General-purpose Declarative Inductive Programming with Domain-Specific Background Knowledge for Data Wrangling Automation

Lidia Contreras-Ochando, César Ferri, José Hernández-Orallo,
Fernando Martínez-Plumed, María José Ramírez-Quintana

Universitat Politècnica de València, Spain
{*liconoc,cferri,jorallo,fmartinez,mramirez*}*@dsic.upv.es*

Susumu Katayama

University of Miyazaki, Japan
skata@cs.miyazaki-u.ac.jp

Abstract

Inductive learning is the general process of inducing a general rule (a hypothesis) from a set of observed examples which describe a problem to be solved. Given one or two examples, humans are good at understanding how to solve a problem independently of its domain, because they are able to detect what the problem is and to choose the appropriate background knowledge according to the context. For instance, presented with the string "8/17/2017" to be transformed to "17th of August of 2017", humans will process this in two steps: (1) they recognise that it is a date and (2) they map the date to the 17th of August of 2017. Inductive Programming (IP) aims at learning declarative (functional or logic) programs from examples. Two key advantages of IP are the use of background knowledge and the ability to synthesize programs from a few input/output examples (as humans do). In this paper we propose to use IP as a means for automating repetitive data manipulation tasks, frequently presented during the process of *data wrangling* in many data manipulation problems. Here we show that with the use of general-purpose declarative (programming) languages jointly with generic IP systems and the definition of domain-specific knowledge, many specific data wrangling problems from different domains can be automatically solved from very few examples. We also propose an integrated benchmark for data wrangling, which we share publicly for the community.

Keywords: Inductive Programming, Data Wrangling Automation, Declarative Programming Languages, Domain-specific Background Knowledge

1. Introduction

The term 'data wrangling' usually refers to a great deal of repetitive and very time-consuming data preparation tasks, such as the acquisition, integration, manipulation, cleansing, enriching and transformation of data from their raw format to a more structured and valuable form for easy access and analysis [1]. The use of ETL[1] tools and other scripting languages for data wrangling partially alleviate the problem, but most of the effort is still manual and non-systematic. Consequently, progress in the (semi-)automation of data wrangling tasks can have an enormous impact in the costs of data science projects and other data manipulation problems, and can also allow data scientists focus on the valuable knowledge discovery process or in the actual task they are doing.

Many data wrangling problems look automatable, especially because the user can indicate a few illustrative examples that can be used by an Inductive Programming (IP) system [2, 3, 4, 5] to infer a pattern, or inductive hypothesis, that can be used to complete the rest of the examples automatically. Table 1 shows one example (dates transformation) that can be completed by non-expert people easily, without further knowledge about the source of the data. It is a very encapsulated problem, inputs and outputs, which should be well handled by machines.

Id	**Input**	**Outputs**		**Id**	**Input**	**Output**
1	25-03-74	*25/03/74*		5	17-05-17	17/05/17
2	29-03-86	*29/03/86*		6	25-08-05	25/08/05
3	11-02-96	11/02/96		7	30-06-75	30/06/75
4	11-17-98	17/11/98		8	...	...

Table 1: Dataset composed of dates (input) and desired output format. An automatic data wrangling system is fed with the two first examples (in italics) and should automatically complete the rest of the cells (outputs).

Nevertheless, many data wrangling problems also require an important degree of background knowledge because they depend on the application

[1] Originally from the data warehousing terminology, ETL is the process responsible for the extraction, transformation and load of the data into a repository.

context of the data. Table 2 shows an example of a common data wrangling problem: given a list of dates, extract the day from each of them. The difficulty in this problem lies in the different date formats (dependent on the original country where the data come from), where the day can be the first, second or third number, and these numbers can be delimited by different symbols. A system for automating this kind of problem based on string transformations may never find the right solution using its search space since it does not know what the real problem is: extracting the first number? the first two digits? or everything before any symbol? In order to understand and complete the transformation, we must know how dates work, their constraints and how they are usually represented. We know that there are only twelve months, that days can only range between 1 and 31 and that years are usually abbreviated with two single digits.

Id	**Input**	**Output**		**Id**	**Input**	**Output**
1	25-03-74	25		5	17/05/17	17
2	03/29/86	29		6	25-08-05	25
3	11.02.18	11		7	06 30 1975	30
4	1998/12/25	25		8	...	...

Table 2: (Dataset composed of dates under very different formats (input) from which the day is extracted (output).

In order to solve this problem, we can split the data wrangling problem into two steps: first, we need to know which the domain is (e.g., dates); and, second, we need to know which transformations we have to apply to the input to obtain the output. For humans this is a relatively easy step because we have information of the context, but it is not so easy for machines. We need to specify relevant background knowledge as well as the necessary transformations (depending on the domain). Of course, some of this knowledge may be insufficient to sort out some ambiguities, such as "11.02.18" (this date can be in DDMMYY, YYMMDD or MMDDYY formats). This problem may be automatically solved by computers (through program synthesis) if they are able to recognise the domain (i.e., dates), and have a sufficiently rich set of functions to deal with the context. Not only does this impose a strong bias that guides the process of finding the transformation pattern that has to be applied but also introduces some useful functions that render the solution (the inferred program) much shorter in terms of the functions involved. This

size of the solution (in terms of primitives/functions involved) is known as the *depth* (d).

Of course, dates are not the only kind of data. If we want to deal with physical addresses, we need to provide functions that handle symbols such as "St", "Rd", order of postcodes, etc. Similarly, if we want to deal with people names, we should understand strings such as "Mrs", "Dr", etc. Since all of these cases are very common in databases and other kinds of data that are processed in data science projects, we can add the relevant functions to a general domain library. However, as more kinds of data are required, this library would become huge. Even if the depth would have not changed for the original date problem, the inductive inference process needs to choose from a much larger space of functions, which makes it much harder. This is known as the *breadth* (b). Clearly, both the depth and the breadth highly influence the hardness of the problem, jointly with the number of examples, n. Actually, for hypothesis-oriented induction, hardness strongly depends on d and b, in a way that is usually exponential, $O(b^d)$ [6, 7]. How can we keep both, and especially b, at very low levels?

In this paper, we control the depth and breadth of the inductive inference problem by choosing a *domain-specific background knowledge* (DSBK) for each kind of problem. Based on any IP system, which is hypothesis-oriented rather than data-oriented, we see that the effort only depends on these two parameters, d and b, being almost constant on the number of examples. The user just needs to suggest which domain to use for a particular problem: dates, times, emails, names, phones, etc. Nevertheless, we envisage that this step is easily automatable too, using some domain inference process that can suggest this to the user, as we discuss at the end of the paper. It is important to remark that the inductive inference engine is the same, independently of whether we are handling dates or telephone numbers. We do not build a data wrangling system specialised for a particular domain-specific language for each case. Instead of this, we allow the system to use different DSBKs.

There are several advantages of this approach. The same data wrangling tool can be used for a diversity of problems and domains, without specialised tools for every domain. Second, a set of DSBK libraries can be provided by the tool but also extended by users and communities, especially if the language for adding or modifying functions is general-purpose and well known (e.g., Haskell [8, 9], Prolog [10], etc.).

Overall, this paper contains several contributions:

- We show how the use of functional programming languages and the definition of domain-specific knowledge, is able to improve the results of other state-of-the-art data wrangling approaches.
- We analyse how the breadth, depth and number of instances affect the efficiency, showing that we can achieve a trade-off between breadth and depth, and still solve many problems with very few examples.
- We provide a set of datasets specifically designed to be the first benchmark for the evaluation of data wrangling tools focusing on column transformation problems.

The paper is organised as follows. Section 2 summarises the most relevant related works. Section 3 addresses the problem of automating data wrangling with an IP system. The domains employed are detailed in Section 3.2. The experimental evaluation is included in Section 4. Finally, Section 5 remarks the conclusions and future work.

2. Related Work

In this section we review some of the existing works that have addressed the problem of data wrangling automation, emphasising on programming by example and inductive programming (IP).

The importance of data wrangling in the quality and cost of data science projects has motivated an enormous effort in techniques and approaches, including commercial platforms that go beyond ETL tools. For instance, *OpenRefine* [11] provides a set of built-in operators to specify data transformations (assuming the data are in a tabular format). *Ajax* [12] brings a SQL-like language to statements extended with advanced facilities for entity resolution) that enables the user to specify the sequence of data transformations. Trifacta *Wrangler* [13] generates a ranked list of suggested transformations and text extractions also inferred automatically from user input, the data type and some heuristics using programming-by-demonstration techniques.

All the above systems are able to use different approaches depending on the data type. For instance, they do not handle numerical data in the same way as they use strings. In general, these tools have predefined "types" or structures for emails, gender, phones, credit cards, social security number,

etc. However, their pattern generation engines are usually based on predefined rules, but not on a fully inductive inference system able to combine different and appropriate to the problem (user-defined) functions.

Given this limitation of commercial systems, research has focused on the inductive inference part of the problem, when the pattern involves the combination of several manipulation functions. This generation of approaches is based on *Inductive Programming* [4, 14], which has recently shown a large potential for data wrangling automation. IP addresses the problem of learning small (but complex) programs from very few representative input/output examples, generated as the user transforms one (or very few) particular instances or fields of the data. The application has been so successful that Microsoft includes some of these tools in Excel, known as *FlashFill* [15], and variants have been derived for other data manipulation problems, such as *FlashExtract* [16], *FlashRelate* [17], and *BlinkFill* [18]. One of the reasons of the success of these systems is the use of domain-specific languages (DSLs). For instance, *FlashFill* is able to make syntactic transformations of strings using restricted forms of regular expressions, conditionals and loops on spreadsheet tables.

The use of DSLs has overcome the limitations of general-purpose IP systems such as Progol [19], IGOR2 [20], *MagicHaskeller* [21], FLIP [22, 23, 24, 25], *Metagol* [26], *gErl* [27, 28, 29, 30, 31] and many others. The languages behind these systems (Prolog, Haskell, Erlang, etc.) have such a diversity of functions and possible combinations that the breadth and depth for the search problem is usually problematic. DSLs, on the other hand, are much more ad-hoc when dealing specific data wrangling problems, and reduce the search space considerably. However, the use of DSL systems for data wrangling automation also brings some disadvantages: (1) Using DSLs implies the use of languages that are specifically defined for a particular type of data processing (e.g., string processing, number processing, etc.). Whenever a new application or domain is required, a new domain-specific language has to be created, and the inductive engine recoded for it. Despite the effort of making this process more efficient in the recent years, it still depends on languages that are not of general purpose, and users need some effort to understand them, in order to create new functions, when possible; (2) These systems work using a specific set of transformations depending on the type of input, assuming the input to be in a unique format. This means that, even when the domain of the data is known or can be inferred, whether different formats of a data type appear in different rows of the same column (such as the

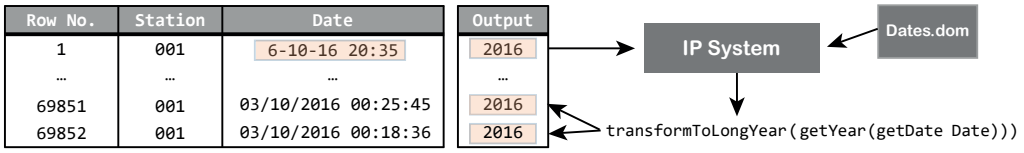


Figure 1: Automating data wrangling with IP: process example. The first row (Data and Output) is used as a input predicate for the IP system. The function returned using the correct domain is applied to the rest of the instances to obtain the outputs.

examples of dates seen in Table 2), the system is unable to find a solution for transforming each example, resulting in the correct transformation of only those examples with the same type of the example used as input.

In this paper, we propose the use of IP systems which (1) are 'specialised' with appropriate libraries that define domain-specific background knowledges (DSBKs), reducing the breadth of the search problem; and (2) are able to extract or transform data from one or few input examples to correct outputs, depending on the data domain and context and independently of the different formats appearing on the input column.

3. Automating data wrangling

The overall idea is to automate the process of transforming data from one format to another, depending on the data domain, using a general-purpose IP system at the core, but enhanced to handle configurable function libraries for each domain (see Figure 1). For this, we do the following steps:

1. We take a dataset of input-output pairs and detect the domain of the data.

2. We set the domain as background knowledge for the IP system.

3. One or more examples are sent to the IP system as inputs, such as the few first rows in Table 1, in the same way a user could complete a few examples. These examples are used as input predicates for the system.

4. With the correct DSBK, The system is able to return a list of transformations addressing the problem as the resulting function (f) that is applied to the rest of the inputs, obtaining the new values for the output column.

3.1. Domain-Specific Induction

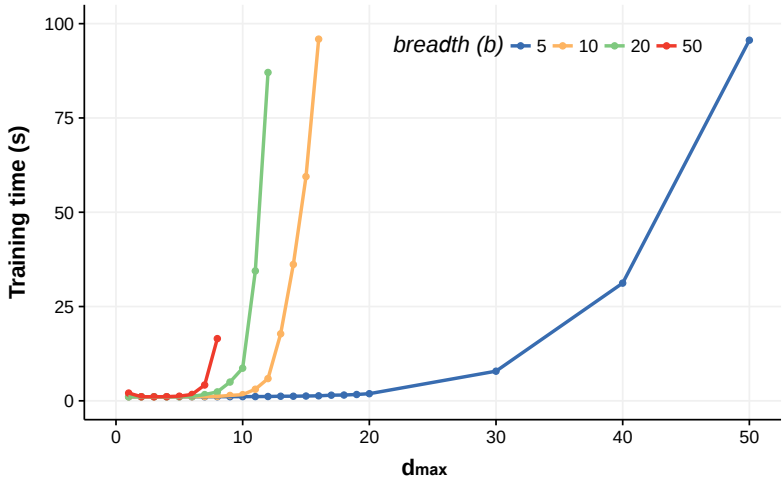
For the purpose of this work we have used *MagicHaskeller* [21] as the IP core system for several reasons[2]. First of all, *MagicHaskeller* is a general-purpose learning system that works with Haskell, a functional programming language that makes it much easier to add domains and transformations. Besides, *MagicHaskeller* is a very powerful system that can solve many problems using only one example from the data. It is also possible to provide *MagicHaskeller* with different data wrangling domains as different sets of background knowledge's functions.

In a nutshell, *MagicHaskeller* is a general-purpose inductive functional programming system that learns Haskell programs from pairs of input-output examples, also expressed in Haskell. *MagicHaskeller* receives an input example (x) and the expected result (y), and returns a list of functions (f) that make the values of the expressions fx and y be equal, which in Haskell notation is expressed as the boolean predicate `f x == y`. *MagicHaskeller* looks for combinations of one or more functions that are defined in its library to work like the f above.

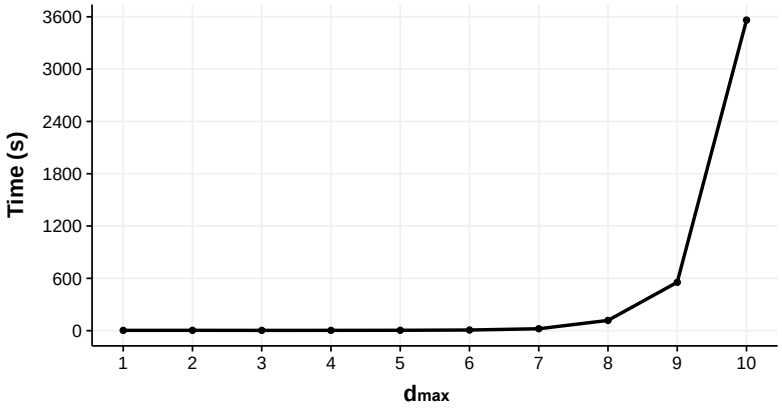
MagicHaskeller works in two steps: *(1)* The *Hypotheses Generation* phase, and *(2)* the *Hypotheses Selection* phase. In (1), *MagicHaskeller* starts with a predefined d_{max} value (maximum d allowed for the solution) and a set of b functions in the library. Then, *MagicHaskeller* continues with the preparation of hypotheses by generating all the type-correct expressions that can be expressed by function application and lambda abstraction using up to the maximum depth (d_{max}) the functions provided in the library. Although *MagicHaskeller* is very powerful for finding the simplest and most effective solutions (that is, those with smallest Kolmogorov complexity), depending on the problem, the solution might require the combination of many function symbols (that is, a solution with a large depth d). When the d required is higher than the d_{max} value used, *MagicHaskeller* is not able to find the solution (because it cannot reach the necessary number of functions combined). Trying to increase the d_{max} value to achieve the result may cause an increment of time over the top. On the contrary, trying to reduce d, we may be tempted to add many powerful and abstract functions to the library. But, in this case, *MagicHaskeller* will have too many primitives to choose from (the breadth value b), and may not find it either because of the time needed to

[2]Note that all the experiments could be replicated using any other IP learning system.

combine all of them.



(a)



(b)

Figure 2: **(a)** Time *MagicHaskeller* needs for training with a set of primitives depending on the maximum number of primitives that are allowed in any synthesised function (d_{max}) and the number of primitives in the set (b). **(b)** Time *MagicHaskeller* needs for training and solve the same problem (concatenate two strings), using a set fixed of $b = 15$ primitives, with varying d_{max} from 1 to 10.

In general, it is usually estimated that for hypothesis-driven inductive inference, the computational complexity might be in the order of $O(b^d)$ [32]. Figure 2 (a) illustrates this by showing the time used by *MagicHaskeller* in

this phase when we vary both the number of functions included in the library (b) and the maximum depth value to obtain the solution (d_{max}).

Finally, in phase (2) we can provide one or more examples (as I/O pairs) to solve a specific problem. *MagicHaskeller* will use the combinations learnt at (1) to find one or more possible solutions to the problem. This solution (if exists) will be a combination of d functions (where $d \leq d_{max}$). In this regard, Figure 2(b) shows the time spent during phases (1) and (2) to solve an specific problem (with actual solution of $d = 1$), using the same set of functions (with $b = 15$), but changing the d_{max} value. We acknowledge that d_{max} value has a strong influence too even when there are solutions that require fewer primitives than the maximum depth. Given the heuristics and optimisations included in *MagicHaskeller*, it is still possible to have solutions in cases where $O(b^d)$ grows very fast, but we still see the exponential behaviour in both cases. In the next sections we will show that a good trade-off between d and b can be achieved by using specific domain libraries. Thus, in that follows, we will refer to our approach as *Domain-Specific Induction (DSI)*.

3.2. Domain-specific Background Knowledge

By default, *MagicHaskeller* includes a list of 189 basic Haskell functions. Table 3 shows some of these functions[3]. Although *MagicHaskeller* is able to solve many string and boolean problems by using its default library [21], this list of functions is not enough to solve more complex problems. For instance, the example shown previously in Table 1 is impossible to solve with *MagicHaskeller*'s default library since there is a need to replace each dash symbol ('-') with a slash symbol ('/'), and *MagicHaskeller* is unable to generate or use any character or digit if it is not defined as constant in its library or if it is not provided as an input parameter.

In order to solve this kind of problem we have to add constants to the library and some new functions to work with string problems. For this particular case, we can solve the problem by adding the primitives in Table 4 to the library.

Following this and some other examples [33] and the most common operators used by other data science tools [34][13][35], we have added to *MagicHaskeller* many new functions for solving common problems related to string

[3]The complete list of functions is published at: `https://github.com/liconoc/DataWrangling-DSI`

Functions	
0 :: Int	1 :: Int
(++) :: forall a . (-¿) ([a]) ([a] -¿ [a])	filter :: forall a . (a -¿ Bool) -¿ [a] -¿ [a]
isLower :: (-¿) Char Bool	words :: [Char] -¿ [[Char]]
(+) :: Int -¿ Int	True :: Bool
False :: Bool	isPunctuation :: (-¿) Char Bool
(+) :: (-¿) Int ((-¿) Int Int)	takeWhile :: forall a . (a -¿ Bool) -¿ [a] -¿ [a]
isDigit :: (-¿) Char Bool	not :: (-¿) Bool Bool
(-) :: Int -¿ Int -¿ Int	(&&) :: (-¿) Bool ((-¿) Bool Bool)
(——) :: (-¿) Bool ((-¿) Bool Bool)	not :: (-¿) Bool Bool
(-) :: Int -¿ Int -¿ Int	reverse :: forall a . [a] -¿ [a]

Table 3: Some default functions in *MagicHaskeller*.

Functions	Description
dash :: [Char]	Constant for dash ('-') symbol
slash :: [Char]	Constant for slash ('/') symbol
changePunctuationString :: [Char] -¿ [Char] -¿ [Char]	Replace a punctuation sign

Table 4: Functions needed to replace a dash symbol with a slash symbol using *MagicHaskeller*.

manipulation. Concretely, we have added 108 functions to solve the following string operations:

- **Constants:** Symbols, numbers, words or list of words.
- **Map:** Boolean functions for checking string structures.
- **Transform:** Functions that return the string transformed using one or more of the following operations:
  - **Add:** Appending elements to a string, adding them at the beginning, ending or a fixed position.
  - **Split:** Splitting the string into two or more strings by positions, constants or a given parameter.
  - **Concatenate:** Joining strings, elements of an array, constants or given parameters with or without adding other parameters or constants between them.
  - **Replace:** Changing one or more string elements by some other given element . This operation includes converting a string to uppercase and lowercase.

- **Exchange:** Swapping elements inside strings.
- **Delete/Drop/Reduce:** Deleting one or more string elements by some other given parameter, a position, size or mapping some parameter or constant.
- **Extraction:** Get one or more string elements.

With this set of functions in the system's library, we are able to solve many common string manipulation problems, such as the example in Table 1. However, the results can be less accurate for different examples. Trying to solve the example in Table 2 using the first row as a predicate (f "25-03-74" == "25") the first three results obtained may be: (1) *takeWhile isDigit "25-03-86"*; (2) *getStartToFirstSymbol "25-03-86" dash*; and (3) *take 2 "25-03-86"*. When we apply these functions to the firs row, we obtain the desired results, but, what happens if we apply these functions to the rest of the table? We can see the results in Table 5. It should be noted that only in the cases when the day is the first element of the date (with solutions 1 and 3) and the next symbol is a dash (with solution 2) the result is correct. The problem here is that we cannot assume that all the data in a column has always the same format. In this case, dates come from very different formats and extracting the first element not always results in getting the day. When data belong to a particular domain and the problem at hand ends up being a very exclusive task pertaining to that domain, more precise functions are needed in order to get correct results considering the context. However, as we have seen in the previous section, it is critical to reduce b while at the same time having the appropriate abstract primitives to learn the function with a short hypothesis (small d). This could be solved by detecting the domain of the data to be transformed and choosing a domain-specific library for it.

A high number of different domains can appear in any data science project related to data manipulation problems. In order to make our experiments and as other data wrangling tools have already done [36][37][38], we have selected some of the most used domains [37] and their most common problems [36] to work with. In this sense, for each domain we have a different background knowledge with a set of possible transformations. As we are working with *MagicHaskeller*, they are represented as Haskell functions. These are independent text files, editable by the user, which can be included as a parameter when *MagicHaskeller* is invoked. The DSBK files are:

- **Dates**: The DSBK includes 23 functions related to date manipulation

Id	Input	Expected Output	Actual Output (1)	Actual Output (2)	Actual Output (3)
1	*25-03-74*	*25*			
2	03/29/86	29	03	03/29/86	03
3	11.02.18	11	11	11.02.18	11
4	1998/12/25	25	1998	1998/12/25	19
5	17/05/17	17	17	17/05/17	17
6	25-08-05	25	25	25	25
7	06 30 1975	30	06	06 30 1975	06
8	...	...	...	...	...

Table 5: Example of a dataset with an input column composed of dates under very different formats, the expected output (day) and the actual outputs obtained using an inductive system with string manipulation functions. The first row is used as input predicate for the system. Green examples are correct results. Red examples are incorrect results. Solution (1): *takeWhile isDigit "input"*; Solution (2): *getStartToFirstSymbol "input" dash*; and Solution (3): *take 2 "input"*.

(and includes 139 primitives from the freetext BK), such as determining whether a substring is a month, getting the day in ordinal format, converting a month to numeric format or extending a two-digit year to a four-digit full format.

- **Emails**: This DSBK includes 9 functions related to email manipulation (and includes 93 primitives from the freetext BK), such as getting the words after or before the '@' symbol, append the '@' symbol at the end of a string or join two strings with the '@' symbol.

- **Names**: The DSBK includes 12 functions related to personal names manipulation (and includes 104 primitives from the freetext BK), such as getting the initials of a name or creating a user login.

- **Phones**: This DSBK includes 5 functions related phone numbers manipulation (and includes 124 primitives from the freetext BK), for example, setting the prefix by a country name or code.

- **Times**: This DSBK has 24 functions to deal with strings containing time (and includes 124 primitives from the freetext BK), such as 12/24h formats or changing time zone.

In total we have used 374 different functions. Although we are considering only six domains besides the basic string manipulation functions, it should be noted that many other domains can be created, and it is also easy to build domains that are defined as the union between two existing domains. Also, *MagicHaskeller* can be called with a small d_{max} parameter with one domain

to get results quickly and, if unsuccessful, try with a larger d_{max} or another domain. In this way, the search effort can be better handled, depending on the knowledge of the domain and the expected size of the solution.

4. Experiments

In this section, we perform a set of experiments to analyse the performance of the proposed approach. We also introduce the data wrangling benchmark and perform a comparison with other data wrangling tools.

The aim of our experiments is to analyse the extended capabilities of an IP learning system as a data wrangler. Besides, the experiments explore the improvement in the results when selecting the right DSBK in front of using a general background knowledge or an inappropriate DSBK. Also, and more importantly, we want to compare with other data wrangling systems on a range of data wrangling problems.

To perform the experiments we have followed a trained/test evaluation procedure, similar to [39, 40, 41, 15, 42]. We have used a set of datasets with different data wrangling problems (explained in the following sub-section) including inputs and expected outputs. For each of these datasets, we use only the first example as the input predicate for the IP system. Then, we fed the system with this first input/output example using, for each dataset, all the different DSBK. The result is a function f that is applied to the rest of the outputs. The accuracy in each case is the result of compare the transformed outputs with the real expected outputs.

For replicability reasons, the source code (scripts, domain files, primitive files *MagicHaskeller*, etc.) and all the results of these experiments are available online [4].

4.1. Data Wrangling Benchmark

Unfortunately, there is no general benchmark or public dataset repository accessible in reusable formats to analyse the quality of new data wrangling tools (for instance, in [40] the authors use a dataset with hundreds of data manipulation problems, but the benchmark is not public). In order to overcome this limitation, we have collected most of the datasets tested previously

[4] `https://github.com/liconoc/DataWrangling-DSI`

in other tools for data manipulation (such as *FlashFill* or *Wrangler*) and presented in the literature [39, 40, 41, 15, 42]. In addition, we have generated new datasets based on problems from these papers.

id	Domain	#Ex.	Description of the problem to solve
1	Freetext	12	Complete brackets (From [40])
2	Freetext	12	Extract the first character (From [43])
3	Freetext	24	Delete punctuation (From [40])
4	Freetext	18	Extract the capital letters (From [40])
5	Freetext	12	Extract a substring (From [44])
6	Dates	26	Change the punctuation of a date (From [41])
7	Dates	26	Extract the day from a date (Generated)
8	Dates	12	Extract the day from a date in ordinal format (Generated)
9	Dates	12	Extract the month from dates (Generated)
10	Dates	12	Extract the name of the month from dates (From [44])
11	Dates	9	Add punctuation to a date (From [44])
12	Dates	25	Change date format and punctuation (Generated)
13	Dates	12	Add punctuation and change the format of a date (From [44])
14	Emails	24	Extract words after '@' (From [44])
15	Emails	18	Join words with '@' (From [39])
16	Names	12	Generate a login from a name (Generated)
17	Names	12	Reduce name from one input (From [15])
18	Names	12	Reduce name from two inputs (From [15])
19	Names	12	Extract the honorific forms (From [15])
20	Phones	12	Add phone prefix by country name (From PROSE)
21	Phones	12	Add phone prefix by country name and '+' symbol (Generated)
22	Phones	12	Add a given phone prefix (From [44])
23	Phones	12	Extract a phone number from a string (From [44])
24	Phones	12	Add punctuation to a phone number (Generated)
25	Times	12	Extract the time from a string (Generated)
26	Times	12	Append a specific given time (minutes or seconds) (Generated)
27	Times	12	Increase the hour by a given value (Generated)
28	Times	12	Convert the time to 24h format (Generated)
29	Times	12	Convert time by a given time zone (Generated)
30	Units	12	Extract the units of a value (From [43]
31	Units	12	Detect the system units by the units of a value (Generated)
32	Units	12	Convert a value to a different unit (Generated)

Table 6: Datasets included in the new data wrangling repository offered for the data science research community. *#Ex.* shows cardinality.*Freetext* represents the functions created for solving string manipulation problems.

Overall, we have collected or generated 32 datasets and we have published them on the first data wrangling dataset repository, which is online and

available at `http://users.dsic.upv.es/~flip/datawrangling/`. Table 6 shows a summary of the datasets in this new repository.

4.2. Results

With a focus on our system, Table 7 shows the results (accuracy) for all the datasets, using just one example (the first one of each dataset), when *MagicHaskeller* is run without extra DSBK (*default*), adding the string manipulation functions (*freetext*), with a particular DSBK (*dates, emails, names, phones, times, units*) and with all DSBKs together (a unique set of primitives with all the functions together). In each case, *MagicHaskeller* returns a potential solution (or nothing if the problem cannot be solved) which is applied to the rest of input examples to see whether the obtained output matches the expected one. Time execution is limited to $120s$ with $d_{max} = 4$.

The results are much better when the right domain is chosen for the problem. Note that putting all domains together (*all*) implies such big a value of b that *MagicHaskeller* could not solve many problems within the maximum time period. In the same way, some specific problems (datasets #10, #12, #13, #21, #27, #29 and #32) cannot be solved using a d_{max}=4 because they need a higher value in order to find the correct solution.It has to be noticed that since all the DSBK contain some functions for string manipulation, many of them can solve problems related to basic string problems (*freetext* domain). Some problems related to specific domains can also be solved by using basic string manipulation functions, therefore, in this case, any DSBK containing these functions is able to solve the problem. For instance, dataset #6 (*dates* domain) can be solved by using constants and the *freetext* function *changePunctuationString*, as we have seen in section 3.2. Since these functions are included in other domains not only *dates* has the best accuracy, but also *freetext*, *emails*, *phones*, *times* and *units*.

We have also compared the performance of our DSI approach using *MagicHaskeller* with other data wrangling tools, concretely, *FlashFill* [15]. *Flashfill* works in the same way as our approach, namely, it uses one or more input instances to try to induce a potential solution, which is then applied to the rest of examples. If no solution is found or the problem at hand is not solvable by *FlashFill*, it returns, respectively, a void function or an error.

Table 8 shows some illustrative outcomes obtained for each dataset and

id	*Domain*	*default*	*freetext*	*dates*	*emails*	*names*	*phones*	*times*	*units*	*all*
1	freetext	0.00	**1.00**	0.00	0.00	0.00	**1.00**	**1.00**	**1.00**	0.00
2	freetext	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.00
3	freetext	0.48	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.00
4	freetext	**1.00**	**1.00**	0.00	0.00	**1.00**	**1.00**	**1.00**	**1.00**	0.00
5	freetext	0.00	**0.55**	0.18	**0.55**	**0.55**	**0.55**	**0.55**	**0.55**	0.00
6	dates	0.00	**1.00**	**1.00**	**1.00**	0.00	**1.00**	**1.00**	**1.00**	0.00
7	dates	0.60	0.60	**1.00**	0.28	0.60	0.60	0.60	0.60	0.00
8	dates	0.00	0.00	**0.91**	0.00	0.00	0.00	0.00	0.00	0.00
9	dates	0.00	0.00	**1.00**	0.00	0.27	0.00	0.00	0.00	0.00
10	dates	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
11	dates	0.00	**1.00**	**1.00**	**1.00**	0.00	0.00	**1.00**	**1.00**	0.00
12	dates	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
13	dates	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
14	emails	0.00	0.04	0.04	**1.00**	0.04	0.04	0.04	0.04	0.00
15	emails	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	0.00
16	names	0.00	0.00	0.00	0.00	**0.91**	0.00	0.00	0.00	0.00
17	names	0.00	0.00	0.00	0.00	**0.91**	0.00	0.00	0.00	0.00
18	names	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00
19	names	0.45	0.73	0.45	0.73	**1.00**	0.73	0.73	0.73	0.00
20	phones	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00
21	phones	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
22	phones	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00
23	phones	0.00	0.27	0.00	0.27	0.27	**1.00**	0.27	0.27	0.00
24	phones	0.00	**1.00**	**1.00**	**1.00**	0.00	**1.00**	0.00	**1.00**	0.00
25	times	0.36	0.91	0.91	0.91	0.91	0.91	**1.00**	0.91	0.00
26	times	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00
27	times	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
28	times	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00
29	times	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
30	units	0.64	0.18	0.18	0.73	0.18	0.18	0.18	**1.00**	0.00
31	units	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00
32	units	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 7: Accuracy obtained per dataset depending on the set of primitives (DSBK) used to train *MagicHaskeller*. The results are obtained with d_{max}=4, $n = 1$ and a maximum execution time of 120s. Maximum accuracy values in bold.

tool[5] as well as the accuracy values for each dataset. The first instance (in italic) for each dataset (*input* column) is the one used for inducing the solution in the different tools. Here, we can see some strength and weakness in each tool. For instance, *Flashfill* works fine with emails and some basic

[5]For the complete description of results see `http://users.dsic.upv.es/~flip/datawrangling/`

id	input	expected output	FlashFill	DSI
3	*1-452-789-4567* 1-406-789-1562 1-4565 Etiam dapibus.	*14527894567* 14067891562 14565 Etiamdapibus	 14067891562 14565 	 14067891562 14565 Etiamdapibus
		Accuracy:	0.48	1
4	*International Business Machines* Principles Of Programming Languages International Conference on Data Mining series Association of Computational Linguistics	*IBM* POPL ICDM ACL	 POPL ICDM ACL	 POPL ICDM ACL
		Accuracy:	1	1
8	*3/29/86* 10 12 69 04/05/99 27/07/2007	*29th* 10th 04th 27th	 12th 05th 07th	 10th 04th 27th
		Accuracy:	0	1
9	*2 of September of 2010, Monday* 13 November 2008 Tuesday, September 16, 1986 February 4, 2008	*September* November September February	 2008 September 2008	 November September February
		Accuracy:	0.36	1
14	*Nancy.FreeHafer@fourthcoffee.com* iabetrae@yahoo.es Sb.edhxo.sk8@hotmail.com dala_aguera_m500@hotmail.com	*fourthcoffee.com* yahoo.es hotmail.com hotmail.com	 yahoo.es hotmail.com hotmail.com	 yahoo.es hotmail.com hotmail.com
		Accuracy:	1	1
15	*Sophia & domain* elizabeth & gmail joypao & hotmail casper & canal13	*Sophia@domain.com* elizabeth@gmail.com joypao@hotmail.com casper@canal13.com	 elizabeth@gmail.com joypao@hotmail.com casper@canal13.com	 elizabeth@gmail.com joypao@hotmail.com casper@canal13.com
		Accuracy:	1	1
17	*Damian Gobbee* Antonio Hege Damancio Hivser-Kleiner Prof. Edward Davis	*D.Gobbee* A.Hege D.Hivser-Kleiner E.Davis	 A.Hege D.Kleiner P.Davis	 A.Hege D.Hivser-Kleiner E.Davis
		Accuracy:	0.63	0.91
19	*Dr. B. Schdur* Prof. H. Huifen Louis Johnson, PhD Robert Mills	*Dr.* Prof. PhD	 Prof. Lou Rob	 Prof. PhD
		Accuracy:	0.72	1
20	*235-7654 & Taiwan* 17-455-81-39 & Spain 618-4390 & Panama 25-613-24-50 & Chile	*(886) 235-7654* (34) 17-455-81-39 (507) 618-4390 (56) 25-613-24-50	 (886) 17-455-81-39 (886) 618-4390 (886) 25-613-24-50	 (34) 17-455-81-39 (507) 618-4390 (56) 25-613-24-50
		Accuracy:	0	1
23	*23/11/18 425-785-4210* 425-613-2450 000-000 [TS]865-000-0000 - 06-23-09 17:58-19:29, 425-743-1650	*425-785-4210* 425-613-2450 865-000-0000 425-743-1650	 2450 000-000 06-23-2009 425-743-1650	 425-613-2450 865-000-0000 425-743-1650
		Accuracy:	0.36	1
25	*08:55 PM CET* 20:15:00 10:05:00 AM UTC 21:20	*08:55* 20:15:00 10:05:00 21:20	 20:15:00 10:05:00 UTC 21:20	 20:15:00 10:05:00 21:20
		Accuracy:	0.91	1
28	*01:34:00 & 5* 01:55 & 5 16:15:12 & 5 21:20 & 5	*06:34:00* 06:55 21:15:12 02:20	 06:55 06:15:12 06:20	 06:55 21:15:12 02:20
		Accuracy:	0.10	1
30	*56.77cl* 84Kg 39.88 A 1nm	*cl* Kg A nm	 Kg A nm	 Kg A nm
		Accuracy:	1	1
31	*56.77cl* 84Kg 39.88 A 1nm	*Volume* Mass Electricity Length	 Volume Volume Volume	 Mass Electricity Length
		Accuracy:	0.10	1

Table 8: Example of results obtained (using *MagicHaskeller* as IP core) compared with *FlashFill*. *Output* is the expected output. The first row of each dataset (*id*) is the example given to *FlashFill* and *MagicHaskeller* to learn. Green and Red colours mean, respectively, correct and incorrect results. The accuracy is $correct\_examples/(total\_examples - n)$, where $n = 1$.

string transformations, but it fails when it has to deal with titles or honorific forms in people names, with problems related to phones prefixes or times and when it has to work with dates in different formats. For its part, DSI using *MagicHaskeller* is able to find the correct solution for the problem at hand, even with only one example, although it still has problems with unexpected punctuation marks (for instance in dataset #17). In summary, the results show that our approach is able to overcome other tools when dealing with data wrangling problems.

5. Conclusions

In this paper we have adapted a general IP tool to deal with a range of data wrangling problems by using domain-specific background knowledge libraries. Given the impact that the size of the library (b) and the size of the solution (d) have when solving a data manipulation problem, we found a trade-off that produces positive solutions for many datasets for which other tools are not able to give a satisfactory solution. This is done without the need of increasing the number of examples or using external information (further unlabelled examples or a distribution of the outputs). Also, the application of this approach is easy, since the user only needs to choose a domain and write one example (or very few for some cases).

Users can also edit and create the domain files in a general-purpose functional programming language such as Haskell, making the system more powerful and able to deal with more and more domains. This contrasts with a mainstream approach based on domain-specific languages, where a change of the DSL to cover other domains cannot be done by the user and might require a redesign of the system. Furthermore, the evaluation performed shows that the DSI approach overcomes the results obtained by other data wrangling tools, such as FlashFill, mainly due to its adaptability to the problem domain for the use of domain-specific background knowledge (DSBK). This shows that for this repetitive snippets of code that are necessary for data manipulation problems, we can replace some of the tedious programming effort by the selection of libraries or the definitions of proper functions to handle existing or new domains. Functional programming languages, as we have seen, are particularly appropriate for this. In the end, these data wrangling systems over functional programming languages can actually have the effect of truly incorporating automated programming and program synthesis as a

toolbox, even if at the level of the generation of small snippets, for these kinds of applications.

Finally, we provide what might be considered as the first public repository of datasets for testing data wrangling tools. Although there are several approaches and systems in the literature dealing with the issue under consideration here, none of them provide public access, nor a complete description of the datasets used for their evaluation. In this way, the evaluation procedures are not replicable and neither is the data reusable. To solve this problem we have collected different problems from the literature and related software, together with a few freshly-generated ones. With all this data, we have generated a variety of datasets for four different domains (dates, emails, names and words) covering different specific problems in each of them. This repository is open and freely available, and it is already being extended with more types of problems and domains.

As future work, we plan to automate the detection of the domain at hand by using machine learning techniques. The idea is to learn a meta-model able to automatically select (or suggest) the appropriate DSBK from the features of the problem. Finally, we plan to integrate everything in a more standalone tool or web service that would allow other users or applications to use this approach in a more standard and accessible fashion. In this way, this approach could be transformed into an API to be used with any language or tool.

References

[1] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, P. Buono, Research directions in data wrangling: Visualizations and transformations for usable and credible data, Inf. Visualization 10 (4) (2011) 271–288.

[2] P. Flener, S. Yılmaz, Inductive synthesis of recursive logic programs: Achievements and prospects, The Journal of Logic Programming 41 (2-3) (1999) 141–195.

[3] S. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, The Journal of Logic Programming 19 (1994) 629–679.

[4] E. Kitzelmann, U. Schmid, Inductive synthesis of functional programs: An explanation based generalization approach, Journal of Machine Learning Research 7 (Feb) (2006) 429–454.

[5] J. Hernández-Orallo, M. J. Ramírez-Quintana, A strong complete schema for inductive functional logic programming, in: International Conference on Inductive Logic Programming, Springer, 1999, pp. 116–127.

[6] R. Henderson, Incremental learning in inductive programming, in: International Workshop on Approaches and Applications of Inductive Programming, Springer, 2009, pp. 74–92.

[7] J. Hernández-Orallo, Deep knowledge: Inductive programming as an answer, in: S. Gulwani, E. Kitzelmann, U. Schmid (Eds.), Approaches and Applications of Inductive Programming (Dagstuhl Seminar 13502), Vol. 3, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2014, pp. 43–66. doi:10.4230/DagRep.3.12.43.
URL http://drops.dagstuhl.de/opus/volltexte/2014/4507

[8] S. P. Jones (Ed.), Haskell 98 Language and Libraries: The Revised Report, http://haskell.org/, 2002.
URL http://haskell.org/definition/haskell98-report.pdf

[9] S. Jones, Compiling haskell by program transformation: A report from the trenches, Programming Languages and Systems, ESOP96 (1996) 18–44.

[10] J. W. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 1993.

[11] K. Ham, Openrefine (version 2.5). http://openrefine. org., Journal of the Medical Library Association: JMLA 101 (3) (2013) 233.

[12] H. Galhardas, D. Florescu, D. Shasha, E. Simon, Ajax: An extensible data cleaning tool, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, ACM, New York, NY, USA, 2000, pp. 590–. doi:10.1145/342009.336568.
URL http://doi.acm.org/10.1145/342009.336568

[13] S. Kandel, A. Paepcke, J. Hellerstein, J. Heer, Wrangler: Interactive visual specification of data transformation scripts, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2011, pp. 3363–3372.

[14] S. Gulwani, J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, B. Zorn, Inductive programming meets the real world, Communications of the ACM 58 (11) (2015) 90–99.

[15] S. Gulwani, Automating string processing in spreadsheets using input-output examples, in: Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, ACM, New York, NY, USA, 2011, pp. 317–330.

[16] V. Le, S. Gulwani, FlashExtract: A framework for data extraction by examples, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, New York, NY, USA, 2014, pp. 542–553. doi:10.1145/2594291.2594333.
URL http://doi.acm.org/10.1145/2594291.2594333

[17] D. W. Barowy, S. Gulwani, T. Hart, B. G. Zorn, FlashRelate: extracting relational data from semi-structured spreadsheets using examples, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, 2015, pp. 218–228.

[18] R. Singh, BlinkFill: Semi-supervised programming by example for syntactic string transformations, Proceedings of the VLDB Endowment 9 (10) (2016) 816–827.

[19] S. Muggleton, Inverse Entailment and Progol, New Generation Computing, SI on Inductive Logic Programming 13 (3-4) (1995) 245–286. URL citeseer.nj.nec.com/muggleton95inverse.html

[20] E. Kitzelmann, M. Hofmann, Igor2–an inductive functional programming prototype, in: Proceedings of the System Demonstrations of the 18th European Conference on Artificial Intelligence, 2008, pp. 29–31.

[21] S. Katayama, An analytical inductive functional programming system that avoids unintended programs, in: Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, ACM, 2012, pp. 43–52.

[22] C. Ferri, J. Hernández-Orallo, M. J. Ramírez-Quintana, Incremental learning of functional logic programs, in: H. Kuchen, K. Ueda (Eds.), Functional and Logic Programming, Vol. 2024 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 233–247. doi:10.1007/3-540-44716-4_15.

[23] C. Ferri-Ramırez, J. Hernández-Orallo, M. Ramırez-Quintana, Learning functional logic classification concepts from databases, in: Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP 2000), pp. 296–308.

[24] C. Ferri Ramírez, J. Hernández Orallo, M. J. Ramírez Quintana, Aprendizaje automático de programas lógico-funcionales, Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial 4 (11).

[25] C. Ferri-Ramırez, J. Hernández-Orallo, M. Ramırez-Quintana, And/or trees for the synthesis of functional logic programs.

[26] S. H. Muggleton, D. Lin, A. Tamaddoni-Nezhad, Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited, Machine Learning 100 (1) (2015) 49–73.

[27] F. Martínez-Plumed, Incremental and developmental perspectives for general-purpose learning systems, Ph.D. thesis, Technical University of Valencia, Spain (2016). doi:10.4995/Thesis/10251/67269.

[28] F. Martınez-Plumed, C. Ferri, J. Hernández-Orallo, M. J. Ramırez-Quintana, Accelerating learning by reusing abstract policies in gerl, Multiconferencia CAEPIA 13 1383–1392.

[29] F. Martınez-Plumed, C. Ferri, J. Hernández-Orallo, M. J. Ramırez-Quintana, Policy reuse in a general learning framework, Multiconferencia CAEPIA 13 (2013) 119–128.

[30] F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, M.-J. Ramírez-Quintana, On the definition of a general learning system with user-defined operators, arXiv preprint arXiv:1311.4235.

[31] F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, M. J. Ramírez-Quintana, Learning with configurable operators and rl-based heuristics, in: International Workshop on New Frontiers in Mining Complex Patterns, Springer, 2012, pp. 1–16.

[32] G. Gottlob, N. Leone, F. Scarcello, On the complexity of some inductive logic programming problems, in: International Conference on Inductive Logic Programming, Springer, 1997, pp. 17–32.

[33] K. Nishida, 7 Most Practically Useful Operations When Wrangling with Text Data in R (Sep. 2016).
URL `https://blog.exploratory.io/7-most-practically-useful-operations-when-wrangling-with-text-data-in-r-7654bd9d1a0c`

[34] Key Features of RapidMiner Studio, https://rapidminer.com/products/studio/feature-list/.

[35] Wrangle Language - Trifacta Wrangler - Trifacta Documentation.
URL `https://docs.trifacta.com/display/PE/Wrangle+Language`

[36] V. Raman, J. M. Hellerstein, Potter's wheel: An interactive data cleaning system, in: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 381–390.

[37] Supported Data Types - Trifacta Wrangler - Trifacta Documentation.
URL `https://docs.trifacta.com/display/PE/Supported+Data+Types`

[38] Data types in Data Models - Microsoft Excel Documentation.
URL https://support.office.com/en-us/article/data-types-in-data-models-e2388f62-6122-4e2b-bcad-053e3da9ba90

[39] S. Bhupatiraju, R. Singh, A.-r. Mohamed, P. Kohli, Deep API programmer: Learning to program with APIs, arXiv preprint arXiv:1704.04327.

[40] K. Ellis, S. Gulwani, Learning to learn programs from examples: Going beyond program structure, 2017.
URL https://www.microsoft.com/en-us/research/publication/learning-learn-programs-examples-going-beyond-program//-structure/

[41] R. Singh, S. Gulwani, Transforming spreadsheet data types using examples, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, ACM, New York, NY, USA, 2016, pp. 343–356. doi:10.1145/2837614.2837668.
URL http://doi.acm.org/10.1145/2837614.2837668

[42] R. Singh, S. Gulwani, Predicting a correct program in programming by example, in: International Conference on Computer Aided Verification, Springer, 2015, pp. 398–414.

[43] L. Contreras-Ochando, F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, M. J. Ramírez-Quintana, General-purpose inductive programming for data wrangling automation, AI4DataSci @ NIPS 2016.

[44] O. Polozov, S. Gulwani, Program synthesis in the industrial world: Inductive, incremental, interactive, in: 5th Workshop on Synthesis (SYNT), 2016.