 1 (between " " and "*`/`*"); 0 (between "*`19`*" and "*`01`*"); 1 (between "*`,`*" and "*`/`*"); 1 (between " " and "–", where "–" is a special gap inserted), and finally* $1 - \frac{\min(2,4)+\min(0,0)}{\max(2,4)} = 0.5$ *(between "*`1854`*" and "*`16`*"). The overall distance is* $\frac{4.5}{6} = 0.75$*, indicating that the two strings have very different patterns.*

Using this pattern-level distance, we can compute that strings within the same columns in Figure 10 all have low distances and are thus "similar", so we can pick one string to represent each column. The same is true for strings between the first two columns, but not for strings between other columns.

With this distance, we use a bottom-up clustering to "de-dup" at the pattern-level across 11 millions input Wikipedia table columns, to produce a total of 3.5K clusters (a three orders of magnitude reduction). From each cluster we take 20 random examples to generate around 50K values (certain clusters have less than 20 values). This reduction makes it possible to use broad-based distribution-analysis, which requires executing all values against all functions. Figure 11 shows a small set of values retained in the process, which are manually determined to be related to date-time. Note that our pattern-distance allows rich variations of date-time formats to be preserved. We defer more details of this step to a long version of the paper in the interest of space.

Note that ideally we would like to use *semantic* labels of columns for de-duplication. Since such semantics are hard to obtain for all columns, we use syntactic pattern as a proxy. This approach is nevertheless effective for semantic values that often have distinctive characteristic (e.g., address lines).

4.2.2 Entropy-based distribution analysis

After obtaining representative examples denoted by $\mathbf{E}$, we use $\mathbf{E}$ as parameters to execute all candidate functions $\mathbf{F}$. The idea is that by observing the distribution of the outcome after executing $\mathbf{E}$ on $f \in \mathbf{F}$, we can infer $\mathbf{E}_f \subset \mathbf{E}$ that are suitable input for f. Specifically, there are two general scenarios.

S1: Function f strictly checks the validity of input parameters, such that if the input values are not of expected formats then f errors out (i.e., throws an exception, returns a null object, etc.). For instance, the `DateTime.Parse()` function will throw a format-error exception if the input is not an expected date-time string that it can handle. For such functions, since only a small fraction of suitable values

Cluster Size	Representative Value
45855	Mar 30, 1998
31368	11 February 1939
15627	August 18
5008	May 1919
4622	17 August
4266	2013/14
3360	1998-06-11
2764	16 December 1983 (aged 27)
2710	11/14/09
1175	08-03-309
952	1977 Apr 04
716	Aug. 1962
358	Fri, Oct 30
318	9 Jan 2010
317	Saturday, May 11, 2013
254	Sunday 28 March
210	November, 2011
307	27-Aug-83
306	MAR/12/1982
266	1963.09.29
191	09/13/1969*
174	November 14*
115	17 Sep, 1981
115	04-AUG-1936
107	Sun. Sep. 13
104	28/4/1993
1284	770 BC
285	Saturday, 28 Mar 2009

Figure 11: Representative values after clustering that are related to date-time, and the number of similar values in each cluster.

in **E** will execute, observing the outcome across all **E** makes $\mathbf{E}_f$ straightforward.

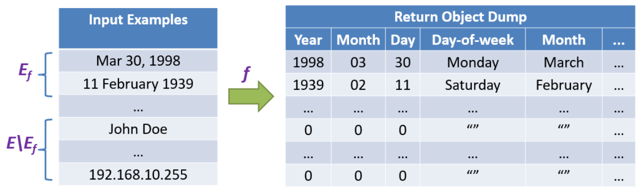
S2: Not all functions strictly check input for errors – many functions may not error out on inappropriate input and still return some results. However, an distribution-based analysis would still reveal $\mathbf{E}_f \subset \mathbf{E}$ that are suitable for f. Specifically, f typically performs some type of transformation for a specific kind of input data, and then populate the return object with appropriate results. For example, a GitHub function that parses date values returns a `date` object with attributes such as `int year`, `int month`, `int day`, `string day-of-the-week`, etc. When valid date strings (e.g., ones in Figure 11) are passed as parameters, these attribute values in return objects are properly populated as shown in Figure 12, with `year=1998`, `month=3`, etc. However, if input values are not date-time but are other strings, the function will likely terminate differently, and attribute values in the return object will have default initial values, in this case 0 for `year, month, day`, and empty string for `day-of-the-week`.

Intuitively after executing **E** on f we may see a large cluster of (likely "default") values returned, and then a small group of diverse values (likely meaningful output for f). Note that this divide between a big group of identical results (likely meaningless) and a small group of diverse results (likely meaningful) actually applies consistently to both **S1** and **S2** – in **S1** we can just view exceptions also as output.

Since these desired distributions correspond to low-entropy distributions in information theory, we use entropy to identify such functions f and their corresponding $\mathbf{E}_f$. Recall that *entropy* of a distribution X is defined as

$$H(X) = -\sum_{i}^{n} P(x_i)\log P(x_i) \tag{3}$$

where $P(x_i)$ is the probability of the i-th outcome x_i in X. Since entropy will naturally grow larger for variables



Input Examples
Mar 30, 1998
11 February 1939
...
John Doe
...
192.168.10.255

Return Object Dump					
Year	Month	Day	Day-of-week	Month	...
1998	03	30	Monday	March	...
1939	02	11	Saturday	February	...
...	...	...	...	...	...
0	0	0	""	""	...
...	...	...	...	...	...
0	0	0	""	""	...

Figure 12: Example distribution analysis to infer examples $\mathbf{E}_f$ suitable for function f.

with larger domains (e.g., `string` vs. `bool`), we apply normalization to make it domain-insensitive, using *normalized entropy* [21] defined as

$$N(X) = -\frac{\sum_i^n P(x_i)\log P(x_i)}{\log n}$$

A small $N(X)$ suggests a highly skewed distribution, from which $\mathbf{E}_f$ can be identified accordingly.

Example 3. *In the example discussed in Figure 12, recall that we have roughly 50K representative examples selected from the previous step as* **E**. *Around 30 of these values are date-time strings (in Figure 11), which are effectively* $\mathbf{E}_f$ *for this function* f*, whose output will be populated with diverse attribute values as illustrated in the top rows of Figure 12. The remaining output will be default initial values, shown at bottom. Suppose we take the attribute* `year` *for analysis, then the normalized entropy can be computed as:* $(-\frac{50K-30}{50K}\log\frac{50K-30}{50K} - 30 \times \frac{1}{50K}\log\frac{1}{50K})/(\log 31) \approx 0.0002$. *From this we conclude that the distribution is highly skewed, and we can identify the small number of* $\mathbf{E}_f$ *accordingly.*

This entropy-based approach is applicable to a wide variety of functions, such as domain-specific functions that process specific data types (ip-addresses, phone-numbers, etc.), as well as general numeric functions such as a factorial function (which only accepts non-negative integers). However, this approach is not effective for functions that perform syntactic transformations that can accept virtually any input (e.g. an upper-case function), because the key underlying assumption that $\mathbf{E}_f$ is a small subset of **E** no longer holds.

Once we infer $\mathbf{E}_f$ for a given function f, these $\mathbf{E}_f$ can be used to populate the index table in Figure 8, which will in turn be used for function ranking, to be discussed next.

From the 1.8M input functions, we identify 62K ones that can be invoked with one parameter, with which we perform the entropy-analysis above. Around 5K (8.1%) functions can accept a small fraction ($\leq 5\%$) of the input examples, which are roughly functions that perform strict data checking.[4] On the other hand, around 40K functions can accept almost all input ($> 95\%$), from which we identify 5K functions with low normalized-entropy. In total we collect parameter examples for around 10K functions using this method.

Combining these with the functions from code analysis (Section 4.1), we are able to index around 50K functions.

5. ONLINE PROGRAM SYNTHESIS

Using techniques in Section 4, offline we can associate each function f with example parameters $\mathbf{E}_f$ suitable for f. In this section, we discuss the following key steps in the online phase: (1) given a user task T with input/output examples, rank all candidate functions **F** to find relevant one

[4]There are 15K (24.2%) functions that cannot accept any input. Further testing using additional examples may help.

to execute; (2) based on the execution result synthesize new programs to produce the target output; (3) re-rank among all synthesized programs that are consistent with T.

5.1 Function Ranking (L1)

Given a task T, naively we can execute all functions in $\mathbf{F}$ to test if they can be used to produce the desired output in T. However executing all of $\mathbf{F}$ is clearly too expensive and too slow for *TDE*. Specifically, the latency of executing one function can range from a few milliseconds to hundreds of milliseconds, so that even with parallelization a reasonable server can only execute up to a few hundred candidate functions within a few seconds. As such, we leverage the $f \leftrightarrow \mathbf{E}_f$ index (Figure 8) to quickly find a small set of promising functions for execution.

We would like to note that ranking discussed in this section is used internally by *TDE* only for selecting promising functions, which is different from the final ranking of synthesized programs that users see from UI (Figure 2). To differentiate rankers used in these two different stages, we use terminology similar to multistage ranking in search engines [31], and refer to this first-level rankers discussed here as *L1-ranking*, and refer to the final ranking of synthesized programs as *L2-ranking* (to be discussed in Section 5.3).

In *TDE* we currently employ two types of L1 function ranking, based on input patterns, and input-output relationships, respectively, which we will describe below. Note that like search engines, the ranking component is extensible by design to plug in other types of rankers.

5.1.1 L1 Ranking by input-patterns

Our first approach of L1 function ranking is based on comparing patterns of input parameters $\mathbf{E}_f$ accepted by function f, and that of input values of task T.

As an example, given the task T_{date} of standardizing date-time shown in Figure 2, intuitively we can see that the *pattern* of input values like `Wed, 12 Jan 2011` in T_{date}, matches pretty well with the one of the known parameter `Mon, 26 October, 1998` for function `System.DateTime.Parse()` in the index table in Figure 8, indicating that this function may be a good match for T_{date}.

We use the same pattern-distance in Definition 3 to measure how well input values in a task matches parameters of a function, and in this case can compute the distance of this value pair to be $\frac{1.57}{9} = 0.17$, which is small, suggesting that `System.DateTime.Parse()` may be a good match for T_{date}. It can be verified that this distance is considerably lower compared to distances with other functions in Figure 8.

Let $T.I$ be the input examples of a task T, the input-only ranking of a function f is $R(f,T) = \min_{v \in \mathbf{E}_f, u \in T.I} d(v,u)$. Since we use distance, a lower score indicates a better match.

We find this ranking approach works well for large varieties of semantic data values, such as ip-address, phone-number, email, date-time, postal-address, etc., most of which have structured patterns with fixed sequences of punctuation that the pattern-based ranking can recognize well.

5.1.2 L1 Ranking by input-output relationship

The first L1 ranker works well for functions whose input are semantic data values with distinctive patterns. However there is a class of functions that perform what we call syntactic transformations, such as functions for camel-casing/title-casing, or ones that trim white spaces or remove consecutive white spaces. For such functions, the input can be any string with no distinctive patterns.

For this reason, we also develop an L1 ranker exploiting the relationship between input and output. Specifically, using a standard grouping of characters in the English alphabet (e.g., lower/upper-case letters, letters, digits, alphanumeric, etc.), we "describe" the syntactic difference between input/output of a function f. As an example, for a function that trims white spaces, a succinct way to describe its input/output difference (observed from repeated executions of f with different parameters) is that white spaces are deleted from the input, while other classes of characters are unaffected (described as {delete: {" "}, insert: $\emptyset$, update: $\emptyset$}).

Then given a user task T, say title-casing, we can similarly describe its input/output difference succinctly as {delete: $\emptyset$, insert: $\emptyset$, update: {lower→upper}}. We can compare the description of T with that of each f, by computing an average similarity score across delete/insert/update categories, and rank functions f accordingly. For this T functions like upper-casing, camel-casing will rank high, while space-trimming functions will rank low.

The two L1 ranking methods are rather complementary (one focusing on functions processing semantic data while the other more suitable for syntactic functions), *TDE* takes the union of top-K functions from both rankers.

5.2 Synthesize Programs using Functions

Let R_K be the top-K functions returned by the L1 rankers, and T the transformation task. Recall that in *TDE*, a total of m (usually 3) input/output examples are provided in T. Let $T.I[i]$ be the i-th input row and $T.O[i]$ be the i-th output cell. The goal is to execute functions in R_K using $T.I[i]$ as input, to generate the target output $T.O[i]$ for all $0 \leq i \leq m$.

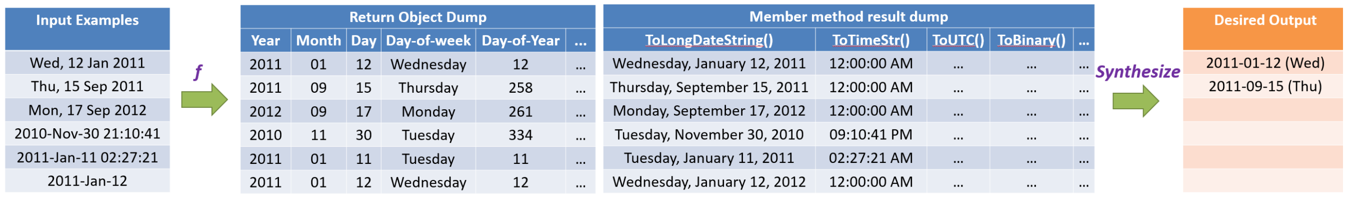
We first discuss program synthesis with single functions, and then parameterized multi-function synthesis.

5.2.1 Program synthesis with single functions

We first consider executing $f \in R_K$ using $T.I[i]$ as input, to produce target $T.O[i]$, $\forall i \in [m]$. Note that the target $T.O[i]$ are strings, and while the output of function f can be strings, in general functions return objects for object-oriented languages such as C# code indexed by *TDE*. As a result, even if we have right functions f for T, it is unlikely that the output of f can exactly match target output. The challenge herein is to automatically synthesize programs to convert result objects into target output strings.

Consider the task T_{date} in Figure 2 again. Since the function `System.DateTime.Parse()` is ranked high for T_{date}, we will execute it using input in $T_{date}.I[i]$ as parameters, which would result in `DateTime` objects. We use a programming language mechanism called *reflection* [27] (available in many programming languages), which allows us to programmatically "open" each live object and iterate through its member properties and methods at run time. Specifically, as Figure 13 shows, we are able to use this reflection to "dump out" values of all member properties of each returned `DateTime` object, which includes a few dozen properties[5] such as `Year`, `Month`, etc. Furthermore, we can use reflection to enumerate member methods in the `DateTime` objects, such as `ToLongDateString()` and `ToUTC()`. Since these are parameter-less methods, we can again invoke from each returned `DateTime` object to produce even more rich intermediate results as shown in the figure.

[5] `https://msdn.microsoft.com/en-us/library/system.datetime.aspx`



Input Examples
Wed, 12 Jan 2011
Thu, 15 Sep 2011
Mon, 17 Sep 2012
2010-Nov-30 21:10:41
2011-Jan-11 02:27:21
2011-Jan-12

Return Object Dump					
Year	Month	Day	Day-of-week	Day-of-Year	...
2011	01	12	Wednesday	12	...
2011	09	15	Thursday	258	...
2012	09	17	Monday	261	...
2010	11	30	Tuesday	334	...
2011	01	11	Tuesday	11	...
2011	01	12	Wednesday	12	...

Member method result dump				
ToLongDateString()	ToTimeStr()	ToUTC()	ToBinary()	...
Wednesday, January 12, 2011	12:00:00 AM	...	...	...
Thursday, September 15, 2011	12:00:00 AM	...	...	...
Monday, September 17, 2012	12:00:00 AM	...	...	...
Tuesday, November 30, 2010	09:10:41 PM	...	...	...
Tuesday, January 11, 2011	02:27:21 AM	...	...	...
Wednesday, January 12, 2012	12:00:00 AM	...	...	...

Desired Output
2011-01-12 (Wed)
2011-09-15 (Thu)

Figure 13: Produce desired output from input in *TDE*: a function invocation followed by program synthesis.

Given the intermediate tables with rich semantic information derived from the input values, the task now is to "assembly" bits and pieces in them to produce target $T.O[i]$. We illustrate it with an example below.

Example 4. *In Figure 13, given that the target output is* `2011-01-12 (Wed)` *and* `2011-09-15 (Thu)`*, it can be seen that they can be produced by concatenating relevant fields from the intermediate tables. Specifically, if we concatenate the* `Year` *field with a "*`-`*", then append with the* `Month` *field followed by a "*`-`*", then with the* `Day` *field followed by a "* `(`*", then with the first three characters of the* `Day-of-week` *field, and finally append with a closing parenthesis "*`)`*". It can be verified that this synthesized program produce the desired target output for both input strings.*

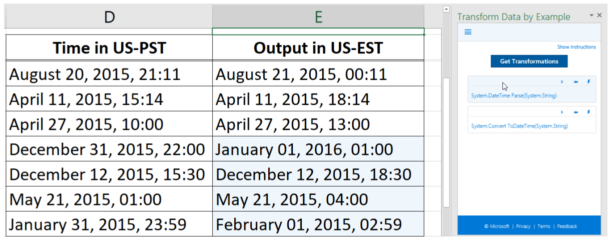
Suppose the desired output is instead `2011-Jan-12 (Wed)` *and* `2011-Sep-15 (Thu)`*. Note that the required months are now* `Jan` *and* `Sep`*, which cannot be produced from the* `Month` *column. For this we take the column corresponding to the output of method* `ToLongDateString()`*, and perform the following operations: We split each value using "*`,`*", and take the second component from the split (the substring after the first comma), from which we take a substring of length 3 starting at the second character. This would produce the desired* `Jan` *and* `Sep`*; all other operations in this synthesized program will remain the same as the previous example.*

This example shows the power of synthesis using intermediate results from member properties and methods – by being able to synthesize multi-step sequences, we produce powerful and expressive programs to match user output.

We note that similar techniques for generating string transformation programs in this step have been the focus of FlashFill-like PBE systems [16, 20, 26]. However, the requirement of *TDE* is unique, because the intermediate tables shown in Figure 13 (from which results are synthesized) can often be very "wide" with hundreds of columns for complex objects. Furthermore, the synthesis algorithm needs to be invoked for hundreds of times for *each* function returned by the L1 ranker. Given that *TDE* needs to be interactive, the synthesis algorithm is required to be highly efficient. In particular, we find existing approaches such as [16] insufficient for *TDE*. We develop new algorithms based on a recursive greedy search. A basic version of this synthesis algorithm was described in [32] (used for a different purpose, which is to auto-join tables). Compared to prior work, our synthesis is (1) substantially more efficient; and (2) provides probabilistic guarantees of success under certain assumptions ([32]). We defer details to a full version of the paper.

5.2.2 Parameter learning in multi-function synthesis

In the previous example, when executing a top-ranked function $f \in R_K$, we use reflection to not only consider all member properties, but also member methods that are parameter-less, since it is straightforward to execute them. However, there are also many *parameterized* member methods that are useful for transformations. For instance, consider the T_{time} shown in Figure 14, where the task is to



D	E
Time in US-PST	Output in US-EST
August 20, 2015, 21:11	August 21, 2015, 00:11
April 11, 2015, 15:14	April 11, 2015, 18:14
April 27, 2015, 10:00	April 27, 2015, 13:00
December 31, 2015, 22:00	January 01, 2016, 01:00
December 12, 2015, 15:30	December 12, 2015, 18:30
May 21, 2015, 01:00	May 21, 2015, 04:00
January 31, 2015, 23:59	February 01, 2015, 02:59

Transform Data by Example
Show Instructions
Get Transformations
© Microsoft | Privacy | Terms | Feedback

System.DateTime Parse(System.String)

System.Convert ToDateTime(System.String)

Figure 14: *TDE* transformation between timezones.

convert input time in US western timezone, to US eastern time. Note that this "+3 hours" operation can lead to a change in the day, month, and year, as shown in the figure.

This transformation would require not only using relevant methods but also appropriate parameters ("+3 hours"). *TDE* performs this transformation by synthesizing the following program: it first invokes `System.DateTime.Parse()` to convert each input string into a `DateTime` object, whose member method `DateTime.Add(Timespan)` is then invoked using a parameter of type `Timespan` corresponding to 3 hours. This leads to a new `DateTime` object, from which we can synthesize the target output as described in Section 5.2.1. The key challenge here is parameterization, or finding an appropriate `Timespan` object as parameter – exhaustive enumeration would not work as the parameter space is infinite.

For parameterization, in *TDE* we perform offline learning for relationships between functions in same classes, to discover concepts such as *inverse relationships*. Specifically, we first identify functions f_1 and f_2 as a candidate pair, if the result of f_1 is of the same type as the parameter of f_2. In the example above, in the class `DateTime` we have the function `TimeSpan DateTime.Subtract(DateTime)` that returns an object of type `TimeSpan`, and we also have function `DateTime DateTime.Add(Timespan)` taking a parameter of type `TimeSpan`. We thus treat the two as a candidate pair. We then instantiate pairs of `DateTime` objects o_1, o_2 (with suitable parameters obtained from indexes in Figure 8), and invoke o_1`.Subtract(`o_2`)` to produce a `TimeSpan` object t_{12}. To test if the inverse relationship holds, we then invoke o_1`.Add(`t_{12}`)` to produce o'_2, and see if o'_2 is identical to o_2. Since this holds true for all pairs of o_1, o_2 tested, we can infer that the two are *inverse* functions.

With the inverse relationship, given T_{time} at run time, we can use $T_{time}.I[i]$ as o_1 and $T_{time}.O[i]$ as o_2, and compute $t_{12} = o_2$`.Subtract(`o_1`)`, which turns out to be 3 hours consistently for all $i \in \{1, 2, 3\}$. We can thus produce a correct program with right parameters as described above.

Another type of parameterized functions we can invoke is the ones that have parameters with limited cardinality. For example, the function `DateTime.ToString(string format)` accepts a parameter with a limited number of formats (e.g., "`MM/dd/yyyy`", etc.). Using the index in Figure 8, if we determine a parameter of f to be of small cardinality, we treat it as an "`enum`" type and "memorize" all its possible values, which we then use to exhaustively invoke f. This allows us to invoke functions like `DateTime.ToString("MM/dd/yyyy")` to create more rich intermediate data from which *TDE* can use to synthesize target output.

5.3 Re-rank synthesized programs (L2)

Given a task T, since the program synthesis process leverages many functions in R_K, it may produce more than one program consistent with all given examples. In *TDE* we show all such programs to users in a ranked list (Figure 2).

In order to appropriately order synthesized programs for user inspection, we introduce another level of ranking, referred to as *L2 ranking.* Note that L2 ranking happens at a later stage and is different from *L1 ranking* (Section 5.1). In L2 ranking, we use execution information not available in L1 ranking to re-rank synthesized programs.

Our observation is that the *complexity* of a synthesized program is often a good indicator of how well it can generalize beyond the few examples in T (typically only 3). For instance, suppose the target output is day-of-the-week such as `Wed`, `Thu`, etc., for the input in Figure 13. A correctly synthesized program that leverages the function `DateTime.Parse()` can produce this target output by simply taking the first three characters of the `day-of-the-week` column in Figure 13. This program would overall (1) invoke an external function, and (2) synthesize a substring operation from the result, for a total complexity of 2.

In comparison, suppose there is a hypothetical function g that generates long random strings. *TDE* may also synthesize a program to produce `Wed` and `Thu`, by taking, say, the 25-th character of gs output (which happen to be `W` and `T`), appended with the 34-th character of g's output (say, `e` and `h`), and finally the 10-th character from g (`d` and `u`). This program is clearly a poor "fit" and cannot generalize well. We can infer this by observing its higher complexity (5), allowing *TDE* to rank this program lower.

Note that this re-ranking is consistent with general principles such as *minimum description length* or *Occam's Razor*.

6. INDEXING WEB SERVICES

So far we focus on leveraging program functions for data transformations. We will briefly describe two additional resources *TDE* utilizes: web services and mapping tables.

Web services provide rich functionalities that can be (1) lookup in deep-web databases; and (2) functional transformations. Existing work discover deep-web databases via deep-web crawling [17, 24], and discover transformation services via WSDL/SOAP [10], or search engines [9].

Since web services are conceptually the same as program functions (but execute remotely), they can also be used by *TDE*. Given that REST APIs on the Web are increasingly popular over WSDL, we focus on REST APIs. We classify a URL in Bing's large URL repository as a service URL, if it is parameterized, and the corresponding content is in XML or JSON (which unlike HTML are intended to be machine-readable). Table 1 shows a list of such services.

By comparing similar URLs (but with different parameters) from the same host, we can recognize the embedded parameters in URLs, and thus index them similar to functions in Table 8. The services are then treated by *TDE* just like functions for ranking. For program synthesis, we parse returned XML/JSON documents and dump value fields, which can then be used to synthesize target output just like program objects returned by functions.

We note that web services are critical for certain complex transformations. For instance, the task in Figure 3 (right) requires generating state and zip-code information that may be missing from the input. We find standalone functions to be insufficient, and only web services (in this case Bing Maps [1]) have sophisticated domain-specific logic/data to perform such transformations.

7. INDEXING MAPPING RELATIONSHIPS

Mapping tables like ones shown in Table 2 is another important resource for transformations, and is consistent with the PBE paradigm. We use a variant of [28] to synthesize mappings from tables and index around 16K in *TDE*.

One key distinction of *TDE* compared to systems such as InfoGather [29] and DataXFormer [9], is that *TDE* produces holistic programs that tightly integrate the logic of program-code and mappings. For instance, in the example of Figure 3 (right), the address-parsing service invoked only returns state abbreviations ("`WA`", "`TX`") but not state full names. Suppose a user enters state full names ("`Washington`", "`Texas`", etc.) as the desired output. *TDE* is able to produce a synthesized program that first invokes the web service to retrieve a JSON document, which is then parsed to extract state abbreviation of each input, from which the `(state-abbreviation→state)` mapping (in Table 2) is used to retrieve the desired ("`Washington`", "`Texas`") as output.

The way *TDE* synthesizes relevant mappings into programs can be summarized as follows. In each recursive step of synthesis (Section 5.2.1, with more details in Algorithm 2 of [32]), *TDE* is given input/output example pairs (I, O) with a changing O as partial progresses are made towards the target. In each step, *TDE* checks if the right-hand-side of any M contains O as a subset. Since this is a set-containment check, offline we use Bloom-filters [11] to index the right-hand-side of each mapping M. At run time, if *TDE* finds a hit M in any step, it update the task as $(I, M^{-1}(O))$ (note that M^{-1} may be multi-valued functions), and see if it can generate programs for any of the new tasks. In the example above, since the output examples are state full names, and *TDE* happens to have the `state-abbreviation→state` mapping that passes containment check, we can update the task's output as left-hand-side of the mapping (state-abbreviations). This new task can now be solved with program invoking a web service followed by a mapping relationship.

8. PRACTICAL CONSIDERATIONS

The public release of *TDE* as an Excel add-in requires addressing many practical issues not typically considered in research prototypes. We briefly discuss a few of them below.

Open-source software licensing. Open-source code are used freely in our research prototype to test the potential of *TDE*. For the public version of *TDE* released in Office Store, however, we had to exclude open-source code with restrictive licensing terms. We use logs collected in an internal release to identify code projects that are known to be useful for at least one real task. For each such project, we manually verify its license (often in License.txt or Readme.markdown for GitHub projects), and only include the ones with permissive licenses (e.g., Apache, and BSD). This leads to a substantial loss of coverage but is nevertheless necessary.

Extensibility. Although *TDE* can support transformations in important domains, there are also many tail domains

Table 1: **Example REST API for transformations (retrieved in July 2016).**

Sample url	Parameters	API description
https://www.google.com/finance/info?q=NASDAQ:MSFT	MSFT	quote and other financial info of a stock
http://openapi.ro/api/geoip/location/103.2.2.3.json	103.2.2.3	reverse ip lookup to find geo info of an ip
http://actorws.epa.gov/actorws/dsstox/v02/smiles.json?casrn=80-05-7	80-05-7	find chemical compound by cas
http://openapi.ro/api/exchange/usd.json?date=2011-02-05	2011-02-05	exchange rate by date

Table 2: **Example mapping relationships.**

Mapping	Example instances
(state-abbreviation, state)	(WA, Washington), (TX, Texas)
(country, country-code)	(Canada, CAN), (Japan, JPN)
(stock-ticker, company)	(MSFT, Microsoft), (ORCL, Oracle)
(model, make)	(Accord, Honda), (Mustang, Ford)

where *TDE* has no support out of the box due to the number of functions it currently indexes. *TDE* is thus designed to be *extensible* – users can point *TDE* to domain-specific code or web services, the transformation logic therein will be made immediately search-able.

Security and Privacy. To safeguard user data, we use .Net *application-domain*[6] (similar to a light-weight process) to isolate function execution. We clamp down application-domains by not allowing them to perform any file writes, or send messages over the internet (except for trusted web services). Additional security checks (both automatic and manual) were performed before the product release.

Poorly-behaving functions. Certain functions may behave badly on unexpected input, for example causing stack-overflow exceptions that can bring down worker processes. We prune out such functions based on execution statistics.

9. EXPERIMENTS

9.1 Experimental setup

Benchmark data sets. We compile a total of 239 data transformation tasks, which we release for future research[7]. These tasks are collected from the following sources.
(1) We identify popular queries of the pattern "convert A to B" in Bing's query logs (by frequency). We discard queries not related to data transformation (e.g., "convert jpg to png"), and manually build input/output examples for each remaining query. Noticing that a large fraction of the queries are for unit conversion (e.g., "convert farenheits to celsius"), we pick 50 most-frequent queries for unit conversion, and also 50 most-frequent queries that are not unit conversion (e.g., "convert EST to PST"), for a total of 100 tasks.
(2) We sample 49 popular questions from StackOverflow.com that are determined to be relevant to data transformations (e.g., "how to get domain name from URL").
(3) We select 46 transformation tasks used as demos in related systems (FlashFill, GoogleRefine, and Trifacta).
(4) We identify 44 common transformations tasks from a list of head domains (e.g., `ip, address, phone`, etc.), which we determine to be important that *TDE* has to perform well.

We note that the benchmark so generated covers diverse types of transformation tasks. A manually analysis classifies around 10 tasks as date-time-related, which is the single largest group by topic (4% of all cases). This is followed by person-names (7 tasks), and then addresses (6 tasks).

Most tasks have either 5 or 6 pairs of input/output examples. For all systems tested, we use 3 input/output pairs as "training" examples, and hold out the remaining ones for "testing", to validate if a synthesized program is correct. A program is predicted correct only if all its output match the ground truth (no partial score). We report average precision of each system, defined as the number of tasks solved divided by the total number of tasks.

Methods compared. We test with following systems.

FlashFill [16]. FlashFill is a pioneering PBE system for data transformation. We run FlashFill in Excel 2016.

Foofah [20]. Foofah is a recent PBE system that handles more general data transformation including table formatting not considered by *TDE*. We obtain their source code on GitHub and set timeout to 60 seconds as suggested in [20].

TDE. This is the research version of *TDE* that indexes around 50K functions from GitHub and StackOverflow. Note that the public version of *TDE* in Office Store indexes substantially less number of functions due to license issues.

TDE-NF. This is *TDE* with no functions and thus cannot handle complex semantic transformations. Nevertheless *TDE*-NF resembles PBE systems like FlashFill and solves transformations with string primitives predefined in DSL.

DataXFormer-UB [9]. Authors of [9] kindly responded, informing us that the system is no longer available for comparisons.[9] We therefore manually simulate how DataXFormer works for each test case. (1) For web services, we manually query Google with up to three keyword queries, e.g., "convert Fahrenheit to Celsius" (note that PBE systems like TDE use only examples with no such keyword information). We then click through top-40 results to identify web forms, where we manually fill in input/output examples and inspect results (in practice not all such forms can be automatically handled, and DataXFormer reports to successfully parse 12 out of 17 web forms tested (71%)). (2) For web tables, we simply test input/output examples against all (over 100M) web tables extracted from Bing's index and look for table-based matches. All of these favor DataXFormer and likely over-estimate its true coverage. We hence term this as *DataXFormer-UB* to represent the upper-bound of the search-engine based approach.

System-A. There are a number of commercial systems that can suggest transformations based only on input (see related work in Section 2). However, their EULA prohibits the publication of any benchmark comparisons. Following the tradition of benchmarking commercial database systems without revealing vendor names, we report anonymized results obtained from an unnamed product henceforth referred to as System-A, which is competitive in this group of products. We manually go through all suggestions produced by System-A and mark it as correct as long as one of its suggestion is correct irrespective of ranking.

OpenRefine-Menu [3]. We also compare with a version of OpenRefine (downloaded in March 2018) using its menu-based transformations.

9.2 Benchmark Evaluation

Quality. Table 3 shows precision results of all methods compared, broken down by categories. Overall, *TDE* produces desired programs for 72% of the cases, substantially better than other systems. This is not surprising, since *TDE* leverages the power of large varieties of domain-specific functions that are absent in other systems. *TDE* performs

[6]`https://en.wikipedia.org/wiki/Application_domain`
[7]`https://github.com/Yeye-He/Transform-Data-by-Example`

[9]DataXFormer was available at `http://dataxformer.org` [9].

Table 3: Precision of benchmark cases, reported as precentage of cases solved (number of cases in parenthesis).

System	Total cases (239)	FF-GR-Trifacta (46)	Head cases (44)	StackOverflow (49)	BingQL-Unit (50)	BingQL-Other (50)
TDE	**72% (173)**	**91% (42)**	**82% (36)**	**63% (31)**	**96% (48)**	**32% (16)**
TDE-NF	53% (128)	87% (40)	41% (18)	35% (17)	96% (48)	10% (5)
FlashFill	23% (56)	57% (26)	34% (15)	31% (15)	0% (0)	0% (0)
Foofah	3% (7)	9% (4)	2% (1)	4% (2)	0% (0)	0% (0)
DataXFormer-UB	38% (90)	7% (3)	36% (16)	35% (17)	62% (31)	**46% (23)**
System-A	13% (30)	52% (24)	2% (1)	10% (5)	0% (0)	0% (0)
OpenRefine-Menu[8]	4% (9)	13% (6)	2% (1)	4% (2)	0% (0)	0% (0)

reasonably well in all sub-categories except `BingQL-Other`, where the coverage is 36%. This category contains diverse transformations (e.g., conversion of color encoding, geo coordinates, etc.) that are difficult. We find the C# code crawled from GitHub lack many such functionalities, which however are often available in other languages (e.g., Python). Extending *TDE* with other languages would clearly help.

TDE-NF uses no external functions and can be considered as a traditional PBE system. Its overall result is reasonable, but it clearly falls short on cases requiring more complex transformations that are difficult to synthesize from scratch.

Both FlashFill and Foofah lag behind *TDE*/*TDE*-NF. We would like to note that while both FlashFill and *TDE* work in the same space of row-to-row transformation, which is exactly what our benchmark is designed to evaluate, the benchmark is unfavorable to Foofah, as it is more focused on orthogonal tasks such as table reformatting (e.g., pivot and un-pivot)[10]. Unifying Foofah-like capabilities with row-to-row transformation is interesting future work.

DataXFormer-UB solves 90 out of the 239 test cases (38%), showing the power of search engines and web services, which however is limited by the lack of program-synthesis. When nontrivial synthesis is required (e.g., output date-time in a specific format, or rounding numbers to a specific precision), vanilla web services can often fall short. In addition, We find that certain classes of transformations, such as names and date-time, are not typically handled by online web services.

System-A can handle 30 (13%) cases. We find System-A's approach the most effective when a test case requires extracting common sub-components from input. Such operations can be more easily predicted and are often solved correctly. However, there are many cases where selection alone is insufficient to fully specify the desired transformation (e.g., add 3 hours for time-zone conversion, switch the order of last/first name, etc.), which is an inherent shortcoming of predicting transformations using input only.

OpenRefine solves only 9 test cases (e.g., upper-casing) using built-in transformations from its menus. This is not entirely surprising, as the types of transformations supported by menu options are typically limited.

L1-Function-ranking. Recall that *TDE* uses L1-rankers (Section 5.1) to select a small set of promising functions from all functions its indexes, so that it can execute and synthesize them at an interactive speed. L1-ranking is a critical component for performance(the better we rank, the faster *TDE* can synthesize relevant programs).

Figure 15 evaluates the effectiveness of our two L1-rankers, where y-axis shows the percentage of cases that can be solved using only top-K functions from L1-rankers, and x-axis shows the number K, which affects response time. As we can see, the two L1-rankers are complementary, and their union is substantially better. Overall around 70% cases can be solved with top-200 functions, and that number goes up

[10]Despite the difference we evaluate Foofah as requested.

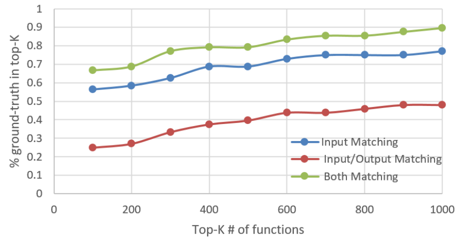


Figure 15: Effectiveness of ranking.

to 90% for top-1000 functions (which corresponds to a response time of around 5 seconds on our machine).

Efficiency. The average end-to-end latency to produce the first correct program (including function ranking, execution and synthesis) is 3.4 seconds, which is reasonably interactive. We note that *TDE* streams back results as they are found – once a worker finds a program it will show up on the right-pane for users to inspect.

9.3 Analysis of real usage logs

Since *TDE* is used by real Excel users, it provides an opportunity to understand how *TDE* performs on real tasks by analyzing user query logs. We use logs collected over several days to obtain 1244 unique transformation tasks (users have to "opt in" for *TDE* to log their queries – the default is opt-out). We manually inspect each query.

For 910 out of the 1244 tasks, *TDE* returns at least one synthesized program consistent with all input/output. We manually inspect users' input/output examples to understand the intent, and then verify the correctness of the result. Out of these, 496 tasks (39.8%) are verified to be correct for the rank-1 program produced (of which 153 invoke at least one function, and 343 use pure string transformations). Verifying lower-ranked programs (e.g. top-10) is more labor-intensive but should lead to a higher success rate.

For the tasks that *TDE* fails (defined as either having no programs produced, or the rank-1 program is judged to be incorrect), we analyze the underlying cause. For 206 tasks (16.5%), users provide only 1 or 2 output examples to demonstrate the task (we recommend 3), which makes the tasks difficult and even ambiguous. For 170 tasks (13.6%), we find the task itself to be ill-formed, due to bad input (e.g., users not understanding this feature and provide only one column of data), input/output in languages other than English (currently not supported), and tasks with unclear intent. For about 40 tasks (3%), a mapping relationship is needed not indexed. The remaining tasks (around 27%) fail mostly due to missing functionalities in *TDE* index.

While our initial experience with *TDE* reveals a number of areas for improvement, it also shows the promise of *TDE* in solving complex transformations using existing domain-specific logic. Just like Google and Bing were not perfect in finding relevant documents in their early days, we hope *TDE* will continue to improve as a "search engine" for data transformation, by growing its index and improving its algorithms using logged user interactions.

10. REFERENCES

[1] Bing maps api. `https://www.microsoft.com/maps/choose-your-bing-maps-API.aspx`.

[2] Informatica Rev. `https://www.informatica.com/products/data-quality/rev.html`.

[3] Openrefine. `openrefine.org`.

[4] Paxata. `https://www.paxata.com/`.

[5] Roslyn compiler framework. `https://github.com/dotnet/roslyn/wiki/RoslynOverview`.

[6] Talend. `https://www.talend.com/`.

[7] Transform Data by Example (from Microsoft Office Store). `https://aka.ms/transform-data-by-example-download`.

[8] Trifacta. `https://www.trifacta.com/`.

[9] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *ICDE*, 2016.

[10] E. Al-Masri and Q. H. Mahmoud. Investigating web services on the world wide web. In *WWW 2008*.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.

[12] K. Chakrabarti, S. Chaudhuri, Z. Chen, K. Ganjam, and Y. He. Data services leveraging bing's data assets. *IEEE Data Eng. Bull.*, 2016.

[13] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, 2003.

[14] M. Gollery. Bioinformatics: Sequence and genome analysis. *Clinical Chemistry*, 2005.

[15] J. Hare, C. Adams, A. Woodward, and H. Swinehart. Forecast snapshot: Self-service data preparation, worldwide, 2016. *Gartner, Inc.*, February 2016.

[16] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *SIGPLAN 2011*.

[17] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep web. *CACM 2007*.

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

[19] Z. Huang and Y. He. Auto-detect: Data-driven error detection in tables. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1377–1392. ACM, 2018.

[20] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.

[21] U. Kumar, V. Kumar, and J. N. KAPUR. Normalized measures of entropy. *International Journal Of General System*, 1986.

[22] V. Le and S. Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, 2014.

[23] H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[24] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *PVLDB, 1(2):1241-1252*, 2008.

[25] R. L. Sallam, P. Forry, E. Zaidi, and S. Vashisth. Gartner: Market guide for self-service data preparation. 2016.

[26] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB, 9(10):816-827*, 2016.

[27] B. C. Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[28] Y. Wang and Y. He. Synthesizing mapping relationships using table corpus. In *SIGMOD*, 2017.

[29] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD 2012*.

[30] C. Yan and Y. He. Synthesizing type-detection logic for rich semantic data types using open-source code. In *SIGMOD*, 2018.

[31] D. Yankov, P. Berkhin, and L. Li. Evaluation of explore-exploit policies in multi-result ranking systems. *arXiv preprint arXiv:1504.07662*, 2015.

[32] E. Zhu, Y. He, and S. Chaudhuri. Auto-join: Joining tables by leveraging transformations. *PVLDB, 10(10):1034-1045*, 2017.