DOI:10.1145/2240236.2240260

Spreadsheet Data Manipulation Using Examples

By Sumit Gulwani, William R. Harris, and Rishabh Singh

Abstract

Millions of computer end users need to perform tasks over large spreadsheet data, yet lack the programming knowledge to do such tasks automatically. We present a *programming by example* methodology that allows end users to automate such repetitive tasks. Our methodology involves designing a domain-specific language and developing a synthesis algorithm that can learn programs in that language from user-provided examples. We present instantiations of this methodology for particular domains of tasks: (a) syntactic transformations of strings using restricted forms of regular expressions, conditionals, and loops, (b) semantic transformations of strings involving lookup in relational tables, and (c) layout transformations on spreadsheet tables. We have implemented this technology as an add-in for the Microsoft Excel Spreadsheet system and have evaluated it successfully over several benchmarks picked from various Excel help forums.

1. INTRODUCTION

The IT revolution over the past few decades has resulted in two significant advances: the digitization of massive amounts of data and widespread access to computational devices. It is thus not surprising that more than 500 million people worldwide use spreadsheets for storing and manipulating data. These business *end users* have myriad diverse backgrounds and include commodity traders, graphic designers, chemists, human resource managers, finance professionals, marketing managers, underwriters, compliance officers, and even mailroom clerks—they are not professional programmers, but they need to create small, *often one-off*, applications to perform business tasks.[4]

Unfortunately, the state of the art of interfacing with spreadsheets is far from satisfactory. Spreadsheet systems, like Microsoft Excel, come with a maze of features, but end users struggle to find the correct features to accomplish their tasks.[12] More significantly, programming is still required to perform tedious and repetitive tasks such as transforming names or phone numbers or dates from one format to another, cleaning data, or extracting data from several text files or Web pages into a single document. Excel allows users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts in Visual Basic or .NET programming languages. However, since end users are not proficient in programming, they find it too difficult to write desired macros or scripts. Moreover, even skilled programmers might hesitate to write a script for a *one-off* repetitive task.

We performed an extensive case study of spreadsheet help forums and observed that string and table processing is a very common class of programming problems that end users struggle with. This is not surprising given that various languages such as Perl, Awk, and Python were designed to support string processing, and that new languages such as Java and C# provide rich support for string processing. During our study, we also observed how novice users specified their desired programs to expert users: most specifications consisted solely of one or more input–output examples. Since input–output examples may underspecify a program, the interaction between a novice and an expert often involved multiple rounds of communication over multiple days. Inspired by this observation, we developed a *programming by example* (PBE), or *inductive synthesis*, methodology[15] that has produced synthesizers that can automatically generate a wide range of string/table manipulating programs in spreadsheets from input–output examples. Each synthesizer takes the role of the forum expert, removing a human from the interaction loop and enabling users to solve their problems in a few seconds instead of few days.

This paper is organized as follows. We start with a brief overview of our PBE methodology (Section 2). We then describe an application of this methodology to perform syntactic string manipulation tasks (Section 3).[6] This is followed by an extension that automates more sophisticated semantic string manipulations requiring background knowledge, which can often be encoded as relational tables (Section 4).[18] We also describe an application of this methodology to perform layout transformations on tables (Section 5).[8] In Section 6, we discuss related work, and in Section 7, we conclude and discuss future work.

2. OVERVIEW

In this section, we outline a general methodology that we have used for developing inductive synthesizers for end-user programming tasks, along with how a user can interact with the synthesizers. In the first step of our methodology, we identify a *domain* of useful tasks that end users struggle with

This paper is based on “Automating String Processing in Spreadsheets using Input-Output,” S. Gulwani, published in *POPL* (2011); “Learning Semantic String Transformations from Examples,” R. Singh and S. Gulwani, *PVLDB 5* (2012), in press; and “Spreadsheet Table Transformations from Examples,” W.R. Harris and S. Gulwani, published in *PLDI* (2011).

Harris’s work was done during an internship at Microsoft Research.
Singh’s work was done during two internships at Microsoft Research.

and can clearly describe with examples, by studying help forums or performing user studies (this paper presents two domains: string manipulation and table manipulation). We then develop the following.

Domain-specific language: We design a domain-specific language *L* that is expressive enough to capture several real-world tasks in the domain, but also restricted enough to enable efficient learning from examples.

Data structure for representing consistent programs: The number of programs in *L* that are consistent with a given set of input–output examples can be huge. We define a data structure *D* based on a version-space algebra[14] to succinctly represent a large set of such programs.

Algorithm for synthesizing consistent programs: Our synthesis algorithm for language *L* applies two key procedures: (i) `Generate` learns the set of all programs, represented using data structure *D*, that are consistent with a given single example. (ii) `Intersect` intersects these sets (each corresponding to a different example).

Ranking: We develop a scheme that ranks programs, preferring programs that are more general. Each ranking scheme is inspired by Occam's razor, which states that a smaller and simpler explanation is usually the correct one. We define a partial order relationship between programs to compare them. Any partial order can be used that efficiently orders programs represented in the version-space algebra used by the data structure *D*. Such an order can be applied to efficiently select the top-ranked programs from among a set represented using *D*. The ranking scheme can also take into account any *test inputs* provided by the user (i.e., new additional inputs on which the user may execute a synthesized program). A program that is undefined on any test input or generates an output whose characteristics are different from that of training outputs can be ranked lower.

2.1. Interaction models

A user provides to the synthesizer a small number of examples, and then can interact with the synthesizer according to multiple models. In one model, the user runs the top-ranked synthesized program on other inputs in the spreadsheet and checks the outputs produced by the program. If any output is incorrect, the user can fix it and reapply the synthesizer, using the fix as an additional example. However, requiring the user to check the results of the synthesized program, especially on a large spreadsheet, can be cumbersome. To enable easier interaction, the synthesizer can run *all* synthesized programs on each new input to generate a set of corresponding outputs for that input. The synthesizer can highlight for the user the inputs that cause multiple distinct outputs. Our prototypes, implemented as Excel add-ins, support this interaction model.

A second model accommodates a user who requires a reusable program. In this model, the synthesizer presents the set of consistent programs to the user. The synthesizer can show the top *k* programs or walk the user through the data structure that succinctly represents all consistent programs and let the user select a program. The programs can be shown using programming-language syntax, or they can be described in a natural language. The differences between different programs can be explained by synthesizing a *distinguishing input* on which the programs behave differently.[10] The user can reapply the synthesizer with the distinguishing input and its desired output as an additional example.

3. SYNTACTIC TRANSFORMATIONS

Spreadsheet users often struggle with reformatting or cleaning data in spreadsheet columns. For example, consider the following task.

Example 1 (Phone Numbers). *An Excel user wants to uniformly format the phone numbers in the input column, adding a default area code of "425" if the area code is missing.*

Input v_1	Output
323-708-7700	323-708-7700
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
235 7654	425-235-7654
745-8139	425-745-8139

Such tasks can be automated by applying a program that performs syntactic string transformations. We now present an expressive domain-specific language of string-processing programs that supports limited conditionals and loops, syntactic string operations such as substring and concatenate, and matching based on regular expressions.[6]

3.1. Domain-specific language

Our domain-specific programming language for performing syntactic string transformations is given in Figure 1(a). A string program *P* is an expression that maps an input state σ, which holds values for *m* string variables $v_1, \dots, v_m$ (denoting the multiple input columns in a spreadsheet), to an output string *s*. The top-level string expression *P* is a `Switch` constructor whose arguments are pairs of Boolean expressions b and trace expressions e. The set of Boolean expressions in a `Switch` construct must be disjoint, that is, for any input state, at most one of the Boolean expressions can be true. The value of *P* in a given input state σ is the value of the trace expression that corresponds to the Boolean expression satisfied by σ. A Boolean expression b is a propositional formula in Disjunctive Normal Form (DNF). A predicate `Match`(v_i, r, *k*) is satisfied if and only if v_i contains at least *k* nonoverlapping matches of regular expression r. (In general, any finite set of predicates can be used.)

A *trace expression* `Concatenate`($f_1, \dots, f_n$) is the concatenation of strings represented by atomic expressions $f_1, \dots, f_n$. An *atomic expression* f is either a constant-string expression `ConstStr`, a substring expression constructed from `SubStr`, or a loop expression constructed from `Loop`.

The substring expression `SubStr`(v_i, p_1, 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.) `SubStr`(v_i, p_1, p_2) is the substring of string v_i in between positions p_1 and p_2. For a nonnegative

Figure 1. (a) Syntax of syntactic string-processing programs. (b) Data structure for representing a set of such programs.

$$\begin{array}{rcl}
\text{String program } P & := & \texttt{Switch}\ ((\mathrm{b}_1, \mathrm{e}_1), \ldots, (\mathrm{b}_n, \mathrm{e}_n)) \ \mid\ \mathrm{e} \\
\text{Boolean condition b} & := & \mathrm{d}_1 \vee \ldots \vee \mathrm{d}_n \\
\text{Conjunction d} & := & \pi_1 \wedge \ldots \wedge \pi_n \\
\text{Predicate } \pi & := & \texttt{Match}(\upsilon_i, \mathrm{r}, k) \mid \neg\, \texttt{Match}(\upsilon_i, \mathrm{r}, k) \\
\text{Trace expr e} & := & \texttt{Concatenate}(\mathrm{f}_1, \ldots, \mathrm{f}_n) \ \mid\ \mathrm{f} \\
\text{Atomic expr f} & := & \texttt{ConstStr}(s) \mid \texttt{SubStr}(\upsilon_i, \mathrm{p}_1, \mathrm{p}_2) \mid \mathrm{Loop}(\lambda w : \mathrm{e}) \\
\text{Position p} & := & \texttt{CPos}(k) \mid \mathrm{Pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c}) \\
\text{Integer expr c} & := & k \mid k_1 w + k_2 \\
\text{Regular expr r} & := & \texttt{TokenSeq}(\mathrm{T}_1, \ldots, \mathrm{T}_n) \ \mid\ \mathrm{T} \ \mid\ \varepsilon
\end{array}$$

(a)

$$\begin{array}{rcl}
\tilde{P} & := & \texttt{Switch}((\mathrm{b}_1, \tilde{\mathrm{e}}_1), \ldots, (\mathrm{b}_n, \tilde{\mathrm{e}}_n)) \\
\tilde{\mathrm{e}} & := & \texttt{Dag}(\tilde{\eta}, \eta^s, \eta^t, \tilde{\xi}, W), \\
 & & \quad \text{where } W : \tilde{\xi} \to 2^{\tilde{\mathrm{f}}} \\
\tilde{\mathrm{f}} & := & \texttt{ConstStr}(s) \\
 & \mid & \texttt{SubStr}(\upsilon_i, \{\tilde{\mathrm{p}}_j\}_j, \{\tilde{\mathrm{p}}_k\}_k) \\
 & \mid & \texttt{Loop}\ (\lambda w : \tilde{\mathrm{e}}) \\
\tilde{\mathrm{p}} & := & \texttt{CPos}(k) \\
 & \mid & \texttt{Pos}(\tilde{\mathrm{r}}_1, \tilde{\mathrm{r}}_2, \tilde{\mathrm{c}})
\end{array}$$

(b)

constant k, CPos(k) denotes the k^{th} position in the subject string. For a negative constant k, CPos(k) denotes the $(\ell + 1 + k)^{th}$ position in the subject string, where ℓ = Length(s). Pos(r_1,r_2,c) is another position expression, where r_1 and r_2 are regular expressions and integer expression c evaluates to a nonzero integer. Pos(r_1,r_2,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 ℓ = 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. The substring construct is quite expressive. For example, in the expression SubStr(ν_i, Pos(r_1,r_2,c), Pos(r_3,r_4,c)), r_2 and r_3 describe the characteristics of the substring in ν_i to be extracted, while r_1 and r_4 describe the characteristics of the surrounding delimiters. We use the expression SubStr2(ν_i, r, c) as an abbreviation to denote the c^{th} occurrence of regular expression r in ν_i, that is, SubStr(ν_i, Pos(ε, r, c), Pos(r, ε, c)).

A regular expression r is ε (which matches the empty string, and therefore can match at any position of any string), a token T, or a token sequence TokenSeq(T_1, ..., T_n). This restricted choice of regular expressions enables efficient enumeration of regular expressions that match certain parts of a string. We use the following finite (but easily extended) set of tokens: (a) StartTok, which matches the beginning of a string, (b) EndTok, which matches the end of a string, (c) a token for each special character, such as hyphen, dot, semicolon, comma, slash, or left/right parenthesis/bracket, and (d) two tokens for each character class C, one that matches a sequence of one or more characters in C, and another that matches a sequence of one or more characters that are not in C. Examples of a character class C include numeric digits (0–9), alphabetic characters (a–zA–Z), lowercase alphabetic characters (a–z), uppercase alphabetic characters (A–Z), alphanumeric characters, and whitespace characters. UpperTok, NumTok, and AlphNumTok match a nonempty sequence of uppercase alphabetic characters, numeric digits, and alphanumeric characters, respectively. DecNumTok matches a nonempty sequence of numeric digits and/or decimal point. HyphenTok and SlashTok match the hyphen character and the slash character, respectively.

The task described in Example 1 can be expressed in our domain-specific language as

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{Switch}\ ((\mathrm{b}_1, \mathrm{e}_1), (\mathrm{b}_2, \mathrm{e}_2)),\ \texttt{where}} \\
\mathrm{b}_1 & \equiv & \texttt{Match}(\nu_1, \texttt{NumTok, 3}) \\
\mathrm{b}_2 & \equiv & \neg\texttt{Match}(\nu_1, \texttt{NumTok, 3}) \\
\mathrm{e}_1 & \equiv & \texttt{Concatenate}(\texttt{SubStr2}(\nu_1, \texttt{NumTok, 1}), \texttt{ConstStr}(\text{“–”}), \\
 & & \qquad \texttt{SubStr2}(\nu_1, \texttt{NumTok, 2}), \texttt{ConstStr}(\text{“–”}), \\
 & & \qquad \texttt{SubStr2}(\nu_1, \texttt{NumTok, 3})) \\
\mathrm{e}_2 & \equiv & \texttt{Concatenate}(\texttt{ConstStr}(\text{“425–”}), \texttt{SubStr2}(\nu_1, \texttt{NumTok, 1}), \\
 & & \qquad \texttt{ConstStr}(\text{“–”}), \texttt{SubStr2}(\nu_1, \texttt{NumTok, 2}))
\end{array}$$

The atomic expression Loop(λw : e) is the concatenation of e_1, e_2, ..., e_n, where e_i is obtained from e by replacing all occurrences of integer w by i, and n is the smallest integer such that evaluation of e_{n+1} is undefined. (It is also possible to define more interesting termination conditions, e.g., based on position expressions or predicates.) A trace expression e is undefined when (i) a constituent CPos(k) expression refers to a position not within its subject string, (ii) a constituent Pos(r_1, r_2, c) expression is such that the subject string does not contain c occurrences of a match bounded by r_1 and r_2, or (iii) a constituent SubStr(ν_i, p_1, p_2) expression has position expressions that are both defined but the first refers to a position that occurs later in the subject string than the position indicated by the second. The following example illustrates the utility of the loop construct.

Example 2 (Generate Abbreviation). *The following task was presented originally as an* Advanced Text Formula.[23]

Input v_1	Output
Association of Computing Machinery	ACM
Principles Of Programming Languages	**POPL**
Foundations of Software Engineering	**FSE**

This task can be expressed in our language as

$$\texttt{Loop}(\lambda w : \texttt{Concatenate}(\texttt{SubStr2}(\nu_1, \texttt{UpperTok}, w))).$$

Our tool synthesizes this program from the first example row and uses it to produce the entries in the second and third rows (shown here in boldface for emphasis) of the output column.

3.2. Synthesis algorithm

The synthesis algorithm first computes, for each input–output example (σ, s), the set of *all* trace expressions that map input σ to output s (using procedure `Generate`). It then intersects these sets for similar examples and learns conditionals to handle different cases (using procedure `Intersect`). The size of such sets can be huge; therefore, we must develop a data structure that allows us to succinctly represent and efficiently manipulate huge sets of program expressions.

Data structure: Figure 1(b) describes our data structure for succinctly representing sets of programs from our domain-specific language. $\widetilde{P}$, $\widetilde{\mathtt{e}}$, $\widetilde{\mathtt{f}}$, and $\widetilde{\mathtt{p}}$ denote representations of, respectively, a set of string programs, a set of trace expressions, a set of atomic expressions, and a set of position expressions. $\widetilde{\mathtt{r}}$ and $\widetilde{\mathtt{c}}$ represent a set of regular expressions and a set of integer expressions; these sets are represented explicitly.

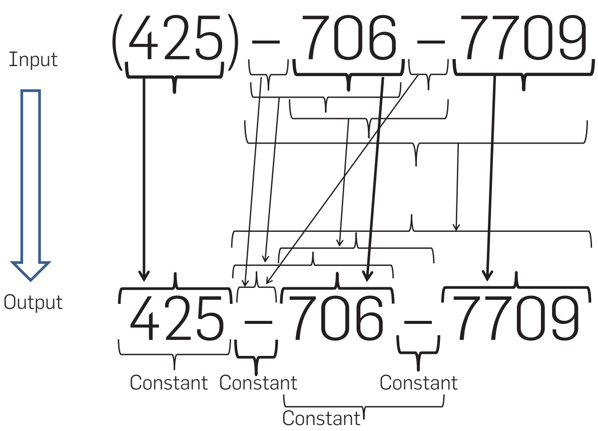
The `Concatenate` constructor used in our string language is generalized to the `Dag` constructor `Dag`$(\widetilde{\eta}, \eta^s, \eta^t, \widetilde{\xi}, W)$, where $\widetilde{\eta}$ is a set of nodes containing two distinctly marked source and target nodes η^s and η^t, $\widetilde{\xi}$ is a set of edges over nodes in $\widetilde{\eta}$ that defines a Directed Acyclic Graph (DAG), and W maps each $\xi \in \widetilde{\xi}$ to a set of atomic expressions. The set of all `Concatenate` expressions represented by a `Dag`$(\widetilde{\eta}, \eta^s, \eta^t, \widetilde{\xi}, W)$ constructor includes exactly those whose ordered arguments belong to the corresponding edges on any path from η^s to η^t. The `Switch`, `Loop`, `SubStr`, and `Pos` constructors are all overloaded to construct sets of the corresponding program expressions that are shown in Figure 1(a). The `ConstStr` and `CPos` constructors can be regarded as producing singleton sets.

The data structure supports efficient implementation of various useful operations including intersection, enumeration of programs, and their simultaneous execution on a given input. The most interesting of these is the intersection operation, which is similar to regular automata intersection. The additional challenge is to intersect edge labels—in the case of automata, the labels are simply sets of characters, while in our case, the labels are sets of string expressions.

Procedure `Generate`**:** The number of trace expressions that can generate a given output string from a given input state can be huge. For example, consider the second input–output pair in Example 1, where the input state consists of one string "(425)-706-7709" and the output string is "425-706-7709". Figure 2 shows a small sampling of different ways of generating parts of the output string from the input string using `SubStr` and `ConstStr` constructors. Each substring extraction task itself can be expressed with a huge number of expressions, as explained later. The following are three of the trace expressions represented in the figure, of which only the second one, shown in the figure in bold, expresses the program expected by the user:

1. Extract substring "425". Extract substring "-706-7709".
2. Extract substring "425". Print constant "-". Extract substring "706". Print constant "-". Extract substring "7709".
3. Extract substring "425". Extract substring "-706". Print constant "-". Extract substring "7709".

Figure 2. Small sampling of different ways of generating parts of an output string from the input string.



We apply two crucial observations to succinctly generate and represent all such trace expressions. First, the logic for generating some substring of an output string is completely decoupled from the logic for generating another disjoint substring of the output string. Second, the total number of different substrings/parts of a string is quadratic (and not exponential) in the size of that string.

The `Generate` procedure creates a Directed Acyclic Graph (DAG) `Dag`$(\widetilde{\eta}, \eta^s, \eta^t, \widetilde{\xi}, W)$ that represents the *trace set* of all trace expressions that generate a given output string from a given input state. `Generate` constructs a node corresponding to each position within the output string and constructs an edge from a node corresponding to any position to a node corresponding to any later position. Each edge corresponds to some substring of the output and is annotated with the set of all atomic expressions that generate that substring. We describe below how to generate the set of all such `SubStr` expressions. Any `Loop` expressions are generated by first generating candidate expressions (by *unifying* the sets of trace expressions associated with the substrings $s[k_1 : k_2]$ and $s[k_2 : k_3]$, where k_1, k_2, and k_3 are the boundaries of the first two loop iterations, identified by considering all possibilities), and then validating them.

The number of substring expressions that can extract a given substring from a given string can be huge. For example, following is a small sample of various expressions that extract "706" from the string "425-706-7709" (call it v_1).

- Second number: `SubStr2`(v_1, `NumTok`, 2).
- 2nd last alphanumeric token:
  `SubStr2`(v_1, `AlphNumTok`, –2).
- Substring between the first hyphen and the last hyphen:
  `SubStr`(v_1, `Pos`(`HyphenTok`, ε, 1), `Pos`(ε, `HyphenTok`, –1)).
- First number that occurs between hyphen on both ends.
  `SubStr`(v_1, `Pos`(`HyphenTok`, `TokenSeq`(`NumTok`, `HyphenTok`), 1), `Pos`(`TokenSeq`(`HyphenTok`, `NumTok`), `HyphenTok`, 1)).
- First number preceded by a number–hyphen sequence.
  `SubStr`(v_1, `Pos`(`TokenSeq`(`NumTok`, `HyphenTok`), `NumTok`, 1), `Pos`(`TokenSeq`(`NumTok`, `HyphenTok`, `NumTok`), ε, 1)).

The substring-extraction problem can be decomposed into two *independent* position-identification problems, each of which can be solved independently. The solutions to the substring-extraction problem can also be maintained succinctly by independently representing the solutions to the two position-identification problems. Note the representation of the `SubStr` constructor in Eq. 1 in Figure 1(b).

Procedure `Intersect`: Given a trace set for each input–output example, the `Intersect` procedure generates the top-level `Switch` constructor. `Intersect` first partitions the examples, so that inputs in the same partition are handled by the same conditional in the top-level `Switch` expression, and then intersects the trace sets for inputs in the same partition. If a set of inputs are in the same partition, then the intersection of trace sets is nonempty. `Intersect` uses a greedy heuristic to minimize the number of partitions by starting with singleton partitions and then iteratively merging partitions that have the highest *compatibility score*, which is a function of the size of the resulting partition and its potential to be merged with other partitions.

`Intersect` then constructs a classifier for each of the resultant partitions, which is a Boolean expression that is satisfied by exactly the inputs in the partition. The classifier for each partition and the intersection of trace sets for the inputs in the partition serve as the Boolean condition and corresponding trace expression in the constructed `Switch` expression, respectively.

Ranking: We prefer `Concatenate` and `TokenSeq` expressions that have fewer arguments. We prefer `SubStr` expressions to both `ConstStr` expressions (it is less likely for constant parts of an output string to also occur in the input) and `Concatenate` expressions (if there is a long substring match between the input and output, it is more likely that the corresponding part of the output was produced by a single substring extraction). We prefer a `Pos` expression to `CPos` expression (giving less preference to extraction expressions based on constant offsets). `StartTok` and `EndTok` are our most preferred tokens; otherwise, we prefer tokens corresponding to a larger character class (favoring generality).

The implementation of the synthesis algorithm is less than 5,000 lines of C# code, and takes less than 0.1 s on average for a benchmark suite of more than 100 tasks obtained from online help forums and the Excel product team.

4. SEMANTIC TRANSFORMATIONS

Some string transformation tasks also involve manipulating strings that need to be interpreted as more than a sequence of characters, for example, as a column entry from some relational table, or as some standard data type such as date, time, currency, or phone number. For example, consider the following task from an Excel help forum.

EXAMPLE 3. *A shopkeeper wants to compute the* `selling price` *of an item (Output) from its* `name` *(Input v_1) and* `selling date` *(Input v_2). The inventory database of the shop consists of two tables: (i)* `MarkupRec` *table that stores* `id`, `name` *and* `markup percentage` *of items, and (ii)* `CostRec` *table that stores* `id`, `purchase date` *(in month/year format), and* `purchase price` *of items. The selling price of an item is computed by adding its purchase price (for the corresponding month) to its markup charges, which in turn is calculated by multiplying the markup percentage by the purchase price.*

Input v_1	**Input v_2**	**Output**
Stroller	10/12/2010	$145.67 + 0.30*145.67
Bib	23/12/2010	$3.56 + 0.45*3.56
Diapers	21/1/2011	**$21.45 + 0.35*21.45**
Wipes	2/4/2009	**$5.12 + 0.40*5.12**
Aspirator	23/2/2010	**$2.56 + 0.30*2.56**

MarkupRec

Id	Name	Markup
S33	Stroller	30%
B56	Bib	45%
D32	Diapers	35%
W98	Wipes	40%
A46	Aspirator	30%
…	…	…

CostRec

Id	Date	Price
S33	12/2010	$145.67
S33	11/2010	$142.38
B56	12/2010	$3.56
D32	1/2011	$21.45
W98	4/2009	$5.12
A46	2/2010	$2.56
…	…	…

To perform the above task, the user must perform a join of the two tables on the common item `Id` column to lookup the item `Price` from its `Name` (v_1) and selling `Date` (substring of v_2). We present an extension to the trace expression (from Section 3.1) that can also manipulate strings present in such relational tables.[18]

4.1. Domain-specific language

We extend the trace expression (from Section 3.1), as shown in Figure 3(a), to obtain the semantic string transformation language that can also perform table lookup operations. The atomic expression f is modified to represent a constant string, a select expression, or a substring of a select expression. A *select expression* e_t is either an input string variable v_i or a lookup expression denoted by `Select(Col, Tab, g)`, where `Tab` is a relational table identifier and `Col` is a column identifier of the table. The Boolean condition g is an ordered conjunction of column predicates $h_1 \wedge \ldots \wedge h_n$, where a column predicate h is an equality comparison between the content of some column of the table and a constant or a trace expression e. We require columns present in these conditions to together constitute a *primary key* of the table to ensure that the select queries produce a single string as opposed to a set of strings.

The task in Example 3 can be represented in the language as

$$\begin{array}{l}
\texttt{Concatenate}\ (f_1, \texttt{ConstStr}(\text{“+0.”}), f_2, \texttt{ConstStr}(\text{“*”}), f_3) \\
\text{where } f_1 \equiv \texttt{Select}(\texttt{Price}, \texttt{CostRec}, \texttt{Id} = f_4 \wedge \texttt{Date} = f_5), \\
f_4 \equiv \texttt{Select}(\texttt{Id}, \texttt{MarkupRec}, \texttt{Name} = v_1), \\
f_5 \equiv \texttt{SubStr}(v_2, \texttt{Pos}(\texttt{SlashTok}, \varepsilon, 1), \texttt{Pos}(\varepsilon, \texttt{EndTok}, 1)), \\
f_2 \equiv \texttt{SubStr2}(f_6, \texttt{NumTok}, 1), f_3 \equiv \texttt{SubStr2}(f_1, \texttt{DecNumTok}, 1), \\
f_6 \equiv \texttt{Select}(\texttt{Markup}, \texttt{MarkupRec}, \texttt{Name} = v_1).
\end{array}$$

Figure 3. Extensions to the syntax and data structure in Figure 1 for semantic processing.

$$
\begin{array}{rcl}
\text{Atomic expr f} & \coloneqq & \texttt{SubStr}(\mathsf{e}_t, \mathsf{p}_1, \mathsf{p}_2) \mid \texttt{ConstStr}(s) \mid \mathsf{e}_t \\
\text{Select expr } \mathsf{e}_t & \coloneqq & v_i \quad \mid \texttt{Select(Col, Tab, g)} \\
\text{Boolean condition g} & \coloneqq & \mathsf{h}_1 \wedge \ldots \wedge \mathsf{h}_n \\
\text{Predicate h} & \coloneqq & \texttt{Col} = s \quad \mid \texttt{Col} = \mathsf{e}
\end{array}
$$

(a)

$$
\begin{array}{rcl}
\tilde{\mathsf{e}}_t & \coloneqq & (\tilde{\eta}, \eta^t, \texttt{Progs}) \text{ where Progs} : \tilde{\eta} \rightarrow 2^{\tilde{\mathsf{f}}} \\
\tilde{\mathsf{f}} & \coloneqq & v_i \mid \quad \texttt{Select(Col, Tab, B)} \\
\mathsf{B} & \coloneqq & \{\tilde{\mathsf{g}}_i\}_i \\
\tilde{\mathsf{g}} & \coloneqq & \tilde{\mathsf{h}}_1 \wedge \ldots \wedge \tilde{\mathsf{h}}_n \\
\tilde{\mathsf{h}} & \coloneqq & \texttt{Col} = s \mid \texttt{Col} = \eta \mid \texttt{Col} = \{s, \eta\}
\end{array}
$$

(b)

The expression f_4 looks up the Id of the item (present in v_1) from the MarkupRec table and f_5 generates a substring of the date present in v_2, which are then used together to lookup the Price of the item from the CostRec table (f_1). The expression f_6 looks up the Markup percentage of the item from the MarkupRec table and f_2 generates a substring of this lookup value by extracting the first numeric token (thus removing the % sign). Similarly, the expression f_3 generates a substring of f_1, removing the \$ symbol. Finally, the top-level expression concatenates the strings obtained from expressions f_1, f_2, and f_3 with constant strings "+0." and "*".

This extended language also enables manipulation of strings that represent standard data types, whose semantic meaning can be encoded as a database of relational tables. For example, consider the following date manipulation task.

EXAMPLE 4 (DATE MANIPULATION). *An Excel user wanted to convert dates from one format to another, and the fixed set of hard-coded date formats supported by Excel 2010 do not match the input and output formats. Thus, the user solicited help on a forum.*

Input v_1	Output
6-3-2008	Jun 3rd, 2008
3-26-2010	**Mar 26th, 2010**
8-1-2009	**Aug 1st, 2009**
9-24-2007	**Sep 24th, 2007**

We can encode the required background knowledge for the date data type in two tables, namely a Month table with 12 entries: (1, January), ..., (12, December) and a DateOrd table with 31 entries (1, st), (2, nd), ..., (31, st). The desired transformation is represented in our language as

$$
\begin{array}{l}
\texttt{Concatenate(SubStr(Select(MW, Month, MN} = \mathsf{e}_1\texttt{),} \\
\qquad \texttt{Pos(StartTok}, \varepsilon, 1\texttt{), CPos(3)), ConstStr(" "), } \mathsf{e}_2\texttt{,} \\
\qquad \texttt{Select(Ord, DateOrd, Num} = \mathsf{e}_2\texttt{), ConstStr(", "), } \mathsf{e}_3\texttt{)}
\end{array}
$$

where e_1 = SubStr2(v_1, NumTok, 1), e_2 = SubStr2(v_1, NumTok, 2), e_3 = SubStr2(v_1, NumTok, 3). (MW,MN) and (Num,Ord) denote the columns of the Month and DateOrd tables, respectively.

4.2. Synthesis algorithm

We now describe the key extensions to the synthesis algorithm for syntactic transformations (Section 3.2) to obtain the synthesis algorithm for semantic transformations.

Data structure: Figure 3(b) describes the data structure that succinctly represents the large set of programs in the semantic transformation language that are consistent with a given input–output example. The data structure consists of a generalized expression $\tilde{\mathsf{e}}_t$, generalized Boolean condition $\tilde{\mathsf{g}}$, and generalized predicate $\tilde{\mathsf{h}}$ (which, respectively, denote a set of select expressions, a set of Boolean conditions g, and a set of predicates h). The generalized expression $\tilde{\mathsf{e}}_t$ is represented using a tuple ($\tilde{\eta}$, η^t, Progs) where $\tilde{\eta}$ denotes a set of nodes containing a distinct target node η^t (representing the output string), and Progs : $\tilde{\eta} \rightarrow 2^{\tilde{\mathsf{f}}}$ maps each node $\eta \in \tilde{\eta}$ to a set consisting of input variables v_i or generalized select expressions Select(Col, Tab, B). A generalized Boolean condition $\tilde{\mathsf{g}}_i$ corresponds to some primary key of table T and is a conjunction of generalized predicates $\tilde{\mathsf{h}}_j$, where each $\tilde{\mathsf{h}}_j$ is an equality comparison of the j^{th} column of the corresponding primary key with a constant string s or some node $\tilde{\eta}$ or both. The two key aspects of this data structure are (i) the use of intermediate nodes for sharing sub-expressions to represent an exponential number of expressions in polynomial space and (ii) the use of Conjunctive Normal Form (CNF) form of Boolean conditions to represent an exponential number of conditionals $\tilde{\mathsf{g}}$ in polynomial space.

Procedure Generate: We first consider the simpler case where there are no syntactic manipulations on the table lookups and the lookups are performed using exact string matches, that is, the predicate h is Col = e_t instead of Col = e. The Generate procedure operates by iteratively computing a set of nodes ($\tilde{\eta}$), where each node $\eta \in \tilde{\eta}$ represents a string val(η) that corresponds to some table entry or an input string. Generate performs an iterative forward reachability analysis of the string values that can be generated in a single step (i.e., using a single Select expression) from the string values computed in the previous step based on *string equality* and assigns the Select expression to the Progs map of the corresponding node. The base case of the procedure creates a node for each of the input string variables. After performing the analysis for a bounded number of iterations, the procedure returns the set of select expressions of the node that corresponds to the output string s, that is, Progs[val^{-1}(s)].

The Generate procedure for the general case, which also includes syntactic manipulations on table lookups, requires a relaxation of the above-mentioned reachability criterion of strings that is based on string equality. We now define a table entry to be reachable from a set of previously reachable strings if the entry can be generated from the set of reachable strings using the `Generate` procedure of Section 3.2. The rest of the reachability algorithm operates just as before.

Procedure `Intersect`: A basic ingredient of the `Intersect` procedure for syntactic transformations is a method to intersect two `Dag` constructs, each representing a set of trace expressions. We replace this by a method to intersect two tuples ($\tilde{\eta}_1$, η^t_1, $\texttt{Progs}_1$) and ($\tilde{\eta}_2$, η^t_2, $\texttt{Progs}_2$), each representing a set of extended trace expressions. The tuple after intersection is ($\tilde{\eta}_1 \times \tilde{\eta}_2$, (η^t_1,η^t_2), $\texttt{Progs}_{12}$), where $\texttt{Progs}_{12}((\tilde{\eta}_1, \tilde{\eta}_2))$ is given by the intersection of $\texttt{Progs}_1(\tilde{\eta}_1)$ and $\texttt{Progs}_2(\tilde{\eta}_2)$.

Ranking: We prefer expressions of smaller depth (fewer nested chains of `Select` expressions) and ones that match longer strings in table entries for indexing. We prefer lookup expressions that use distinct tables (for join queries) as opposed to using the same table twice. We prefer conditionals with fewer predicates. We prefer predicates that compare columns with other table entries or input variables (as opposed to comparing columns with constant strings).

We implemented our algorithm as an extension to the Excel add-in (Section 3.2) and evaluated it successfully over more than 50 benchmark problems obtained from various help forums and the Excel product team. For each benchmark, our implementation learned the desired transformation in <10 s (88% of them taking <1 s each) requiring at most three input–output examples (with 70% of them requiring only one example). The data structure had size between 100 and 2,000 (measured as the number of terminal symbols in the data-structure syntax), with an average size of 600, and typically represented 10^{20} expressions.

5. TABLE LAYOUT TRANSFORMATIONS

End users often transform a spreadsheet table not by changing the data stored in the cells of a table, but instead by changing how the cells are grouped or arranged. In other words, users often transform the *layout* of a table.[8]

EXAMPLE 5. *The following example input table and subsequent example output table were provided by a novice on an Excel user help thread to specify a layout transformation:*

	Qual 1	Qual 2	Qual 3
Andrew	01.02.2003	27.06.2008	06.04.2007
Ben	31.08.2001		05.07.2004
Carl		18.04.2003	09.12.2009

Andrew	Qual 1	01.02.2003
Andrew	Qual 2	27.06.2008
Andrew	Qual 3	06.04.2007
Ben	Qual 1	31.08.2001
Ben	Qual 3	05.07.2004
Carl	Qual 2	18.04.2003
Carl	Qual 3	09.12.2009

The example input contains a set of dates on which tests were given, where each date is in a row corresponding to the name of the test taker, and in a column corresponding to the name of the test. For every date, the user needs to produce a row in the output table containing the name of the test taker, the name of the test, and the date on which the test was taken. If a date cell in the input is empty, then no corresponding row should be produced in the output.

5.1. Domain-specific language

We may view every program P that transforms the layout of a table as constructing a map m_P from the cells of an input table to the coordinates of the output table. For a cell c in an input table, if $m_P(c) = (\mathrm{row}, \mathrm{col})$, P fills the cell in the output table at the coordinate (row, col) with the data in c. A program in our language of layout transformations is defined syntactically as a finite collection of *component programs*, each of which builds a map from input cells to output coordinates (Figure 4: table program). We designed our language on the principle that most layout transformations can be implemented by a set of component programs that construct their map using one of the two complementary procedures: *filtering* and *associating*.

When a component program filters, it scans the cells of the input table in row-major order, selects a subset of the cells, and maps them in order to a subrange of the coordinates in the output table. A *filter program* $\texttt{Filter}(\varphi, \texttt{SEQ}_{i,j,k})$ (Figure 4: filter program) is a *mapping condition* φ, which is a function whose body is a conjunction of predicates over input cells drawn from a fixed set and an *output sequencer* $\texttt{SEQ}_{i,j,k}$, where i, j, and k are nonnegative integers. For a mapping condition φ and sequencer $\texttt{SEQ}_{i,j,k}$, the filter program $\texttt{Filter}(\varphi, \texttt{SEQ}_{i,j,k})$ scans an input table and maps each cell that satisfies φ to the coordinates in the output table between columns i and j, starting at row k, in row-major order.

For the tables in Example 5, the filter program $F_1 = \texttt{Filter}(\lambda c.(c.data \neq \text{“”} \wedge c.col \neq 1 \wedge c.row \neq 1), \texttt{SEQ}_{3,3,1})$ maps each date, that is, each nonempty cell not in column 1 and not in row 1, to its corresponding cell in column 3 of the output table, starting at row 1. Call this map m_{F_1}.

A table program can also construct a map using spatial relationships between cells in the input table and spatial relationships between coordinates in the output table; we call this construction *association*. When a table program associates, it takes a cell c in the input table mapped by some filter program F, picks a cell c_1 in the input table whose coordinate is related to c, finds the coordinate $m_F(c)$ that c maps to under m_F, picks a coordinate d_1 whose coordinate is related to $m_F(c)$, and maps c_1 to d_1.

An associative program $A = \texttt{Assoc}(F, s_0, s_1)$ (Figure 4: Associative program) is constructed from a filter program F and two *spatial functions* s_0 and s_1, each of which may be of

Figure 4. Syntax of layout transformations.

Table program $P := \texttt{TabProg}(\{\mathrm{K}_i\}_i)$
Component program $\mathrm{K} := \mathrm{F} \mid \mathrm{A}$
Filter program $\mathrm{F} := \texttt{Filter}(\varphi, \mathrm{SEQ}_{i,\,j,\,k})$
Associative program $\mathrm{A} := \texttt{Assoc}(\mathrm{F}, \mathrm{S}_1, \mathrm{S}_2)$
Spatial function $\mathrm{S} := \texttt{RelCol}_i \mid \texttt{RelRow}_j$

the form $\texttt{RelCol}_i$ or $\texttt{RelRow}_j$. The spatial function $\texttt{RelCol}_i$ takes a cell c as input, and returns the cell in the same row as c and in column i. The spatial function $\texttt{RelRow}_j$ takes a cell c as input, and returns the cell in row j and in the same column as c. For each cell c in the domain of m_K, the map of A contains an entry $m_A(s_0(c)) = s_1(m_F(c))$.

For the example tables in Example 5, and the filter program F_1 introduced above that maps to column 3 of the example output table, the associative program $A_1 = \texttt{Assoc}(F_1, \texttt{RelCol}_1, \texttt{RelCol}_1)$ constructs a map to every cell in column 1 of the output table. To construct its map, A_1 takes each cell c in the input table mapped by F_1, finds the cell $\texttt{RelCol}_1(c)$ in the same row as c and in column 1, finds the coordinate $m_{F_1}(c)$ that F_1 maps c to, finds the coordinate $\texttt{RelCol}_1(m_{F_1}(c))$, and maps $\texttt{RelCol}_1(c)$ to $\texttt{RelCol}_1(m_{F_1}(c))$: that is, A_1 sets $m_{A_1}(\texttt{RelCol}_1(c)) = \texttt{RelCol}_1(m_{F_1}(c))$. Similarly, the associative program $A_2 = \texttt{Assoc}(F_1, \texttt{RelRow}_1, \texttt{RelCol}_2)$ constructs a map to every cell in column 2 of the example output table. The table program $\texttt{TabProg}(\{F_1, A_1, A_2\})$ takes the input table in Example 5 and maps to every cell in the output table.

It is possible that the ranges of constituent component programs of a table program may overlap on a given input table. In such a case, if two cells with different values are mapped to the same output coordinate, then we say that the table program is undefined on the input table.

5.2. Synthesis algorithm

The synthesis algorithm generates the set of all table programs that are consistent with each example, intersects the sets, and picks a table program from the intersection that is consistent with all of the examples.

Data structure for sets of table programs: To compactly represent sets of table programs, our synthesis algorithm uses a table program itself. Let a component program K be *consistent* with an input–output example if when K is applied to the example input and K maps an input cell c, then the cell in the output table at coordinate $m_K(c)$ has the same data as cell c in the input table. Let a set of component programs $\mathcal{K}$ *cover* an example if, for each cell coordinate d in the example output, there is some component $K \in \mathcal{K}$ and cell c in the input table such that $d = m_K(c)$. Let a table program $\texttt{TabProg}(\mathcal{K})$ be *consistent with an example* if $\mathcal{K}$ is consistent with the example and $\mathcal{K}$ covers the example. For a fixed input–output example, $\texttt{TabProg}(\mathcal{K})$ *stores* $\texttt{TabProg}(\mathcal{K}')$ if $\mathcal{K}' \subseteq \mathcal{K}$ covers the example.

Procedure `Generate`: From a single input–output example, `Generate` constructs a table program that stores the set of all table programs that are consistent with the example by constructing the set $\mathcal{K}^*$ of all consistent component programs, in three steps. First, from the example input and output, `Generate` defines a set of spatial functions and map predicates. Second, from the set of map predicates, `Generate` collects the set of all consistent filter programs. Third, from the set of consistent filter programs, `Generate` constructs the set of consistent associative programs. To generate associative programs, `Generate` combines each consistent filter program with pairs of spatial functions defined in the first step and checks if the resulting associative program is consistent. If so, then `Generate` adds the associative program to the set of consistent component programs. The table program $\texttt{TabProg}(\mathcal{K}^*)$ stores all table programs that are consistent with the example if any exist, and is thus called the *complete table program* of the example.

Procedure `Intersect`: Given two sets of table programs represented as table programs stored in $\texttt{TabProg}(\mathcal{K}_0)$ and $\texttt{TabProg}(\mathcal{K}_1)$, `Intersect` efficiently constructs the intersection of the sets as all consistent table programs stored in $\texttt{TabProg}(\mathcal{K}_0 \cap \mathcal{K}_1)$.

The synthesis algorithm applies `Generate` to construct the complete table program for each example, applies `Intersect` to the set of complete table programs, and checks if the resulting table program $\texttt{TabProg}(\mathcal{K}_I)$ is consistent with each of the examples. If so, it returns $\texttt{TabProg}(\mathcal{K}'_I)$ for some subset $\mathcal{K}'_I$ of $\mathcal{K}_I$ that covers each of the examples. The exact choice of $\mathcal{K}'_I$ depends on the ranking criteria.

Ranking: We prefer table programs that are constructed from smaller sets of component programs, as such table programs are intuitively simpler. The subset order over sets of component programs thus serves as a partial order for ranking. Also, suppose that a table program P_0 uses a filter program F_0, while another table program P_1 uses a filter program F_1 that builds the same map as F_0, but whose condition is a conjunction of fewer predicates than the condition of F_0. Then, we prefer P_1, as the condition used by F_1 is intuitively more general.

To evaluate our synthesis algorithm, we implemented it as a plug-in for the Excel spreadsheet program and applied it to input–output tasks taken directly from over 50 real-world Excel user help threads. The tool automatically inferred programs for each task. The tool usually ran in under 10 s and inferred a program that behaved as we expected, given the user's description in English of their required transformation. If the highest-ranking program inferred by the tool behaved in an unexpected way on an input, it inferred a program that behaved as expected after taking at most two additional examples.

6. RELATED WORK

The Human Computer Interaction (HCI) community has developed *programming by demonstration* (PBD) systems[1] for data cleaning, which require the user to specify a complete demonstration or trace visually on the data instead of writing code: SMARTedit[14] for text manipulation, Simultaneous Editing[16] for string manipulation, and Wrangler[11] for table transformations. Our system is based on programming by example (as opposed to demonstration); it requires the user to provide only the initial and final states, as opposed to also providing the intermediate states. This renders our system more usable,[13] but at the expense of complicating the learning problem. Furthermore, we synthesize programs over a more sophisticated language consisting of conditionals and loops.

The database community has studied the *view synthesis* problem,[2, 22] which aims to find a succinct query for a given relational view instance. Our semantic string transformation synthesis also infers similar queries, but infers from very few examples and over a richer language that combines lookup operations with syntactic manipulations. Our table layout synthesis addresses the challenges of dealing with spreadsheet tables, which, unlike database relations, have ordering relationships between rows and carry meta-information in some cells.

The PADS project in the programming languages community has simplified ad hoc data-processing tasks for programmers by developing domain-specific languages for describing data formats, and learning algorithms for inferring such formats using annotations provided by the user.[3] The learned format can then be used by programmers to implement custom data-analysis tools. In contrast, we focus on automating the entire end-to-end process for relatively simpler tasks, which include not only learning the text structure from the inputs, but also learning the desired transformation from the outputs, without any user annotations.

The area of program synthesis is gaining renewed interest. However, it has traditionally focused on synthesizing nontrivial algorithms[20] (e.g., graph algorithms[9] and program inverses[19]) and discovering intricate code-snippets (e.g., bit-vector tricks,[7] switching logic in hybrid systems[21]). In this paper, we apply program synthesis to discover simpler programs required by the much larger class of spreadsheet end-users. Various classes of techniques have been explored for program synthesis: exhaustive search, logical reasoning, probabilistic inference, and version-space algebras (for a recent survey, see Gulwani[5]). Because the data manipulations required by end users are usually relatively simple, we can apply version-space algebras, which allow real-time performance, unlike other techniques. Version-space algebras were pioneered by Mitchell for refinement-based learning of Boolean functions,[17] while Lau et al. extended the concept to learning more complex functions in a PBD setting.[14] Our synthesis algorithms lift the concepts of version-space algebra to the PBE setting, for a fairly expressive string expression language involving conditionals and loops.

7. CONCLUSION AND FUTURE WORK

General-purpose computational devices, such as smartphones and computers, are becoming accessible to people at large at an impressive rate. In the future, even robots will become household commodities. Unfortunately, programming such general-purpose platforms has never been easy, because we are still mostly stuck with the model of providing step-by-step, detailed, and syntactically correct instructions on *how* to accomplish a certain task, instead of simply describing *what* the task is. Program synthesis has the potential to revolutionize this landscape, when targeted for the right set of problems and using the right interaction model.

This paper reports our initial experience with designing domain-specific languages and inductive synthesizers for string and table transformations. Our choice of domains was motivated by our study of help-forum problems that end users struggled with. A next step is to develop a general framework that can allow synthesizer writers to easily develop domain-specific synthesizers of the kind described in this paper, similar to how declarative parsing frameworks allow a compiler writer to easily write a parser.

Acknowledgments

We thank Guy L. Steele Jr. for providing insightful and detailed feedback on multiple versions of this draft.

References

1. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*, MIT Press, 1993.
2. Das Sarma, A., Parameswaran, A., Garcia-Molina, H., Widom, J. Synthesizing view definitions from data. In *ICDT* (2010).
3. Fisher, K., Walker, D. The PADS project: an overview. In *ICDT* (2011).
4. Gualtieri, M. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals* (Apr. 2009).
5. Gulwani, S. Dimensions in program synthesis. In *PPDP* (2010).
6. Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *POPL* (2011).
7. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R. Synthesis of loop-free programs. In *PLDI* (2011).
8. Harris, W.R., Gulwani, S. Spreadsheet table transformations from examples. In *PLDI* (2011).
9. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M. A simple inductive synthesis methodology and its applications. In *OOPSLA* (2010).
10. Jha, S., Gulwani, S., Seshia, S., Tiwari, A. Oracle-guided component-based program synthesis. In *ICSE* (2010).
11. Kandel, S., Paepcke, A., Hellerstein, J., Heer, J. Wrangler: Interactive visual specification of data transformation scripts. In *CHI* (2011).
12. Ko, A.J., Myers, B.A., Aung, H.H. Six learning barriers in end-user programming systems. In *VL/HCC* (2004).
13. Lau, T. Why P B D systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI* (2008).
14. Lau, T., Wolfman, S., Domingos, P., Weld, D. Programming by demonstration using version space algebra. *Mach. Learn. 53*(1–2) (2003).
15. Lieberman, H. *Your Wish is My Command: Programming by Example*, Morgan Kaufmann, 2001.
16. Miller, R.C., Myers, B.A., Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference* (2001).
17. Mitchell, T.M. Generalization as search. *Artif. Intell. 18*, 2 (1982).
18. Singh, R., Gulwani, S. Learning semantic string transformations from examples. *PVLDB 5* (2012), in press.
19. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S. Path-based inductive synthesis for program inversion. In *PLDI* (2011).
20. Srivastava, S., Gulwani, S., Foster, J. From program verification to program synthesis. In *POPL* (2010).
21. Taly, A., Gulwani, S., Tiwari, A. Synthesizing switching logic using constraint solving. In *VMCAI* (2009).
22. Tran, Q.T., Chan, C.Y., Parthasarathy, S. Query by output. In *SIGMOD* (2009).
23. Walkenbach, J. *Excel 2010 Formulas*, John Wiley and Sons, 2010.

Sumit Gulwani (sumitg@microsoft.com), Microsoft Research, Redmond, WA.

William R. Harris (wrharris@cs.wisc.edu), Univ. of Wisconsin, Madison, WI.

Rishabh Singh (rishabh@csail.mit.edu), MIT CSAIL, Cambridge, MA.