Synthesizing Number Transformations from Input-Output Examples

Rishabh Singh[1⋆] and Sumit Gulwani[2]

[1] MIT CSAIL, Cambridge, MA, USA
[2] Microsoft Research, Redmond, WA, USA

Abstract. Numbers are one of the most widely used data type in programming languages. Number transformations like formatting and rounding present a challenge even for experienced programmers as they find it difficult to remember different number format strings supported by different programming languages. These transformations present an even bigger challenge for end-users of spreadsheet systems like Microsoft Excel where providing such custom format strings is beyond their expertise. In our extensive case study of help forums of many programming languages and Excel, we found that both programmers and end-users struggle with these number transformations, but are able to easily express their intent using input-output examples.

In this paper, we present a framework that can learn such number transformations from very few input-output examples. We first describe an expressive number transformation language that can model these transformations, and then present an inductive synthesis algorithm that can learn all expressions in this language that are consistent with a given set of examples. We also present a ranking scheme of these expressions that enables efficient learning of the desired transformation from very few examples. By combining our inductive synthesis algorithm for number transformations with an inductive synthesis algorithm for syntactic string transformations, we are able to obtain an inductive synthesis algorithm for manipulating data types that have numbers as a constituent sub-type such as date, unit, and time. We have implemented our algorithms as an Excel add-in and have evaluated it successfully over several benchmarks obtained from the help forums and the Excel product team.

1 Introduction

Numbers represent one of the most widely used data type in programming languages. Number transformations like formatting and rounding present a challenge even for experienced programmers. First, the custom number format strings for formatting numbers are complex and take some time to get accustomed to, and second, different programming languages support different format strings, which makes it difficult for programmers to remember each variant.

⋆ Work done during an internship at Microsoft Research.

Number transformations present an even bigger challenge for end-users: the large class of users who do not have a programming background but want to create small, *often one-off*, applications to support business functions [4]. Spreadsheet systems like Microsoft Excel support a finite set of commonly used number formats and also let users write their own custom formats using a number formatting language similar to that of .NET. This hard-coded set of number formats is often insufficient for the user's needs and providing custom number formats is typically beyond their expertise. This leads them to solicit help on various online help forums, where experts typically respond with the desired formulas (or scripts) after few rounds of interaction, which spans over a few days.

In an extensive case study of help forums of many programming languages and Excel, we found that even though both programmers and end-users struggled while performing these transformations, they were able to easily express their intent using input-output examples. In fact, in some cases the initial English description of the task provided by the users on forums was inaccurate and only after they provided a few input-output examples, the forum experts could provide the desired code snippet.

In this paper, we present a framework to learn number formatting and rounding transformations from a given set of input-output examples. We first describe a domain-specific language for performing number transformations and an inductive synthesis algorithm to learn the set of all expressions that are consistent with the user-provided examples. The key idea in the algorithm is to use the interval abstract domain [2] to represent a large collection of consistent format expressions symbolically, which also allows for efficient intersection, enumeration, and execution of these expressions. We also present a ranking mechanism to rank these expressions that enables efficient learning of the desired transformation from very few examples.

We then combine the number transformation language with a syntactic string transformation language [6] and present an inductive synthesis algorithm for the combined language. The combined language lets us model transformations on strings that represent data types consisting of number as a constituent subtype such as date, unit, time, and currency. The key idea in the algorithm is to succinctly represent an exponential number of consistent expressions in the combined language using a `Dag` data structure, which is similar to the BDD [1] representation of Boolean formulas. The `Dag` data structure consists of program expressions on the edges (as opposed to Boolean values on BDD edges). Similar to the BDDs, our data structure does not create a quadratic blowup after intersection in practice.

We have implemented our algorithms both as a stand-alone binary and as an Excel add-in. We have evaluated it successfully on over 50 representative benchmark problems obtained from help forums and the Excel product team.

This paper makes the following key contributions:

- We develop an expressive number transformation language for performing number formatting and rounding transformations, and an inductive synthesis algorithm for learning expressions in it.

– We combine the number transformation language with a syntactic string transformation language to manipulate richer data types.
– We describe an experimental prototype of our system with an attractive user interface that is ready to be deployed. We present the evaluation of our system over a large number of benchmark examples.

2 Motivating Examples

We motivate our framework with the help of a few examples taken from Excel help forums.

Example 1 (Date Manipulation). An Excel user stated that, as an unavoidable outcome of data extraction from a software package, she ended up with a series of dates in the input column v_1 as shown in the table. She wanted to convert them into a consistent date format as shown in the output column such that both month and day in the date are of two digits.

Input v_1	Output
1112011	01/11/2011
12012011	12/01/2011
1252010	**01/25/2010**
11152011	**11/15/2011**

It turns out that no Excel date format string matches the string in input column v_1. The user struggled to format the date as desired and posted the problem on a help forum. After a few rounds of interactions (in which the user provided additional examples), the user managed to obtain the following formula for performing the transformation:

```
TEXT(IF(LEN(A1)=8,DATE(RIGHT(A1,4),MID(A1,3,2),LEFT(A1,2)),
     DATE(RIGHT(A1,4),MID(A1,2,2),LEFT(A1,1))),"mm/dd/yyyy")
```

In our tool, the user has to provide only the first two input-output examples from which the tool learns the desired transformation, and executes the synthesized transformation on the remaining strings in the input column to produce the corresponding outputs (shown in bold font for emphasis).

We now briefly describe some of the challenges involved in learning this transformation. We first require a way to extract different substrings of the input date for extracting the day, month, and year parts of the date, which can be performed using the syntactic string transformation language [6]. We then require a number transformation language that can map `1` to `01`, i.e. format a number to two digits. Consider the first input-output example `1112011 -> 01/11/2011`. The first two characters in the output string can be obtained by extracting `1` from the input string from any of the five locations where it occurs, and formatting it to `01` using a number format expression. Alternatively, the first `0` in the output string can also be a constant string or can be obtained from the 3^{rd} last character in the input. All these different choices for each substring of the output string leads to an exponential number of choices for the complete transformation. We use an efficient data structure for succinctly representing such exponential number of consistent expressions in polynomial space.

Example 2 (Duration Manipulation). An Excel user wanted to convert the "raw data" in the input column to the lower range of the corresponding "30-min interval" as shown in the output column. An expert responded by providing the following macro, which is quite unreadable and error-prone.

Input v_1	Output
0d 5h 26m	5:00
0d 4h 57m	4:30
0d 4h 27m	**4:00**
0d 3h 57m	**3:30**

```
FLOOR(TIME(MID(C1,FIND(" ",C1)+1,FIND("h",C1)- FIND(" ",C1)-1)+0,
MID(C1,FIND("h",C1)+2,FIND("m",C1)-FIND("h",C1)-2)+0,0)*24,0.5)/24
```

Our tool learns the desired transformation using only the first two examples. In this case, we first need to be able to extract the hour and minute components of the duration in input column v_1, and then perform a rounding operation on the minute part of the input to round it to the lower 30-min interval.

3 Overview of the Synthesis Approach

In this section, we define the formalism that we use in the paper for developing inductive synthesizers [8].

Domain-specific language: We develop a domain-specific language L that is expressive enough to capture the desired tasks and, at the same time, is concise for enabling efficient learning from examples.

Data structure for representing a set of expressions: The number of expressions that are consistent with a given input-output example can potentially be very large. We, therefore, develop an efficient data structure D that can succinctly represent a large number of expressions in L.

Synthesis algorithm: The synthesis algorithm `Synthesize` consists of the following two procedures:

- `GenerateStr`: The `GenerateStr` procedure learns the set of all expressions in the language L (represented using the data structure D) that are consistent with a given input-output example (σ_i, s_i). An input state σ holds values for m string variables $v_1, \ldots, v_m$ (denoting m input columns in a spreadsheet).
- `Intersect`: The `Intersect` procedure intersects two sets of expressions to compute the common set of expressions.

The synthesis algorithm `Synthesize` takes as input a set of input-output examples and generates a set of expressions in L that are consistent with them. It uses `GenerateStr` procedure to generate a set of expressions for each individual input-output example and then uses the `Intersect` procedure to intersect the corresponding sets to compute the common set of expressions.

```
Synthesize((σ1, s1), ..., (σn, sn))
  P := GenerateStr(σ1, s1);
  for i = 2 to n:
      P' := GenerateStr(σi, si);
      P := Intersect(P, P');
  return P;
```

Ranking: Since there are typically a large number of consistent expressions for each input-output example, we rank them using the Occam's razor principle that

states that smaller and simpler explanations are usually the correct ones. This enables users to provide only a few input-output examples for quick convergence to the desired transformation.

4 Number Transformations

In this section, we first describe the number transformation language L_n that can perform formatting and rounding transformations on numbers. We then describe an efficient data structure to succinctly represent a large number of expressions in L_n, and present an inductive synthesis algorithm to learn all expressions in the language that are consistent with a given set of input-output examples.

4.1 Number Transformation Language L_n

$$
\begin{array}{rcl}
\text{Expr. } e_n & := & \texttt{Dec}(u, \eta_1, f) \\
& | & \texttt{Exp}(u, \eta_1, f, \eta_2) \\
& | & \texttt{Ord}(u) \\
& | & \texttt{Word}(u) \\
& | & u \\
\text{Dec. Fmt. } f & := & (\odot, \eta) \mid \bot \\
\text{Number } u & := & v_i \\
& | & \texttt{Round}(v_i, r) \\
\text{Round Fmt. } r & := & (z, \delta, m) \\
\text{Mode } m & := & \downarrow \mid \uparrow \mid \updownarrow \\
\text{Num. Fmt. } \eta & := & (\alpha, \beta, \gamma)
\end{array}
$$

(a)

$$
\begin{aligned}
[\![\texttt{Dec}(u, \eta_1, f)]\!]\sigma &= [\![(\texttt{Int}([\![u]\!])^{\texttt{R}}, \eta_1)]\!]^{\texttt{R}}\sigma \star [\![f]\!]\sigma \\
[\![\texttt{Exp}(u, \eta_1, f, \eta_2)]\!]\sigma &= [\![(\texttt{Int}([\![u]\!])^{\texttt{R}}, \eta_1)]\!]^{\texttt{R}}\sigma \star \\
&\quad\ [\![f]\!]\sigma \star [\![(\texttt{E}([\![u]\!])^{\texttt{R}}, \eta_2)]\!]^{\texttt{R}}\sigma \\
[\![\texttt{Ord}(u)]\!]\sigma &= \texttt{numToOrd}([\![u]\!]\sigma) \\
[\![\texttt{Word}(u)]\!]\sigma &= \texttt{numToWord}([\![u]\!]\sigma) \\
[\![(\odot, \eta)]\!]\sigma &= [\![\odot]\!]\sigma \star [\![(\texttt{Frac}([\![u]\!]), \eta)]\!]\sigma \\
[\![\bot]\!]\sigma &= \epsilon \\
[\![v_i]\!]\sigma &= \sigma(v_i) \\
[\![\texttt{Round}(v_i, r)]\!]\sigma &= \texttt{RoundNumber}(\sigma(v_i), z, \delta, m) \\
&\quad\ \text{where } r = (z, \delta, m) \\
[\![(d, \eta)]\!]\sigma &= \texttt{FormatDigits}(d, \alpha, \beta, \gamma) \\
&\quad\ \text{where } \eta = (\alpha, \beta, \gamma)
\end{aligned}
$$

(b)

Fig. 1. The (a) syntax and (b) semantics of the number transformation language L_n. The variable v_i denotes an input number variable, z, δ, α, β, and γ are integer constants, and $\star$ denotes the concatenation operation.

The syntax of the number transformation language L_n is shown in Figure 1(a). The top-level expression e_n of the language denotes a number formatting expression of one of the following forms:

- $\texttt{Dec}(u, \eta_1, f)$: formats the number u in decimal form (e.g. `1.23`), where η_1 denotes the number format for the integer part of u ($\texttt{Int}(u)$), and f represents the optional format consisting of the decimal separator and the number format for the fractional part ($\texttt{Frac}(u)$).
- $\texttt{Exp}(u, \eta_1, f, \eta_2)$: formats the number u in exponential form (e.g. `1.23E+2`). It consists of an additional number format η_2 as compared to the decimal format expression, which denotes the number format of the exponent digits of u.

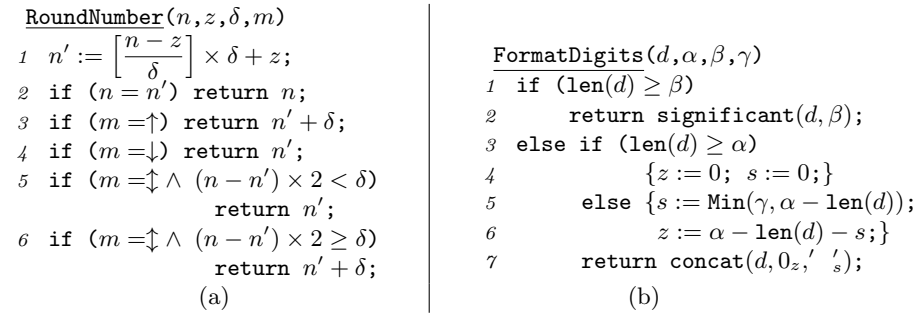


Fig. 2. The functions (a) `RoundNumber` for rounding numbers and (b) `FormatDigits` for formatting a digit string

- `Ord`(u): formats the number u in *ordinal* form, e.g. it formats the number 4 to its ordinal form $4^{\texttt{th}}$.
- `Word`(u): formats the number u in *word* form, e.g. it formats the number 4 to its word form `four`.

The number u can either be an input number variable v_i or a number obtained after performing a rounding transformation on an input number. A rounding transformation `Round`(v_i, z, δ, m) performs the rounding of number present in v_i based on its rounding format (z, δ, m), where z denotes the *zero* of the rounding interval, δ denotes the interval size of the rounding interval, and m denotes one of the rounding mode from the set of modes {upper($\uparrow$), lower($\downarrow$), nearest($\updownarrow$)}.

We define a *digit string* d to be a sequence of digits with trailing whitespaces. A number format η of a digit string d is defined by a 3-tuple (α, β, γ), where α denotes the minimum number of significant digits and trailing whitespaces of d in the output string, β denotes the maximum number of significant digits of d in the output string, and γ denotes the maximum number of trailing whitespaces in the output string. A number format, thus, maintains the invariant: $\gamma \leq \alpha \leq \beta$.

The semantics of language L_n is shown in Figure 1(b). A digit string d is formatted with a number format (α, β, γ) using the `FormatDigits` function shown in Figure 2(b). The `FormatDigits` function returns the first β digits of the digit string d (with appropriate rounding) if the length of d is greater than the maximum number of significant digits β to be printed. If the length of d is lesser than β but greater than the minimum number of significant digits α to be printed, it returns the digits itself. Finally, if the length of d is less than α, it appends the digit string with appropriate number of zeros (z) and whitespaces (s) as computed in Lines 5 and 6. The semantics of the rounding transformation is to perform the appropriate rounding of number denoted by v_i using the `RoundNumber` function shown in Figure 2(a). The function computes a number n' which lies on the number line defined by zero z with unit separation δ as shown in Figure 3. It returns the value n' or $(n' + \delta)$ based on the rounding mode m and the distance between n and n' as described in Figure 2(a).

The semantics of a decimal form formatting expression on a number u is to concatenate the reverse of the string obtained by formatting the reverse of

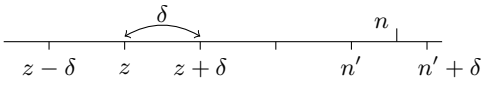


Fig. 3. The `RoundNumber` function rounding-off number n to n' or $n' + \delta$

integral part $\texttt{Int}(u)$ with the string obtained from the decimal format f. Since the `FormatDigits` function adds only trailing zeros and whitespaces to format a digit string, the formatting of the integer part of u is performed on its reverse digit string and the resulting formatted string is reversed again before performing the concatenation. The semantics of decimal format f is to concatenate the decimal separator $\odot$ with the string obtained by formatting the fractional part $\texttt{Frac}(u)$. The semantics of exponential form formatting expression is similar to that of the decimal form formatting expression and the semantics of ordinal form and word form formatting expressions is to simply convert the number u into its corresponding ordinal form and word form respectively.

We now present some examples taken from various help forums that can be represented in the number transformation language L_n.

Example 3. A python programmer posted a query on the `StackOverflow` forum after struggling to print double values from an array of doubles (of different lengths) such that the decimal point for each value is aligned consistently across different columns. He posted an example of the desired formatting as shown on the right. He also wanted to print a single 0 after the decimal if the double value had no decimal part.

Input v_1	Output
3264.28	3264.28
53.5645	53.5645
235	235.0
5.23	**5.23**
345.213	**345.213**
3857.82	**3857.82**
536	**536.0**

The programmer started the post saying *"This should be easy"*. An expert replied that after a thorough investigation, he couldn't find a way to perform this task without some post-processing. The expert provided the following python snippet that pads spaces to the left and zeros to the right of the decimal, and then removes trailing zeros:

```
ut0 = re.compile(r'(\d)0+$')
thelist = textwrap.dedent(
     '\n'.join(ut0.sub(r'\1', "%20f" % x) for x in a)).splitlines()
print '\n'.join(thelist)
```

This formatting transformation can be represented in L_n as $\texttt{Dec}(v_1, \eta_1, (\text{“.”}, \eta_2))$, where $\eta_1 \equiv (4, \infty, 4)$ and $\eta_2 \equiv (4, \infty, 3)$.

Example 4. This is an interesting post taken from a help forum where the user initially posted that she wanted to round numbers in an excel column to nearest 45 or 95, but the examples later showed that she actually wanted to round it to *upper* 45 or 95.

Input v_1	Output
11	45
32	45
46	95
1865	**1895**

Some of the solutions suggested by experts were:

```
=Min(Roundup(A1/45,0)*45,Roundup(A1/95,0)*95)
=CEILING(A1+5,50)-5
=A1-MOD(A1,100)+IF(MOD(A1,100)>45,95,45)
```

This rounding transformation can be expressed in our language as:
$\texttt{Dec}(\texttt{Round}(v_1, (45, 50, \uparrow)), (0, \infty, 0), \perp)$.

4.2 Data structure for a set of expressions in L_n

Figure 4 describes the syntax and semantics of the data structure for succinctly representing a set of expressions from language L_n. The expressions $\tilde{e}_n$ are now associated with a set of numbers $\tilde{u}$ and a set of number formats $\tilde{\eta}$. We represent the set of numbers obtained after performing rounding transformation in two ways: $\texttt{Round}(v_i, \tilde{r})$ and $\texttt{Round}(v_i, n_p)$, which we describe in more detail in section 4.3. The set of number formats $\tilde{\eta}$ are represented using a 3-tuple (i_1, i_2, i_3), where i_1, i_2 and i_3 denote a set of values of α, β and γ respectively using an interval domain. This representation lets us represent $O(n^3)$ number of number format expressions in $O(1)$ space, where n denotes the length of each interval.

The semantics of evaluating the set of rounding transformations $\texttt{Round}(v_i, \tilde{r})$ is to return the set of results of performing rounding transformation on v_i for all rounding formats in the set $\tilde{r}$. The expression $\texttt{Round}(v_i, (n_1, n_1'))$ represents an infinite number of rounding transformations (as there exists an infinite number of rounding formats that conform to the rounding transformation $n_1 \rightarrow n_1'$). For evaluating this expression, we select one conforming rounding format with $z = 0$, $\delta = n_1'$ and an appropriate m as shown in the figure. The evaluation of a set of format strings $\tilde{\eta} = (i_1, i_2, i_3)$ on a digit string d returns a set of values, one for each possible combination of $\alpha \in i_1$, $\beta \in i_2$ and $\gamma \in i_3$. Similarly, we obtain a set of values from the evaluation of expression $\tilde{e}_n$.

4.3 Synthesis Algorithm

Procedure $\texttt{GenerateStr}_n$: The algorithm `GenDFmt` in Figure 5 takes as input two digit sequences d_1 and d_2, and computes the set of all number formats $\tilde{\eta}$ that are consistent for formatting d_1 to d_2. The algorithm first converts the digit sequence d_1 to its canonical form d_1' by removing trailing zeros and whitespaces from d_1. It then compares the lengths $\texttt{l}_1$ of d_1' and $\texttt{l}_2$ of d_2. If $\texttt{l}_1$ is greater than $\texttt{l}_2$, then we can be sure that the digits got truncated and can therefore set the interval for i_2 (the maximum number of significant digits) to be $[\texttt{l}_2, \texttt{l}_2]$. The intervals for α and γ are set to $[0, \texttt{l}_2]$ because of the number format invariant. On the other hand if $\texttt{l}_1$ is smaller than $\texttt{l}_2$, we can be sure that the least number of significant digits need to be $\texttt{l}_2$, *i.e.* we can set the interval i_1 to be $[\texttt{l}_2, \texttt{l}_2]$. Also, we can set the interval i_2 to $[\texttt{l}_2, \infty]$ because of the number format invariant. For interval i_3, we either set it to $[\xi, \xi]$ (when $\texttt{l}_2 - \xi \neq \texttt{l}_1$) or $[\xi, \texttt{l}_2]$ (when $\texttt{l}_2 - \xi = \texttt{l}_1$) where ξ denotes the number of trailing spaces in d_2. In the former case, we can be sure about the exact number of trailing whitespaces to be printed.

$\tilde{e}_n$	:=	$\texttt{Dec}(\tilde{u}, \tilde{\eta}_1, \tilde{f})$	$[\![\texttt{Dec}(\tilde{u}, \tilde{\eta}_1, \tilde{f})]\!] = \{\texttt{Dec}(u, \eta_1, f) \mid u \in \tilde{u}, \eta_1 \in \tilde{\eta}_1, f \in \tilde{f}\}$
	|	$\texttt{Exp}(\tilde{u}, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2)$	$[\![\texttt{Exp}(\tilde{u}, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2)]\!] = \{\texttt{Exp}(u, \eta_1, f, \eta_2) \mid u \in \tilde{u}, \eta_1 \in \tilde{\eta}_1,$
	|	$\texttt{Ord}(\tilde{u})$	$f \in \tilde{f}, \eta_2 \in \tilde{\eta}_2\}$
	|	$\texttt{Word}(\tilde{u})$	$[\![\texttt{Ord}(\tilde{u})]\!] = \{\texttt{Ord}(u) \mid u \in \tilde{u}\}$
	|	$\tilde{u}$	$[\![\texttt{Word}(\tilde{u})]\!] = \{\texttt{Word}(u) \mid u \in \tilde{u}\}$
$\tilde{f}$	:=	$(\odot, \tilde{\eta}) \mid \perp$	$[\![(\odot, \tilde{f})]\!] = \{(\odot, f) \mid f \in \tilde{f}\}$
$\tilde{u}$	:=	v_i	$[\![\perp]\!] = \epsilon$
	|	$\texttt{Round}(v_i, \tilde{r})$	$[\![v_i]\!] = \{v_i\}$
	|	$\texttt{Round}(v_i, n_p)$	$[\![\texttt{Round}(v_i, \tilde{r})]\!] = \{\texttt{Round}(v_i, (z, \delta, m)) \mid (z, \delta, m) \in \tilde{r}\}$
Pair n_p	:=	(n_1, n_1')	$[\![\texttt{Round}(v_i, n_p)]\!] = \{\texttt{Round}(v_i, (0, n_1', m)) \mid n_p \equiv (n_1, n_1'),$
$\tilde{\eta}$	:=	(i_1, i_2, i_3)	$\texttt{if } (n_1 \leq n_1') \ m \equiv \uparrow \texttt{else } m \equiv \downarrow\}$
Interval i	:=	(l, h)	$[\![(d, (i_1, i_2, i_3))]\!] = \{(d, \alpha, \beta, \gamma) \mid \alpha \in i_1, \beta \in i_2, \gamma \in i_3\}$
		(a)	(b)

Fig. 4. The (a) syntax and (b) semantics of a data structure for succinctly representing a set of expressions from language L_n.

The $\texttt{GenerateStr}_n$ algorithm in Figure 5 learns the set of all expressions in L_n that are consistent with a given input-output example. The algorithm searches over all input variables v_i to find the inputs from which the output number n' can be obtained. It first converts the numbers $\sigma(v_i)$ and n' to their *canonical forms* n_{C} and n'_{C} respectively in Line 3. We define canonical form of a number to be its decimal value. If the two canonical forms n_{C} and n'_{C} are not equal, the algorithm tries to learn a rounding transformation such that n_{C} can be rounded to n'_{C}. We note that there is not enough information present in one input-output example pair to learn the exact rounding format as there exists an infinite family of such formats that are consistent. Therefore, we represent such rounding formats symbolically using the input-output example pair $(n_{\text{C}}, n'_{\text{C}})$, which gets concretized by the `Intersect` method in Figure 6. The algorithm then normalizes the number $\sigma(u)$ with respect to n' using the `Normalize` method in Line 6 to obtain $n = (n_i, n_f, n_e)$ such that both n and n' are of the same form. For decimal and exponential forms, it learns a set of number formats $\tilde{\eta}$ for each of its constituent digit strings from the pairs $(n_i^{\text{R}}, n_i'^{\text{R}})$, (n_f, n'_f), and $(n_e^{\text{R}}, n_e'^{\text{R}})$ where n_i^{R} denotes the reverse of digit string n_i. As noted earlier, we need to learn the number format on the reversed digit strings for integer and exponential parts. For ordinal and word type numbers, it simply returns the expressions to compute ordinal and word forms of the corresponding input number respectively.

Procedure $\texttt{Intersect}_n$: The $\texttt{Intersect}_n$ procedure for intersecting two sets of L_{n} expressions is described as a set of rules in Figure 6. The procedure computes the intersection of sets of expressions by recursively computing the intersection of their corresponding sets of sub-expressions. We describe below the four cases of

```
GenDFmt(d1: inp digits, d2: out digits)
 1  d1' := RemoveTrailingZerosSpaces(d1);
 2  l1 := len(d1'); l2 := len(d2);
 3  ξ := numTrailingSpaces(d2);
 4  if (l1 > l2)
 5        (i1, i2, i3) := ([0, l2], [l2, l2], [0, l2]);
 6  else if (l1 < l2) {
 7           i1 := [l2, l2]; i2 := [l2, ∞];
 8           if(l2 − ξ = l1) i3 := [ξ, l2];
 9           else i3 := [ξ, ξ];}
10  else (i1, i2, i3) := ([0, l2], [l2, ∞], [0, l2]);
11  return η̃(i1, i2, i3);

Normalize(n: inp number, n': out number)
  n1 = n = (ni, nf, ne);
  if(Type(n) = ExpNum ∧ Type(n') ≠ ExpNum)
     n1 := n × 10^ne ;
  if(Type(n) ≠ ExpNum ∧ Type(n') = ExpNum)
     {n' = (ni', nf', ne'); n1 := n/10^ne' ;}
  return n1 ;
```

```
GenerateStr_n(σ: inp state, n': out number)
 1  Sn := ∅;
 2  foreach input variable vi:
 3     nc = Canonical(σ(vi)); nc' = Canonical(n');
 4     if (nc ≠ nc') u := Round(vi, (nc, nc'));
 5     else u := vi;
 6     (ni, nf, ne) := Normalize(σ(u), n');
 7     match n' with
 8       DecNum(ni', nf', ⊙) →
 9         η̃1 := GenDFmt(ni^R, ni'^R);
10         if (⊙ = ϵ) Sn := Sn ∪ Dec(u, η1, ⊥);
11         else {η̃2 := GenDFmt(nf, nf');
12                 Sn := Sn ∪ Dec(u, η̃1, ⊙, η̃2);}
13       ExpNum(ni', nf', ne', ⊙) →
14         η̃1 := GenDFmt(ni^R, ni'^R);
15         η̃3 := GenDFmt(ne^R, ne'^R);
16         if (⊙ = ϵ) Sn := Sn ∪ Exp(u, η̃1, ⊥, η̃3);
17         else { η̃2 := GenDFmt(nf, nf');
18                Sn := Sn ∪ Exp(u, η̃1, ⊙, η̃2, η̃3);}
19       OrdNum(ni') →
20         Sn := Sn ∪ Ord(u);
21       WordNum(ni') →
22         Sn := Sn ∪ Word(u);
23  return Sn;
```

Fig. 5. The $\texttt{GenerateStr}_n$ procedure for generating the set of all expressions in language L_{n} that are consistent with the given set of input-output examples

intersecting rounding transformation expressions. The first case is of intersecting a finite rounding format set $\tilde{r}$ with another finite set $\tilde{r}'$. The other two cases intersect a finite set $\tilde{r}$ with an input-output pair n_p, which is performed by selecting a subset of the finite set of rounding formats that are consistent with the pair n_p. The final case of intersecting two input-output pairs to obtain a finite set of rounding formats is performed using the `IntersectPair` algorithm shown in Figure 7.

Consider the example of rounding numbers to nearest 45 or 95 for which we have the following two examples: $32 \rightarrow 45$ and $81 \rightarrow 95$. Our goal is to learn the rounding format (z, δ, m) that can perform the desired rounding transformation. We represent the infinite family of formats that satisfy the rounding constraint for each example as individual pairs $(32, 45)$ and $(81, 95)$ respectively. When we intersect these pairs, we can assign z to be 45 without loss of generality. We then compute all divisors $\tilde{\delta}$ of $95-45 = 50$. With the constraint that $\delta \geq$ ($\texttt{Max}(45-32, 95-81) = 14$), we finally arrive at the set $\tilde{\delta} = \{25, 50\}$. The rounding modes m are appropriately learned as shown in Figure 7. For decimal numbers, we compute the divisors by first scaling them appropriately and then re-scaling them back for learning the rounding formats. In our data structure, we do not store all divisors

```
IntersectPair((n1, n1'),(n2, n2'))
  z := n1';
  δ̃ := Divisors(‖n2' − n1'‖);
  S := ∅;
  foreach δ ∈ δ̃:
    if(δ ≥ Max(‖n1 − n1'‖, ‖n2 − n2'‖))
      if(2 × Max(‖n1 − n1'‖, ‖n2 − n2'‖) ≤ δ)
        S := S ∪ (z, δ, ↕);
      if(n1 > n1' ∧ n2 > n2')
        S := S ∪ (z, δ, ↓);
      if(n1 < n1' ∧ n2 < n2')
        S := S ∪ (z, δ, ↑);
  return S;
```

Fig. 7. Intersection of `Round` expressions

$$
\begin{aligned}
\texttt{Intersect}_n(\texttt{Dec}(\tilde{u},\tilde{\eta}_1,\tilde{f}),\texttt{Dec}(\tilde{u}',\tilde{\eta}_1',\tilde{f}')) &= \texttt{Dec}(\texttt{Intersect}_n(\tilde{u},\tilde{u}'),\texttt{Intersect}_n(\tilde{\eta}_1,\tilde{\eta}_1'),\\
&\qquad \texttt{Intersect}_n(\tilde{f},\tilde{f}'))\\
\texttt{Intersect}_n(\texttt{Exp}(\tilde{u},\tilde{\eta}_1,\tilde{f},\tilde{\eta}_2),\texttt{Exp}(\tilde{u},\tilde{\eta}_1',\tilde{f}',\tilde{\eta}_2')) &= \texttt{Exp}(\texttt{Intersect}_n(\tilde{u},\tilde{u}'),\texttt{Intersect}_n(\tilde{\eta}_1,\tilde{\eta}_1'),\\
&\qquad \texttt{Intersect}_n(\tilde{f},\tilde{f}'),\texttt{Intersect}_n(\tilde{\eta}_2,\tilde{\eta}_2'))\\
\texttt{Intersect}_n(\texttt{Ord}(\tilde{u}),\texttt{Ord}(\tilde{u}')) &= \texttt{Ord}(\texttt{Intersect}_n(\tilde{u},\tilde{u}'))\\
\texttt{Intersect}_n(\texttt{Word}(\tilde{u}),\texttt{Word}(\tilde{u}')) &= \texttt{Word}(\texttt{Intersect}_n(\tilde{u},\tilde{u}'))\\
\texttt{Intersect}_n(v_i,v_i) &= v_i\\
\texttt{Intersect}_n((\odot,\tilde{\eta}),(\odot',\tilde{\eta}')) &= (\texttt{Intersect}_n(\odot,\odot'),\texttt{Intersect}_n(\tilde{\eta},\tilde{\eta}'))\\
\texttt{Intersect}_n(\texttt{Round}(v_i,\tilde{r}),\texttt{Round}(v_i,\tilde{r}')) &= \texttt{Round}(v_i,\texttt{Intersect}_n(\tilde{r},\tilde{r}'))\\
\texttt{Intersect}_n(\texttt{Round}(v_i,\tilde{r}),\texttt{Round}(v_i,n_p)) &= \texttt{Round}(v_i,\texttt{Intersect}_n(\tilde{r},n_p))\\
\texttt{Intersect}_n(\texttt{Round}(v_i,n_p),\texttt{Round}(v_i,\tilde{r})) &= \texttt{Round}(v_i,\texttt{Intersect}_n(n_p,\tilde{r}))\\
\texttt{Intersect}_n(\texttt{Round}(v_i,n_p),\texttt{Round}(v_i,n_p')) &= \texttt{Round}(v_i,\texttt{IntersectPair}(n_p,n_p'))\\
\texttt{Intersect}_n((i_1,i_2,i_3),(i_1',i_2',i_3')) &= (\texttt{Intersect}_n(i_1,i_1'),\texttt{Intersect}_n(i_2,i_2'),\\
&\qquad \texttt{Intersect}_n(i_3,i_3'))\\
\texttt{Intersect}_n((l,h),(l',h')) &= (\texttt{Max}(l,l'),\texttt{Min}(h,h'))
\end{aligned}
$$

Fig. 6. The $\texttt{Intersect}_n$ function for intersecting sets of expressions from language L_n. The $\texttt{Intersect}_n$ function returns ϕ in all other case not covered above.

explicitly as this set might become too large for big numbers. We observe that we only need to store the greatest and least divisors amongst them, and then we can intersect two such sets efficiently by computing the `gcd` of the two corresponding greatest divisors and the `lcm` of the two corresponding least divisors.

Ranking: We rank higher the lower value for α in the interval i_1 (to prefer lesser trailing zeros and whitespaces), the higher value of β in i_2 (to minimize un-necessary number truncation), the lower value of γ in i_3 (to prefer trailing zeros more than trailing whitespaces), and the greatest divisor in the set of divisors $\tilde{\delta}$ of the rounding format (to minimize the length of rounding intervals). We rank expressions consisting of rounding transformations lower than the ones that consist of only number formatting expressions.

Theorem 1 (Correctness of Learning Algorithm for L_n).
(a) The procedure $\texttt{GenerateStr}_n$ is sound and complete. The complexity of $\texttt{GenerateStr}_n$ is $O(|s|)$, where $|s|$ denotes the length of the output string.
(b) The procedure $\texttt{Intersect}_n$ is sound and complete.

Example 5. Figure 8 shows a range of number formatting transformations and presents the format strings that are required to be provided in Excel, .NET, Python and C, as well as the format expressions that are synthesized by our algorithm. An `N.A.` entry denotes that the corresponding formatting task cannot be done in the corresponding language.

Formatting of Doubles				
Input String	**Output String**	**Excel/C# Format String**	**Python/C Format String**	**Synthesized format** $\texttt{Dec}(u, \eta_1, (\text{“.”}, \eta_2))$ or $\texttt{Exp}(u, \eta_1, (\text{“.”}, \eta_2), \eta_3)$
123.4567 123.4	123.46 123.40	#.00	.2f	$\eta_1 \equiv ([0,3],[3,\infty],[0,3])$ $\eta_2 \equiv ([2,2],[2,2],[0,0])$
123.4567 123.4	123.46 123.4	#.##	N.A.	$\eta_1 \equiv ([0,3],[3,\infty],[0,3])$ $\eta_2 \equiv ([0,1],[2,2],[0,1])$
123.4567 3.4	123.46 03.40	00.00	05.2f	$\eta_1 \equiv ([2,2],[3,\infty],[0,0])$ $\eta_2 \equiv ([2,2],[2,2],[0,0])$
123.4567 3.4	123.46 03.4	00.##	N.A.	$\eta_1 \equiv ([2,2],[3,\infty],[0,0])$ $\eta_2 \equiv ([0,1],[2,2],[0,1])$
9723.00 0.823	9.723E+03 8.23E-01	#.### E 00	N.A.	$\eta_1 \equiv ([0,1],[1,\infty],[0,1])$ $\eta_2 \equiv ([0,3],[3,\infty],[0,3])$ $\eta_3 \equiv ([2,2],[2,\infty],[0,0])$
243 12	00243 00012	00000	05d	$\eta_1 \equiv ([5,5],[5,\infty],[0,0])$
1.2 18	1.2␣ 18.␣␣	#.??	N.A.	$\eta_1 \equiv ([0,1],[2,\infty],[0,1])$ $\eta_2 \equiv ([2,2],[2,\infty],[2,2])$
1.2 18	␣␣1.2␣␣ ␣18.␣␣␣	???.???	N.A.	$\eta_1 \equiv ([3,3],[3,\infty],[2,3])$ $\eta_2 \equiv ([3,3],[3,\infty],[3,3])$
1.2 18	␣␣1.20␣ ␣18.00␣	???.00?	N.A.	$\eta_1 \equiv ([3,3],[3,\infty],[2,3])$ $\eta_2 \equiv ([3,3],[3,\infty],[1,1])$

Fig. 8. We compare the custom number format strings required to perform formatting of doubles in Excel/C# and Python/C languages. An N.A. entry in a format string denotes that the corresponding formatting is not possible using format strings only. The last column presents the corresponding L_n expressions (␣ denotes whitespaces).

5 Combining Number Transformations with Syntactic String Transformations

In this section, we present the combination of number transformation language L_n with the syntactic string transformation language L_s [6] to obtain the combined language L_c, which can model transformations on strings that contain numbers as substrings. We first present a brief background description of the syntactic string transformation language and then present the combined language L_c. We also present an inductive synthesis algorithm for L_c obtained by combining the inductive synthesis algorithms for L_n and L_s respectively.

Syntactic String Transformation Language L_s (Background) Gulwani [6] introduced an expression language for performing syntactic string transformations. We reproduce here a small subset of (the rules of) that language and call it L_s (with e_s being the top-level symbol) as shown in Figure 9. The formal semantics of L_s can be found in [6]. For completeness, we briefly describe some key aspects of this language. The top-level expression e_s is either an atomic expression f or is obtained by concatenating atomic expressions $\text{f}_1,\ldots,\text{f}_n$ using

$$
\begin{aligned}
e_s &:= \texttt{Concatenate}(\mathrm{f}_1, \ldots, \mathrm{f}_n) \mid \mathrm{f} \\
\text{Atomic expr } \mathrm{f} &:= \texttt{ConstStr}(s) \mid v_i \mid \texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2) \\
\text{Position } \mathrm{p} &:= k \mid \texttt{pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c}) \\
\text{Integer expr } \mathrm{c} &:= k \mid k_1 w + k_2 \\
\text{Regular expr } \mathrm{r} &:= \epsilon \mid \mathrm{T} \mid \texttt{TokenSeq}(\mathrm{T}_1, \ldots, \mathrm{T}_n)
\end{aligned}
$$

Fig. 9. The syntax of syntactic string transformation language L_s.

the `Concatenate` constructor. Each atomic expression f can either be a constant string $\texttt{ConstStr}(s)$, an input string variable v_i, or a substring of some input string v_i. The substring expression $\texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2)$ is defined partly by two *position expressions* p_1 and p_2, each of which implicitly refers to the (subject) string v_i and must evaluate to a position within the string v_i. (A string with ℓ characters has $\ell + 1$ positions, numbered from 0 to ℓ starting from left.) $\texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2)$ is the substring of string v_i in between positions p_1 and p_2. A position expression represented by a non-negative constant k denotes the k^{th} position in the string. For a negative constant k, it denotes the $(\ell+1+k)^{th}$ position in the string, where $\ell = \texttt{Length}(s)$. $\texttt{pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c})$ is another position expression, where r_1 and r_2 are regular expressions and integer expression c evaluates to a non-zero integer. $\texttt{pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c})$ evaluates to a position t in the subject string s such that r_1 matches some suffix of $s[0 : t]$, and r_2 matches some prefix of $s[t : \ell]$, where $\ell = \texttt{Length}(s)$. Furthermore, if c is positive (negative), then t is the $|\mathrm{c}|^{th}$ such match starting from the left side (right side). We use the expression $s[t_1 : t_2]$ to denote the substring of s between positions t_1 and t_2. We use the notation $\texttt{SubStr2}(v_i, \mathrm{r}, \mathrm{c})$ as an abbreviation to denote the c^{th} occurrence of regular expression r in v_i, i.e., $\texttt{SubStr}(v_i, \texttt{pos}(\epsilon, \mathrm{r}, \mathrm{c}), \texttt{pos}(\mathrm{r}, \epsilon, \mathrm{c}))$.

A regular expression r is either ϵ (which matches the empty string, and therefore can match at any position of any string), a token T, or a token sequence $\texttt{TokenSeq}(\mathrm{T}_1, \ldots, \mathrm{T}_n)$. The tokens T range over a finite extensible set and typically correspond to character classes and special characters. For example, tokens `CapitalTok`, `NumTok`, and `WordTok` match a nonempty sequence of uppercase alphabetic characters, numeric digits, and alphanumeric characters respectively.

A `Dag` based data structure is used to succinctly represent a set of L_s expressions. The `Dag` structure consists of a node corresponding to each position in the output string s, and a map W maps an edge between node i and node j to the set of all L_c expressions that can compute the substring $s[i..j]$. This representation enables sharing of common subexpressions amongst the set of expressions and represents an exponential number of expressions using polynomial space.

Example 6. An Excel user wanted to modify the delimiter in dates present in a column from "/" to "-", and gave the following input-output example "08/15/2010" → "08-15-2010". An expression in L_s that can perform this transformation is: $\texttt{Concatenate}(\mathrm{f}_1, \texttt{ConstStr}(\text{“}-\text{”}), \mathrm{f}_2, \texttt{ConstStr}(\text{“}-\text{”}), \mathrm{f}_3)$, where $\mathrm{f}_1 \equiv \texttt{SubStr2}(v_1, \texttt{NumTok}, 1)$, $\mathrm{f}_2 \equiv \texttt{SubStr2}(v_1, \texttt{NumTok}, 2)$, and $\mathrm{f}_3 \equiv \texttt{SubStr2}(v_1,$

NumTok, 3). This expression constructs the output string by concatenating the first, second, and third numbers of input string with constant strings "-".

5.1 The Combination Language L_c

$$
\begin{aligned}
\mathrm{f} &:= \texttt{ConstStr}(s) \mid v_i \\
&\mid \texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2) \mid e_n \\
u &:= g \mid \texttt{Round}(g, r) \\
g &:= v_i \mid \texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2)
\end{aligned}
$$

The grammar rules R_c for the combined language L_c are obtained by taking the union of the rules for the two languages R_n and R_s with the top-level rule e_s. The modified rules are shown in the figure on the right. The combined language consists of an additional expression rule g that corresponds to either some input column v_i or a substring of some input column. This expression g is then passed over to the number variable expression u for performing number transformations on it. This rule enables the combined language to perform number transformations on substrings of input strings. The top-level expression of the number language e_n is added to the atomic expr f of the string language. This enables number transformation expressions to be present on the Dag edges together with the syntactic string transformation expressions.

The transformation in Example 1 is represented in L_c as: $\texttt{Concatenate}(\mathrm{f}_1, \texttt{ConstStr}(\texttt{"/"}), \mathrm{f}_2, \texttt{ConstStr}(\texttt{"/"}), \mathrm{f}_3)$, where $\mathrm{f}_1 \equiv \texttt{Dec}(g_1, (2, \infty, 0), \perp)$, $g_1 \equiv \texttt{SubStr}(v_1, 1, -7)$, $\mathrm{f}_2 \equiv \texttt{SubStr}(v_1, -7, -5)$, and $\mathrm{f}_3 \equiv \texttt{SubStr}(v_1, -5, -1)$. The transformation in Example 2 is represented as: $\texttt{Concatenate}(\mathrm{f}_1, \texttt{":"}, \mathrm{f}_2)$, where $\mathrm{f}_1 \equiv \texttt{SubStr2}(v_1, \texttt{NumTok}, 2)$, $\mathrm{f}_2 \equiv \texttt{Dec}(u_1, (2, \infty, 0), \perp)$, and $u_1 \equiv \texttt{Round}(\texttt{SubStr2}(v_1, \texttt{NumTok}, 3), (0, 30, \downarrow))$.

5.2 Data structure for representing a set of expressions in L_c

Let $\tilde{R}_n$ and $\tilde{R}_s$ denote the set of grammar rules for the data structures that represent a set of expressions in L_n and L_s respectively. We obtain the grammar rules $\tilde{R}_c$ for succinctly representing a set of expressions of L_c by taking the union of the two rule sets $\tilde{R}_n$ and $\tilde{R}_s$ with the updated rules as shown in Figure 10(a). The updated rules have expected semantics and can be defined as in Figure 4(b).

$$
\begin{aligned}
\tilde{\mathrm{f}} &:= \cdots \mid \tilde{e}_n \\
\tilde{u} &:= \tilde{g} \mid \texttt{Round}(\tilde{g}, \tilde{r}) \\
\tilde{g} &:= v_i \mid \texttt{SubStr}(v_i, \tilde{\mathrm{p}}_1, \tilde{\mathrm{p}}_2)
\end{aligned}
$$

(a)

$\underline{\texttt{GenerateStr}_c}(\sigma\texttt{: Inp, } s\texttt{: Out})$
$\tilde{\eta} = \{0, \cdots, \texttt{Length}(s)\}$;
$\eta^s = 0$;
$\eta^t = \texttt{Length}(s)$;
$\tilde{\xi} = \{\langle i, j\rangle \mid 0 \le i < j < \texttt{Length}(s)\}$;
foreach substring s[i..j] of s:
  $W[\langle i, j\rangle]$ = $\texttt{ConstStr}(s\texttt{[i..j]})$
    $\cup$ $\texttt{GenerateStr}_s(\sigma, s\texttt{[i..j]})$
    $\cup$ $\texttt{GenerateStr}'_n(\sigma, s\texttt{[i..j]})$
return $\texttt{Dag}(\tilde{\eta}, \eta^s, \eta^t, \tilde{\xi}, W)$;

(b)

Fig. 10. (a) The data structure and (b) the $\texttt{GenerateStr}_c$ procedure for L_c expressions.

5.3 Synthesis Algorithm

Procedure $\texttt{GenerateStr}_c$**:**

We first make the following two modifications in the $\texttt{GenerateStr}_n$ procedure to obtain $\texttt{GenerateStr}'_n$ procedure. The first modification is that we now search over all substrings of input string variables v_i instead of just v_i in Line 2 in Figure 5. This lets us model transformations where number transformations are required to be performed on substrings of input strings. The second modification is that we replace each occurence of v_i by $\texttt{GenerateStr}_s(\sigma, v_i)$ inside the loop body. This lets us learn the syntactic string program to extract the corresponding substring from the input string variables. The $\texttt{GenerateStr}_c$ procedure for the combined language is shown in the Figure 10(b). The procedure first creates a `Dag` of $(\texttt{Length}(s) + 1)$ number of nodes with start node $\eta^s = 0$ and target node $\eta^t = \texttt{Length}(s)$. The procedure iterates over all substrings `s[i..j]` of the output string `s`, and adds a constant string expression, a set of substring expressions ($\texttt{GenerateStr}_s$) and a set of number transformation expressions ($\texttt{GenerateStr}'_n$) that can generate the substring `s[i..j]` from the input state σ. These expressions are then added to a map $W[\langle i, j\rangle]$, where W maps each edge $\langle i, j\rangle$ of the dag to a set of expressions in L_c that can generate the corresponding substring `s[i..j]`.

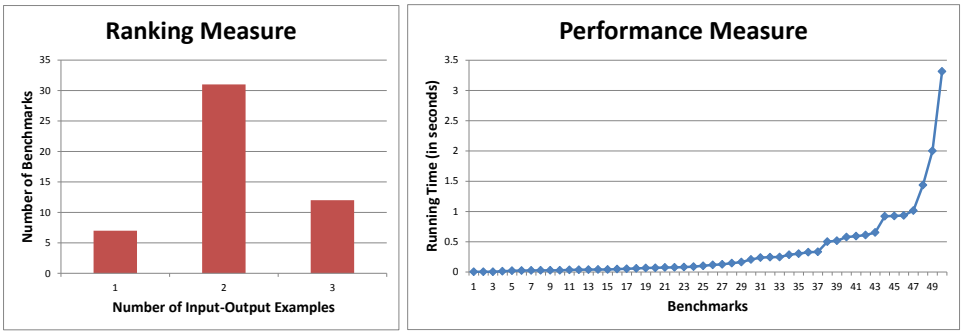
Procedure $\texttt{Intersect}_c$**:** The rules for $\texttt{Intersect}_c$ procedure for intersecting sets of expressions in L_c are obtained by taking the union of intersection rules of $\texttt{Intersect}_n$ and $\texttt{Intersect}_s$ procedures together with corresponding intersection rules for the updated and new rules.

Ranking: The ranking scheme of the combined language L_c is obtained by combining the ranking schemes of languages L_n and L_s. In addition, we prefer substring expressions corresponding to longer input substrings that can be formatted or rounded to obtain the output number string.

Theorem 2 (Correctness of Learning Algorithm for combined language). *(a) The procedure* $\texttt{GenerateStr}_c$ *is sound and complete with complexity* $O(|s|^3 l^2)$*, where* $|s|$ *denotes the length of the output string and* l *denotes the length of the longest input string.*
(b) The procedure $\texttt{Intersect}_c$ *is sound and complete.*

6 Experiments

We have implemented our algorithms in C# as an add-in to the Microsoft Excel spreadsheet system. The user provides input-output examples using an Excel table with a set of input and output columns. Our tool learns the expressions in L_c for each output column separately and executes the learned set of expressions on the remaining entries in the input columns to generate their corresponding outputs. We have evaluated our implementation on over 50 benchmarks obtained from various help forums, mailing lists, books and the Excel product team. More details about the benchmark problems can be found in [22].



(a) (b)

Fig. 11. (a) Number of examples required and (b) the running time of algorithm (in seconds) to learn the desired transformation

The results of our evaluation are shown in Figure 11. The experiments were run on an Intel Core-i7 1.87 Ghz CPU with 4GB of RAM. We evaluate our algorithm on the following two dimensions:

Ranking: Figure 11(a) shows the number of input-output examples required by our tool to learn the desired transformation. All benchmarks required at most 3 examples, with majority (76%) taking only 2 examples to learn the desired transformation. We ran this experiment in an automated counter-example guided manner such that given a set of input-output examples, we learned the transformations using a subset of the examples (training set). The tool iteratively added the failing test examples to the training set until the synthesized transformation conformed to all the remaining examples.

Performance: The running time of our tool on the benchmarks is shown in Figure 11(b). Our tool took at most 3.5 seconds each to learn the desired transformation for the benchmarks, with majority (94%) taking less than a second.

7 Related Work

The closest related work to ours is our previous work on synthesizing syntactic string transformations [6]. The algorithm presented in that work assumes strings to be a sequence of characters and can only perform concatenation of input substrings and constant strings to generate the desired output string. None of our benchmarks presented in this paper can be synthesized by that algorithm as it lacks reasoning about the semantics of numbers present in the input string.

There has been a lot of work in the HCI community for automating end-user tasks. Topes [20] system lets users create abstractions (called topes) for different data present in the spreadsheet. It involves defining constraints on the data to generate a context free grammar using a GUI and then this grammar is used to validate and reformat the data. There are several *programming by demonstration* [3] (PBD) systems that have been developed for data validation, cleaning

and formatting, which requires the user to specify a complete demonstration or trace visualization on a representative data instead of code. Some of such systems include Simultaneous Editing [18] for string manipulation, SMARTedit [17] for text manipulation and Wrangler [15] for table transformations. In contrast to these systems, our system is based on programming by example (PBE) – it requires the user to provide only the input and output examples without providing the intermediate configurations which renders our system more usable [16], although at the expense of making the learning problem harder. Our expression languages also learns more sophisticated transformations involving conditionals. The by-example interface [7] has also been developed for synthesizing bit-vector algorithms [14], spreadsheet macros [8] (including semantic string manipulation [21] and table layout manipulation [12]), and even some intelligent tutoring scenarios (such as geometry constructions [10] and algebra problems [23]).

Programming by example can be seen as an instantiation of the general program synthesis problem, where the provided input-output examples constitutes the specification. Program synthesis has been used recently to synthesize many classes of non-trivial algorithms, e.g. graph algorithms [13], bit-streaming programs [26, 9], program inverses [27], interactive code snippets [11, 19], and data-structures [24, 25]. There are a range of techniques used in these systems including exhaustive search, constraint-based reasoning, probabilistic inference, type-based search, theorem proving and version-space algebra. A recent survey [5] explains them in more details. Lau et al. used the version-space algebra based technique for learning functions in a PBD setting [17], our system uses it for learning expressions in a PBE setting.

8 Conclusions

We have presented a number transformation language that can model number formatting and rounding transformations, and an inductive synthesis algorithm that can learn transformations in this language from a few input-output examples. We also showed how to combine our system for number transformations with the one for syntactic string transformations [6] to enable manipulation of data types that contain numbers as substrings (such as date and time). In addition to helping end-users who lack programming expertise, we believe that our system is also useful for programmers since it can provide a consistent number formatting interface across all programming languages.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
3. A. Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, 1993.

4. M. Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, April 2009.
5. S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
6. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
7. S. Gulwani. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012.
8. S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *Communications of the ACM*, 2012. To Appear.
9. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
10. S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61, 2011.
11. T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.
12. W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.
13. S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.
14. S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.
15. S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.
16. T. Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI Workshop on Usable AI*, 2008.
17. T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.
18. R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2001.
19. D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.
20. C. Scaffidi, B. A. Myers, and M. Shaw. Topes: reusable abstractions for validating data. In *ICSE*, pages 1–10, 2008.
21. R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012. (To appear).
22. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. Technical Report MSR-TR-2012-42, Apr 2012.
23. R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012. (To appear).
24. R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, pages 289–299, 2011.
25. A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.
26. A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.
27. S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.