Programming by Examples

(and its applications in Data Wrangling)

Sumit GULWANI [a,1],
[a] *Microsoft Corporation, Redmond, WA, USA*

Abstract. Programming by Examples (PBE) has the potential to revolutionize end-user programming by enabling end users, most of whom are non-programmers, to create scripts for automating repetitive tasks. PBE involves synthesizing intended programs in an underlying domain-specific language (DSL) from example based specifications (Ispec). We formalize the notion of Ispec and discuss some principles behind designing useful DSLs for synthesis.

A key technical challenge in PBE is to search for programs that are consistent with the Ispec provided by the user. We present a divide-and-conquer based search paradigm that leverages deductive rules and version space algebras for manipulating sets of programs.

Another technical challenge in PBE is to resolve the ambiguity that is inherent in the Ispec. We show how machine learning based ranking techniques can be used to predict an intended program within a set of programs that are consistent with the Ispec. We also present some user interaction models including program navigation and active-learning based conversational clarification that communicate actionable information to the user to help resolve ambiguity in the Ispec.

The above-mentioned concepts are illustrated using practical PBE systems for data wrangling (including FlashFill, FlashExtract, FlashRelate), several of which have already been deployed in the real world.

Keywords. Program Synthesis, Programming by Examples, Inductive Synthesis, Deductive Synthesis, Version space algebras, Active learning, End-user Programming, Spreadsheets, Log files, Data Wrangling, Semi-structured data

1. Introduction

Program synthesis is the task of synthesizing a program that satisfies a given specification [7]. The traditional view of program synthesis has been to synthesize programs from logical specifications that relate the inputs and outputs of the program. A typical academic exercise in program synthesis is to synthesize complicated algorithms such as sorting algorithms [31], graph algorithms [12], and bitvector algorithms [10]. For instance, the logical specification for a sorting algorithm would state that the sorting algorithm takes as input an array $A[1 :: n]$ and outputs another array $B[1 :: n]$ s.t. B is a permutation of A, and B is sorted, i.e.,

[1]Corresponding Author: Sumit Gulwani, One Microsoft Way, Redmond, WA, USA, E-mail: sumitg@microsoft.com

$$\forall 1 \leq i < n : B[i] \leq B[i+1] \quad \wedge$$
$$\exists \sigma, \text{ a permutation of } [1 :: n], \text{such that } \forall 1 \leq i < n : B[i] = A[\sigma(i)]$$

Programming by examples (PBE) is a subfield of program synthesis, where the specification comes in the form of input-output examples. There are two key distinguishing aspects of PBE that motivate dedicated investment in PBE technology. First, program synthesis is a very hard problem in general—in contrast, the nature of example-based specification in PBE makes PBE more tractable than program synthesis since reasoning about concrete input states is much easier than dealing with properties over symbolic program states. Second, writing logical/relational/complete specifications is hard for those even with a programming background—in contrast, PBE can enable non-programmers to create programs for automating repetitive tasks. Today, billions of users have access to computational devices. However, 99% of these end users do not have programming expertise and they often struggle with repetitive tasks in various domains that could otherwise be automated using small scripts. PBE has the potential to revolutionize this landscape since users can often specify their intent using examples as has been observed on various help forums [9].

We start out by describing three useful PBE tools in the space of data wrangling (§2). Then, we present a unifying theory and methodology for PBE (§3).

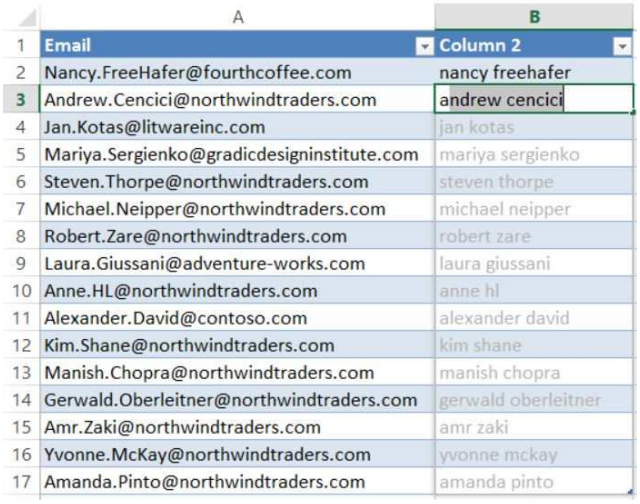
2. Practical Applications in Data Wrangling

PBE has been applied to various domains [5,18], and some recent applications include parsing [17], refactoring [20], query construction [27], and repetitive structured drawings [4]. An important application area is that of household robotics, wherein each household has its own unique geography for the robot to navigate and unique set of chores for the robot to perform—example-based training can be an effective means for programming robots in such settings. However, while we wait for household robotics to become more common, the killer application of PBE today is in the space of data wrangling/-cleaning/manipulation.

Data Wrangling refers to the process of transforming the data from its raw format to a more structured format that is amenable to analysis and visualization. It is estimated that data scientists spend 80% of their time in data wrangling. Data is locked up into documents of various types such as text/log files, semi-structured spreadsheets, webpages, JSON/XML, and pdf documents. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data. However, this makes it extremely hard to extract the underlying data for several tasks such as processing, querying, altering the presentation view, or transforming data to another storage format. PBE can make data wrangling a delightful experience for the masses.

2.1. FlashFill

FlashFill [8,28] is a PBE tool for automating string transformations. The user provides examples of input-output strings, and FlashFill generates a program to perform similar transformations on other input strings.



	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

Figure 1. FlashFill [8]: An Excel 2013 feature that automates repetitive string transformations using examples. Once the user performs one instance of the desired transformation (row 2, col. B) and proceeds to transforming another instance (row 3, col. B), FlashFill learns a program Concatenate(ToLower(Substring(v,WordToken,1)), " ", ToLower(Substring(v,WordToken,2))) that extracts the first two words in input string v (col. A), converts them to lowercase, and concatenates them separated by a space character.

Excel 2013: FlashFill shipped as a feature in Excel 2013, where it allows users to create a new derived column based on existing columns. Figure 1 illustrates this feature. The Excel product team built a good UI that avoids the discoverability issue; the result being that this was a well received feature among popular media [2], which carried quotes like:

- "With some experimentation, you may find that Flash Fill is smarter than you expect" [PC Magazine]
- "Excel 2013's coolest new feature that should have been available years ago" [CNN Money]
- "In fact it's so good it feels like magic" [Tech Radar]
- "Excel Flash Fill Is A Brilliant Time Saver" [Life Hacker]
- "shock-and-awe feature", "Genius", "most notable feature in Excel"
- "Sometimes it just takes a simple new feature in a popular piece of software to remind us how computer science just does cool stuff." [Computational Complexity Blog]

2.2. *FlashExtract*

FlashExtract [15] is a PBE tool for extracting structured (tabular or hierarchical) data from semi-structured text/log files and webpages. For each field in the output data schema, the user provides positive/negative instances of that field and FlashExtract generates a program to extract all instances of that field. Figure 2 illustrates the capability of this tool.

[2] http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html

Label 1	Label 2	Label 3
Ana Trujillo	Redmond	(757) 555-1634
Antonio Moreno	Renton	(411) 555-2786
Thomas Hardy	Seattle	(412) 555-5719
Christina Berglund	Redmond	(443) 555-6774
Hanna Moos	Puyallup	(376) 555-2462
Frederique Citeaux	Redmond	(689) 555-2770

Figure 2. FlashExtract [15]: A PBE technology for extracting data from text files and web pages using examples. Once the user highlights one or two examples of each field in a different color (in the text file on the left side), FlashExtract extracts more such instances and arranges them in a structured data format (table on the right side).

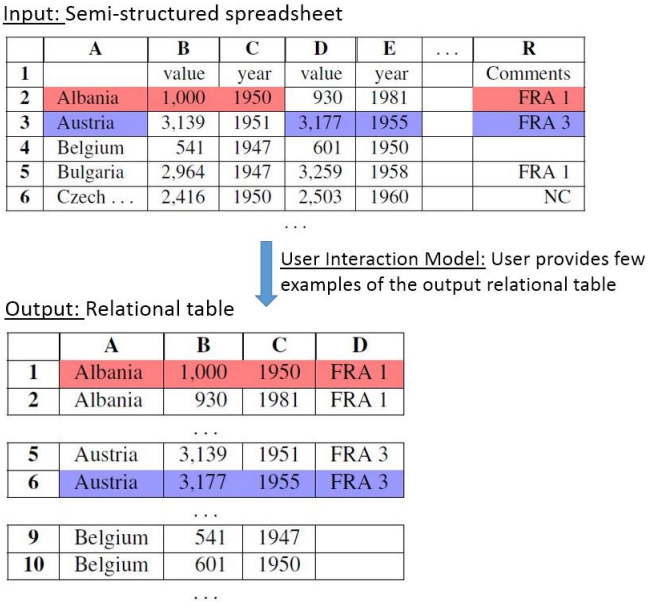
Powershell: FlashExtract shipped as the *ConvertFrom-String* cmdlet in Powershell in Windows 10, wherein the user provides examples of the strings to be extracted by inserting tags around them in text. [3] Since then, several Microsoft MVPs (Most Valued Professionals) have built UI experiences on top of this cmdlet. Many blogs publicized this feature [4], and labeled it as "New kid on the block", "This is super cool !!", "Serious Text wrangling", and "must admit that this cmdlet is to me one of the best improvement that came with WMF5.0 and Powershell v5".

Operations Management Suite: This product is a new Saas service for IT Pros to collect machine data from any cloud and get operational insights. The FlashExtract capability has been surfaced in this product to allow these IT Pros to extract custom fields from log files [5].

2.3. FlashRelate

FlashRelate [2] is a PBE tool for extracting tabular/relational data out of semi-structured spreadsheets. The user provides examples of tuples in the output table, and FlashRelate generates a program to extract more such tuples from the input semi-structured spreadsheet. Figure 3 illustrates the capability of FlashRelate.

[3] `http://bit.ly/convertfrom-string`
[4] `http://research.microsoft.com/en-us/um/people/sumitg/flashextract.html`
[5] `http://tinyurl.com/oms-custom-fields`



	A	B	C	D	E	...	R
1		value	year	value	year		Comments
2	Albania	1,000	1950	930	1981		FRA 1
3	Austria	3,139	1951	3,177	1955		FRA 3
4	Belgium	541	1947	601	1950		
5	Bulgaria	2,964	1947	3,259	1958		FRA 1
6	Czech ...	2,416	1950	2,503	1960		NC

...

	A	B	C	D
1	Albania	1,000	1950	FRA 1
2	Albania	930	1981	FRA 1

...

5	Austria	3,139	1951	FRA 3
6	Austria	3,177	1955	FRA 3

...

9	Belgium	541	1947	
10	Belgium	601	1950	

...

Figure 3. FlashRelate [2]: A PBE technology that can transform the top semi-structured table into the bottom structured table once the user provides a couple of examples of the tuples in the output table (for instance, the highlighted ones).

Spreadsheets are easy to use since they allow users great flexibility to store data. This flexibility comes at a price: users often treat spreadsheets as a poor man's database, leading to creative solutions for storing high-dimensional data by using spatial layouts involving headers, whitespace, and relative positioning. Thus while spreadsheets allow compact and intuitive visual representations of data well suited for human understanding, their flexibility complicates the use of powerful data-manipulation tools (e.g., PowerBI, relational query engines) that expect data in a certain form. We call these spreadsheets semi-structured because their data is in a regular format that is nonetheless inaccessible to data-processing tools. It is conjectured that around 50% of spreadsheets contain data in this not-easy-to-use format.

3. Overview

We now highlight the two main challenges in designing a good PBE system, and present a brief overview of our approach in handling those challenges. This section also provides a roadmap for the various technical sections in this article.

Challenge 1: Search Algorithm

An important challenge in PBE relates to designing a real-time search algorithm that can facilitate an interactive session with the user. One of our key ideas is to restrict the search over an appropriate domain-specific language (DSL), as also suggested by the recent SyGuS methodology [1]. Another key idea is to exploit the semantics of the

operators in the underlying DSL to perform an intelligent search as opposed to, say, only performing an enumerative brute-force search. Each of the PBE systems described in §2 is associated with a DSL and a search algorithm that is specialized to searching over that DSL. Gulwani et.al. describe this general methodology in a CACM research highlights article [9].

It turns out that there is a unifying theme that runs beneath each of those DSL-specialized search algorithms. We capture this theme as part of a generic PBE search algorithm that is parameterized by a DSL. The generic search algorithm (§7) takes as input an example-based specification (§4), a DSL (§5), and produces a set of programs in that DSL that are consistent with the example-based specification. The set of programs synthesized by the search algorithm are represented succinctly using a data-structure called version space algebras (§6).

Challenge 2: Dealing with Ambiguity

Examples are an ambiguous form of specification: there can be different programs that are consistent with the provided examples, but these programs differ in their behavior on some other inputs. If the user does not provide a large set of representative examples, the PBE system may synthesize unintended programs. In 2009, Tessa Lau presented a critical discussion of PBE systems noting that adoption of PBE systems is not yet widespread, and proposing that this is mainly due to lack of usability and confidence in such systems [13].

We present two complementary techniques that alleviate usability and confidence issues that hamper adoption of PBE systems. First, we discuss use of ranking techniques to predict an intended program from an under-representative specification (§8)—this increases usability of the PBE system. Second, we discuss user interaction models for PBE that provide transparency in the working of the underlying system and help resolve ambiguities in the specification (§9)—this increases user confidence in the underlying system.

4. Inductive Specification

Inductive synthesis or PBE traditionally refers to the process of synthesizing a program from input-output examples. More formally, the specification consists of *conjunction* of pure example-based constraints, where each constraint is specified by a pair of concrete input state σ and concrete output value, and constrains the synthesized program to generate the specified output value on input σ. We generalize this formalism in two ways: (a) Instead of requiring the concrete value of the output, we allow an arbitrary predicate over the output. (b) Furthermore, we allow Boolean connectives over those predicates. We refer to the resulting specification kind as *Ispec*, denoted by the symbol ψ, which is a conjunction of pairs $\langle \sigma, \phi \rangle$, where σ is an input state and ϕ is a unary Boolean formula over a program's output o.

Generalization (a) is useful when describing the complete output on a given input is too cumbersome or not useful, but the user can easily express part of the output or more generally some properties of the output. For instance, consider the repetitive task of extracting a list L of strings from a given document D. The input in this case is the

document D and the output is the desired list of strings. If the user were to provide the complete output list, there might be nothing left to automate (unless the user wants to run the synthesized script on another document of the same kind—even in that case, providing the complete output would be time consuming and error prone). An alternative of constructing a mock input [33] document that is small and representative would be rather cumbersome. However, the user can easily provide a few examples of strings that *should belong to the output list L*. The user can also easily provide examples of strings that *should not belong to the output list*; this is useful when the underlying synthesizer learns a program that extracts some unintended strings. The user may also provide a *contiguous subsequence of strings in L* (as opposed to providing an arbitrary sample) or for that matter, specify a *prefix of the output list*; in either of these cases, the user has implicitly also asserted what does not belong to the output list. Such intent can be easily communicated by the user and also provides valuable information to the synthesizer.

The generalization to allowing Boolean connectives (including disjunctions), arises as part of the specification refinement process that happens internally in the synthesizer algorithms. Specifications on DSL operators get refined into specifications on operator parameters, and the latter specifications are often shaped as arbitrary Boolean formulas. For instance, in FlashFill, the problem of synthesizing a substring operator that extracts a given substring s from a given input string v is reduced to the problem of synthesizing position extraction logics that return any occurrence of s in v (which is a disjunctive specification).

An Ispec is not only much easier to provide for a user (compared to declarative specifications), but also allows efficient synthesis algorithms. The presence of input states in the specification allow us to only consider the program behavior on those specific input states as opposed to analyzing the program behavior over all possible inputs. Thus, even a simple enumerative search strategy can cluster program expressions that produce the same result on those input states. Most significantly, a top-down search strategy becomes feasible, wherein Ispec for a program expression can be reduced to Ispec for its sub-expressions.

5. Domain Specific Language

We use functional domain-specific languages to restrict the search space for a program synthesizer. These languages are characterized by a set of operators, and a syntactic restriction on how those operators can be composed with each other (as opposed to allowing all possible type-safe composition of those operators).

The choice of a domain-specific language should be guided by various factors.

- Balanced Expressivity: On one hand, the DSL should be expressive enough to represent a wide variety of tasks in the underlying task domain. But, on the other hand, it should be restricted enough to allow efficient search.
- Choice of Operators: The DSL should be made up of operators that have efficient *witness functions* (§7) in order to allow an efficient top-down deductive search strategy.
- Naturalness: The programs in the DSL should involve natural computational patterns that can be easily understood by the users. This can increase user's confidence in the system. In fact, these computational patterns should be similar to how

programmers might have written the code themselves. These programs might be read by users, who might then select between these programs, edit them, and even use them as part of larger workflows.

The FlashFill DSL represents programs that transform an n-ary tuple of strings into another string. These programs involve computing substrings of the strings in the input tuple, and then concatenating them appropriately. There is also support for restricted forms of loops that concatenate a sequence of substrings from the input string to facilitate more sophisticated string transformations as in abbreviation computation or string reversal. There is support for conditional computation at the very top level to account for multiple data formats in the input.

The FlashExtract DSL represents programs that transform a large string (representing a semi-structured text/log file) into a list of strings. These programs involve splitting the file into a list of lines, filtering this list, and then mapping it to another list using a substring operator.

The FlashRelate DSL represents programs that transform a two-dimensional array (representing a semi-structured spreadsheet) into a list of n-tuples (representing a relational table). These programs involve computing one of the columns in the output relational table using a filter operation over the two-dimensional array, followed by deriving other columns as map operations over an existing column using spatial offsets in the input spreadsheet.

A DSL can be specified using a context-free grammar. The use of "let" construct allows imposing useful restrictions on the allowed set of programs. We also allow (explicitly) paramterizing a non-terminal by the set of all free variables that occur in the program expressions that the non-terminal expands to. Such a parameterization not only makes it easy to understand the various variable bindings but also makes it easy to share sub-languages across various DSLs.

5.1. Case study: Language for substring computation

We discuss below an iterative design of a language for substring computation that is used as a sub-language inside both FlashFill and FlashExtract DSLs. This discussion reflects the various design principles behind designing languages for synthesis.

Consider the following natural choice for the substring operator, whose semantics is to extract the first match of regular expression R in string X.

$$\texttt{Substring}(X, R)$$

If R is a simple or basic regular expression, this operator is not very expressive since it does not take into account the context around the data to be extracted. For instance, it cannot describe extraction of "second word", or "content within parentheses". On the other hand, if we allow R to be an extended regular expression that allows binding variables to parts of the match, we get too expressive of a construct that hinders both learnability and readability. So, we seek a construct that involves simple regular expressions but also takes context into account.

We thus use the following choice for the substring operator, which takes as input a string X, and two position expressions P and P' that evaluate to positions/indices within the string X, and returns the substring between those positions.

$$\texttt{Substring}(X, P, P')$$

Position expr $P := \texttt{Pos}(X, R_1, R_2, K)$

The choice for position expression P includes the $\texttt{Pos}(X, R_1, R_2, K)$ operator, which returns the K^{th} position within the string X such that (some suffix of) the left side of that position matches with regular expression R_1 and (some prefix of) the right side of that position matches with regular expression R_2. Consider the expression $e = \texttt{Substring}(x, p_1, p_2)$, where $p_1 = \texttt{Pos}(x, r_1, r_2, k)$ and $p_2 = \texttt{Pos}(x, r_1', r_2', k')$. When $r_1 = r_2' = \varepsilon$, e describes the substring. When $r_2 = r_1' = \varepsilon$, e describes the context around the substring. The general case is thus very expressive, allowing properties of both the substring to be extracted and its context. Furthermore, each of the involved regular expressions is simple. Following are some substring tasks that can now be expressed:

- Second word: $\texttt{Substring}(x, \texttt{Pos}(x, \varepsilon, \mathit{Word}, 2), \texttt{Pos}(x, \mathit{Word}, \varepsilon, 2))$
- Content within brackets: $\texttt{Substring}(x, \texttt{Pos}(s, \text{`['}, \varepsilon, 1), \texttt{Pos}(x, \varepsilon, \text{`]'}, 1))$
- Last 7 characters: $\texttt{Substring}(x, \texttt{Pos}(x, \varepsilon, \varepsilon, \text{-}7))$

The Substring language as defined above does not restrict the occurrences of non-terminal X to evaluate to the same string expression and is thus unnecessarily too general. We can fix this by using a "let" construct.

$$\text{let } x = X \text{ in}$$

$$\texttt{Substring}(x, P, P')$$

Position expr $P := \texttt{Pos}(x, R_1, R_2, K)$

When we look at the definition of the non-terminal P, it is not immediately clear what the free variable x is bound to. We fix this by explicitly parameterizing the free variable in the definition of P.

$$\text{let } x = X \text{ in}$$

$$\text{let } p_1 = P[x] \text{ in}$$

$$\text{let } p_2 = P[x] \text{ in}$$

$$\texttt{Substring}(x, p_1, p_2)$$

Position expr $P[y] := \texttt{Pos}(y, R_1, R_2, K)$

While the Substring language above can express "the second word" or "the first 7 characters", it cannot express "the first 7 characters of the second word". We enable such expressiveness by allowing the ending position of the substring to also be expressed relative to that of the starting position of the substring. Below, `Suffix` (x,p) denotes the suffix of string x starting after position p.

$$\text{let } x = X \text{ in}$$

$$\text{let } p_1 = P[x] \text{ in}$$

$$\text{let } p_2 = P[x] \mid p_1 + P[\texttt{Suffix}(x, p_1)] \text{ in}$$

$$\texttt{Substring}(x, p_1, p_2)$$

Position expr $P[y] := \texttt{Pos}(y, R_1, R_2, K)$

Now, we can express "first 7 characters in second word" as: $\texttt{Substring}(x, p_1, p_1 + 7)$, where $p_1 = \texttt{Pos}(x, \varepsilon, Word, 2)$.

The following is another extension that enables computing the starting position of the substring relative to that of another position in the substring.

$$\begin{aligned}
&\text{let } x = X \text{ in} \\
&\text{let } p_1 = P[x] \mid (\text{let } p_0 = P[x] \text{ in } (p_0 + P[\texttt{Suffix}(s, p_0)])) \text{ in} \\
&\text{let } p_2 = P[x] \mid p_1 + P[\texttt{Suffix}(x, p_1)] \text{ in} \\
&\texttt{Substring}(x, p_1, p_2) \\
\text{Position expr } P[y] &:= \texttt{Pos}(y, R_1, R_2, K)
\end{aligned}$$

This allows expressing "second word within brackets" as: $\texttt{Substring}(x, p_1, p_1 + \texttt{Pos}(\texttt{Suffix}(x, p_1), Word, \varepsilon, 1))$, where $p_1 = p_0 + \texttt{Pos}(\texttt{Suffix}(x, p_0), \varepsilon, Word, 2)$ and $p_0 = \texttt{Pos}(x, `[', \varepsilon, 1)$.

6. Version Space Algebras

Our PBE methodology manipulates *sets* of programs, for two reasons: First, since examples are an ambiguous form of specification, there are many programs that are consistent with the example-based specification. We might want to compute many/all of them in order to drive various user interaction models (§9). Second, the divide-and-conquer based deductive search strategy (§7) often requires computing the set of all solutions to the sub-problems in order to construct a solution to the top-level problem. In other words, the top-down deductive search strategy often requires computing the set of all (sub-)expressions that satisfy appropriate (sub-)specifications to synthesize expression(s) that satisfy a given specification.

The sets of programs that thus arise are often huge, several powers of 10. Even *representing* them explicitly would not be feasible, let along operating over such explicit set representations. However, it turns out that the programs in these sets share subexpressions, and as a result these programs can be represented succinctly using appropriate data structures, referred to as *Version-space algebras* (VSAs). VSAs were initially defined by Mitchell [22] in the context of machine learning and were later used by Lau et.al. for programming by demonstration [14], but were restricted to tree-based representations. We have generalized the notion of VSAs to graph-based representations and have also defined various useful operations over it [24].

A VSA data structure is a directed (and often acyclic) graph, where each node represents a set of program expressions. A leaf node is annotated with a set of program expressions and it represents the set containing those expressions. There are two kinds of parent nodes. A (parent) *union* node simply has the semantics that it represents the union of the sets of programs that are represented by its children nodes. A (parent) *join* node with n children is annotated with an n-ary operator F with the semantics that it represents the set of all program expressions of the kind $F(e_1, \ldots, e_n)$, where e_i belongs to the set of program expressions represented by the i^{th} child of the node.

Note that a VSA data structure provides two kinds of sharing among program expressions. One is provided by the join node that represents the set of programs obtained

by taking the cross-product of the sets of programs represented by its children. The other sharing is provided by virtue of having multiple incoming edges into a node, which represents the fact that we do not create two different copies of the same set of program expressions.

There are several useful operations that can be defined over VSAs in a domain-independent manner.

- Union: VSA $\times$ VSA $\rightarrow$ VSA
  The Union operation is implemented by simply creating a union VSA node whose children are the input VSAs.
- Intersect: VSA $\times$ VSA $\rightarrow$ VSA
  The Intersect operation is performed by computing a cross-product of the two input VSAs using an algorithm similar to that of automata intersection.
- TopRank: VSA $\times$ Ranking function $\times\, k \rightarrow$ Set of k top-ranked programs
  The TopRank operation is implemented recursively by taking the top-ranked k programs from the children VSAs associated with the top-level operator and then identifying the top-ranked k programs at the top-level. This requires the ranking function to be monotonic over an expression structure (higher-ranked subexpressions produce higher-ranked expressions).
- Cluster: VSA $\times$ Set of input states $\rightarrow \{\text{VSA}_i\}_i$
  The Cluster operation takes as input a VSA and a set of input states $\tilde{\sigma}$, and partitions the input VSA into the smallest output set of VSAs $\{\text{VSA}_i\}_i$ such that the union of the sets of the programs represented by $\text{VSA}_1, \ldots \text{VSA}_n$ equals the set of programs represented by the input VSA, and all programs in any VSA_i produce the same output on any input in $\tilde{\sigma}$. The Cluster operation is implemented recursively by clustering the children VSAs and then appropriately building top-level clusters and merging some.
- Filter: VSA $\times\, \psi \rightarrow$ VSA
  The Filter operation takes as input a VSA and an inductive specification ψ and produces the (largest) subset of the input VSA that satisfies ψ. The Filter operation can be implemented by performing clustering based on all inputs states present in ψ, deleting those top-level nodes that do not satisfy ψ, and then merging nodes to regain any sharing.

7. Search Algorithm

A simple search strategy would be to *enumerate* all programs in the underlying DSL, say in order of increasing size [32]. However, this approach alone will not scale for expressive DSLs of the kind used inside PBE tools described in §2. Another alternative strategy would be to reduce the (second-order) search problem to (first-order) *constraint solving* using various recent techniques from literature [30,31,10], and leverage off-the-shelf SAT/SMT constraint solvers like Z3 [3]. This allows leveraging the huge engineering advances that have been made in SAT/SMT constraint solving in an easy manner. Unfortunately, our experience with this strategy has been that it is not very robust and real-time. Furthermore, there is no easy way to incorporate ranking or to enumerate all/many solutions (useful for ambiguity resolution). Similar challenges exist with *stochastic search techniques* [26].

Our novel deductive search methodology [24] is based on standard algorithmic paradigm of divide-and-conquer. It recursively reduces the problem of synthesizing a program expression e of a certain kind and that satisfies a certain inductive specification ψ to simpler sub-problems (where the search is either over sub-expressions of e or over sub-specifications of ψ), followed by appropriately combining those results. The *reduction logic* for reducing a synthesis problem to simpler synthesis problems depends on the nature of the involved expression e and the inductive specification ψ. Observe that, in contrast to enumerative search, this search methodology is top-down, where it fixes the top-part of an expression and then searches for its sub-expressions. Enumerative search is typically bottom-up, where it enumerates smaller sub-expressions before enumerating larger expressions.

We start out by discussing the notion of witness functions, which is used in defining reduction logics for function applications (§7.2).

7.1. Witness Functions

A *witness function* for an operator is a backward transformer (i.e., a transformer for computing preconditions) that translates the specification on the output of that operator to a specification on the parameters of that operator. We require the resultant specification to be in DNF (disjunctive normal form), where each disjunct consists of conjunctions of constraints, each of which involves only one of the parameters. A witness function is said to be *sound* if it generates an under-approximation to the weakest precondition (i.e., a sufficient characterization). A sound witness function is said to be *precise* if it generates the weakest precondition (i.e., a necessary and sufficient characterization). Often one can design good heuristics to make a witness function efficient, albeit at the cost of making it theoretically imprecise by not considering unlikely cases.

We give below several instances of witness functions. We use the notation $e \models \psi$ to denote the predicate $\psi[e/o]$, i.e., the predicate obtained from ψ by replacing o by e.

Witness Function for Concatenate operator

Consider the operator `Concatenate` : `String` $\times$ `String` $\rightarrow$ `String`, which takes as input two strings and concatenates them. Consider the witness function for the `Concatenate` operator that transforms an equality specification by non-deterministically splitting the output string into two parts and asserting that the prefix is generated by the first parameter and the suffix is generated by the second parameter. Following is an application instance of this precise witness function.

$$\texttt{Concatenate}(e_1, e_2) \models \langle \sigma, o = \text{“Abc”} \rangle \iff \bigvee_{i=0}^{3} (e_1 \models \langle \sigma, o = \text{“Abc”}[0,i] \rangle \ \wedge \ e_2 \models \langle \sigma, o = \text{“Abc”}[i,3] \rangle) \quad (1)$$

where $s[i,j]$ denotes the substring of string s between the i^{th} and the j^{th} positions.

Suppose the `Concatenate` operator has a strong type signature in the underlying DSL where it takes as input two non-empty strings.

`Concatenate` : Non-empty`String` $\times$ Non-empty`String` $\rightarrow$ Non-empty`String`

Then, its above-mentioned witness function can be strengthened to not consider splits where the prefix or suffix is empty. The application in Eq. 1 can then be strengthened as follows:

$$\texttt{Concatenate}(e_1, e_2) \models \langle \sigma, o = \text{“Abc”} \rangle \iff \bigvee_{i=1}^{2} (e_1 \models \langle \sigma, o = \text{“Abc”}[0, i] \rangle \wedge e_2 \models \langle \sigma, o = \text{“Abc”}[i, 3] \rangle) \quad (2)$$

A good heuristic to make the above-mentioned witness function more efficient is to avoid those cases that split the output string across standard character class boundaries such as [a-z], [A-Z], [0-9]. The application in Eq. 2 can be further strengthened as follows:

$$\texttt{Concatenate}(e_1, e_2) \models \langle \sigma, o = \text{“Abc”} \rangle \Leftarrow e_1 \models \langle \sigma, o = \text{“A”} \rangle \wedge e_2 \models \langle \sigma, o = \text{“bc”} \rangle$$

More sophisticated heuristics can inspect the input strings to determine the likely case splits of the output string.

Witness function for Substring operator

Consider the operator $\texttt{Substring} : \texttt{String} \times \texttt{Position} \times \texttt{Position} \rightarrow \texttt{String}$, which takes as input a string, and two positions within that string, and returns the substring between those two positions. Consider the witness function for the `Substring` operator that reduces an equality specification by identifying all occurrences of the output string within the input string, and asserting that the position expression parameters of the `Substring` operator should evaluate to respective positions. Following is an application instance of this precise witness function.

$$\begin{aligned}\texttt{Substring}(x, p_1, p_2) \models \langle x : \text{“Ab cd Ab”}, o = \text{“Ab”} \rangle \iff \\ (p_1 \models \langle x : \text{“Ab cd Ab”}, o = 0 \rangle \wedge p_2 \models \langle x : \text{“Ab cd Ab”}, o = 2 \rangle) \vee \\ (p_1 \models \langle x : \text{“Ab cd Ab”}, o = 6 \rangle \wedge p_2 \models \langle x : \text{“Ab cd Ab”}, o = 8 \rangle)\end{aligned}$$

Witness function for Map operator

Consider the standard map operator $\texttt{Map} : (\text{Function } f : (T_1 \rightarrow T_2)) \times \texttt{List}(T_1) \rightarrow \texttt{List}(T_2)$, which takes as input a list and a function f that operates over elements of that list, and returns another list by applying the input function to each element of the input list. There is no useful witness function for a generic map operator. However, for strongly typed `Map` operators, where we know something about the behavior of the two parameters to `Map`, it might be possible to define a witness function. One such general condition is the existence of an inverse g to the function f under the constraints established by the rich types associated with the parameters of the Map operator. In such a case, a precise witness function can be defined for prefix/suffix/subsequence specifications, which assert that certain specific elements c should belong to the output list (in an input state σ). The key idea is to replace any output list element c by $g(c)$ and assert that the resultant specification should be satisfied by the second parameter of the `Map` operator. For the

first parameter f of the `Map` operator, assert that it should satisfy $\bigwedge_c \langle \sigma[x \mapsto g(c)], o = c \rangle$, i.e., for each element c, it should transform the input state, extended with the function argument x bound to $g(c)$, to c.

For instance, consider the strongly typed `Map` operator in the FlashExtract language whose second parameter resolves to a list of line regions from the input text file and whose first parameter is a function f that produces a substring region of its input region. Then, the function g simply maps a region s_i (nested within a line) to its enclosing line region ℓ_i. Following is an application instance of the above-mentioned witness function for this strongly typed `Map` operator.

$$\texttt{Map}(\lambda x : e_1, e_2) \models \langle \sigma, \texttt{Prefix}([s_1, s_2], o) \rangle = \\ e_2 \models \langle \sigma, \texttt{Prefix}([\ell_1, \ell_2], o) \rangle \wedge e_1 \models \bigwedge_{i=1,2} \langle \sigma[x \mapsto \ell_i], o = s_i \rangle$$

Witness function for If-then-else operator

Consider the if-then-else operator $\texttt{ITE} : \texttt{Boolean Expr} \times T \times T \to T$ with the standard semantics. Consider the witness function for the `ITE` operator that non-deterministically partitions the input states in the specification into two sets and asserts that the Boolean expression is such that it evaluates to true on all states in one partition and to false on all states in the other partition, and that the respective branches handle the specifications associated with those states. Following is an application instance of this precise witness function.

$$\texttt{ITE}(B, S_1, S_2) \models \langle \sigma_1, \phi_1 \rangle \wedge \langle \sigma_2, \phi_2 \rangle \iff \gamma_1 \vee \gamma_2 \vee \gamma_3 \vee \gamma_4, \text{ where}$$

$$\gamma_1 = B \models \langle \sigma_1, o = true \rangle \wedge \langle \sigma_2, o = false \rangle \wedge S_1 \models \langle \sigma_1, \phi_1 \rangle \wedge S_2 \models \langle \sigma_2, \phi_2 \rangle$$

$$\gamma_2 = B \models \langle \sigma_1, o = true \rangle \wedge \langle \sigma_2, o = true \rangle \wedge S_1 \models \langle \sigma_1, \phi_1 \rangle \wedge \langle \sigma_2, \phi_2 \rangle$$

$$\gamma_3 = B \models \langle \sigma_1, o = false \rangle \wedge \langle \sigma_2, o = true \rangle \wedge S_1 \models \langle \sigma_2, \phi_2 \rangle \wedge S_2 \models \langle \sigma_1, \phi_1 \rangle$$

$$\gamma_4 = B \models \langle \sigma_1, o = false \rangle \wedge \langle \sigma_2, o = false \rangle \wedge S_2 \models \langle \sigma_1, \phi_1 \rangle \wedge \langle \sigma_2, \phi_2 \rangle$$

If the choice for B is closed under negation, and that S_1 and S_2 are the same non-terminals, then it suffices to only consider cases γ_1 and γ_2. If we further make the assumption that the user has provided representative scenarios, we can rule out γ_2.

7.2. Reduction Logics

There are several reduction logics that apply to general expressions and specifications, regardless of any domain. Let e be any non-terminal or right-hand side of any production rule in the underlying DSL. We use the notation $\{e \models \psi\}$ to denote the set of program expressions of kind e that satisfy the inductive specification ψ.

The following reduction logics handle Boolean connectives in the specification.

$$\{e \models \langle \sigma, \phi_1 \vee \phi_2 \rangle\} = \text{Union}(\{e \models \langle \sigma, \phi_1 \rangle\}, \{e \models \langle \sigma, \phi_2 \rangle\})$$

$$\{e \models \langle \sigma, \phi_1 \wedge \phi_2 \rangle\} = \text{Intersect}(\{e \models \langle \sigma, \phi_1 \rangle\}, \{e \models \langle \sigma, \phi_2 \rangle\})$$

$$\{e \models \psi_1 \wedge \psi_2\} = \text{Intersect}(\{e \models \psi_1\}, \{e \models \psi_2\})$$

The Intersect and Union operations are the ones that are supported by the VSA data-structure (§6).

The use of Intersect operation in the above reduction logics might be expensive because of the quadratic blowup in the implementation of the Intersect operation. Another alternative strategy to handle conjunctive specifications is as follows, where Filter operation is another one of those operations supported by the VSA data-structure.

$$\{e \models \psi_1 \wedge \psi_2\} = \text{Filter}(\{e \models \psi_1\}, \psi_2)$$

The following reduction logic applies when e is a non-terminal. Suppose e is defined to be either e_1 or e_2 in the underlying DSL. Then,

$$\{e \models \psi\} = \text{Union}(\{e_1 \models \psi\}, \{e_2 \models \psi\})$$

We now discuss the case when e is a function application, say $F(e_1, e_2)$. Suppose the witness function for F translates the specification over function application, namely $F(e_1, e_2) \models \psi$, into $\bigvee_i e_1 \models \psi_i \wedge e_2 \models \psi_i'(e_2)$ (specification over function parameters). Then, the following reduction logic applies:

$$\{F(e_1, e_2) \models \psi\} = \bigcup_i (F(\{e_1 \models \psi_i\}, \{e_2 \models \psi_i'\}))$$

If the witness function for F is sound, the above reduction logic leads to computation of correct solutions. Otherwise, it can be refined as follows:

$$\{F(e_1, e_2) \models \psi\} = \text{Filter}(\bigcup_i (F(\{e_1 \models \psi_i\}, \{e_2 \models \psi_i'\})), \psi)$$

If the witness function for F is precise, the above reduction logics preserves all correct solutions. If a precise witness function is not efficient, we can employ a multi-phase approach, wherein synthesis is first performed using efficient (but imprecise) witness functions, and if no satisfactory solution is returned, synthesis is then repeated using precise witness functions.

8. Ranking

There are often many programs in an underlying DSL that are consistent with a given set of training examples. If the user does not provide a representative set of input-output examples, several of these programs might be unintended, i.e., they would produce an undesired behavior on some other test input. Hence, the simple strategy of picking an arbitrary program from this set might not yield an intended program. Insistence on a GIGO (Garbage in, Garbage out) philosophy requiring users to always provide representative inputs can hinder the usability of PBE technologies. We address the problem of learning an intended program from a small number of examples by leveraging ranking functions over programs.

8.1. Program features

A basic ranking scheme can be specified by defining a preference order over program expressions based on their features [8]. Two general principles that are useful across various domains are: **prefer smaller expressions** (inspired by the classic notion of Kolmogorov complexity) and **prefer expressions with fewer constants** (to force generalization).

Suppose the DSL underlying a PBE system includes constant expressions and conditional expressions with equality predicates. Then, if the user provides a set of input-output examples $\{(i_1, o_1), (i_2, o_2), \ldots, (i_n, o_n)\}$, then the PBE system might generate a program that is simply a sequence of conditionals with constant branch expressions, namely

if $(i = i_1)$ then o_1
else if $(i = i_2)$ then o_2
$\ldots$
else if $(i = i_n)$ then o_n

A useful generalization can be enforced by having the ranking scheme prefer smaller expressions over larger expressions (which in this case can be a program with no or smaller number of conditionals that perform a more non-trivial computation than a simple case split).

Suppose the DSL underlying a PBE system includes constant expressions. If the user provides an input-output example (i, o), then the PBE system (if guided only by the preference to pick smaller expressions) might generate a program that is simply the constant expression o. A useful generalization can be enforced by having the ranking scheme also prefer non-constant expressions above constant expressions.

For specific DSLs, more specific preferences can be defined based on the operators that occur in their expressions. These preferences should be such that it should allow efficient computation of top-ranked programs from a VSA representation. For instance, one can define a **partial order between different production rules** for each non-terminal in the DSL.

8.2. Data features

The likelihood of a program being the intended one not only depends on the structure of that program, but also on features of the input data on which that program will be executed and the output data produced by executing that program. Consider the task of extracting years from input strings of the kind shown in the table below.

Input	Output
Missing page numbers, 1993	1993
64-67, 1995	1995

The program P1: "Extract 1^{st} number from the end" can perform the intended task. However, if the user provides only the first example, another reasonable program that can be synthesized is P2: "Extract 1^{st} number from the beginning". There is no clear way to rank $P1$ higher than $P2$ from just examining their structure. However, the output pro-

duced by $P1$ (on the various test inputs), namely $\{1993, 1995, ...\}$ is a more meaningful set (of 4 digit numbers that are likely years) than the one produced by $P2$, namely $\{1993, 64\}$ (which manifests greater variability). The meaningfulness or similarity of the generated output can be captured via various features such as IsYear, numeric deviation, IsPersonName, and number of characters.

8.3. Learning weights using Machine Learning

We have argued above the need for leveraging various features, related to both program structure and test data, for ranking. The presence of multiple features for ranking leads to the question: How much relative weightage do we give to the various features in order to compute a final ranking score that can be used to compare the likelihood of various programs that are consistent with the user specification. The weights for various features can be learned using machine learning techniques in an offline manner [28]. There are two key aspects in learning these weights, namely generation of training data, and constraints on what these weights should be.

The training data is defined by a collection of tasks. A task is associated with a set of *intended programs* in the DSL that can correctly perform that task; note that there can be multiple programs in a DSL that have the same intended behavior on the set of test inputs that the user cares about. The set S_1 of intended programs can be generated by providing a representative enough specification to the underlying synthesizer. Now, for each such task, we can also generate the set S_2 of programs that match any weak form of that representative specification (for instance, say the set of programs that are induced by any one input-output example out of the 3 input-output examples in the representative specification). Note that S_2 is a superset of S_1. Now, we want the weights to be such that any of the (intended) programs in S_1 should be ranked higher than each (unintended) program in $S_2 - S_1$. One way to learn these weights is to employ a gradient descent based method to optimize an appropriate loss function that aims to rank any intended program higher than all unintended programs [28].

9. User Interaction Models

While use of ranking in the synthesis methodology (as discussed in §8) attempts to avoid selecting an unintended program, it cannot always succeed. Hence, it is important to design appropriate user interaction models for the PBE paradigm that can provide the equivalent of debugging experience in standard programming environments.

There are two important goals for a user interaction model that is associated with a PBE technology [19]. First, it should provide transparency to the user about the synthesized program(s). Second, it should guide the user in resolving ambiguities in the provided specification. (The ranking scheme, which tries to avoid unintended programs, cannot always succeed.) §9.2 and §9.3 describe two complementary user interaction models towards these goals.

9.1. Motivational Case Studies

FlashFill: The PBE engine behind FlashFill, which was released as an Excel 2013 feature, received many positive reviews from popular media. However, the user interface

for FlashFill left a lot to be desired. The FlashFill UI simply executes the highest ranked program synthesized by FlashFill on new inputs without displaying the synthesized program. While this has the advantage of simplicity, it does not inspire user confidence on sensitive data. John Walkenbach, an author renowned for his Excel textbooks, labeled FlashFill as a "controversial" feature. He wrote "It's a great concept, but it can also lead to lots of bad data. (...) Be very careful. (...) [M]ost of the extracted data will be fine. But there might be exceptions that you don't notice unless you examine the results very carefully."[6]

FlashExtract: The PBE engine behind FlashExtract, which powers the *ConvertFrom-String* cmdlet in Powershell, was very well received by Microsoft MVPs (Most Valued Professionals). However, the MVPs also complained that they had no visibility into the process for debugging purposes. This prompted release of an improved version of ConvertFrom-String cmdlet that provides a flag to enable display of the top-ranked program synthesized by FlashExtract. While better than not having such a flag, an MVP still complained: "If you can understand this, you're a better person than I am."

9.2. Program Navigation

A typical PBE interface might pick the top-ranked program and use it to automate the user's task (as in FlashFill), or even display that program to the user. The Program Navigation user interaction model [19] builds over this typical UI experience in two ways: First, it leverages a paraphrasing engine to paraphrase any DSL program into natural language such as English. This enables non-programmers to have a better chance of understanding the synthesized programs. It also makes it easy for programmers to understand the meaning of the synthesized programs, specially since they might not be familiar with the underlying DSL.

Second, it allows users to navigate between all programs synthesized by the underlying search engine (as opposed to displaying only the top-ranked program) and to pick one that is intended. The number of such programs can be huge (several powers of 10); however, they share common sub-expressions and are described succinctly using VSAs. The Program Navigation UI model leverages this sharing to create a navigational interface that allows the user to select from different ranked choices for various parts of top-ranked programs.

9.3. Conversational Clarification

Conversational Clarification is a novel complementary novel user interaction model [19] based on active learning, wherein the system asks questions to the user to resolve ambiguities in the user's specification with respect to the available test data. These questions are generated after the PBE search engine has synthesized multiple programs that are consistent with the user-provided specification. The system executes these multiple programs on the test data to identify any discrepancies in the execution and uses that as the basis for asking questions to the user. The user responses are used to refine the initial specification and the process of program synthesis is repeated.

[6] `http://spreadsheetpage.com/index.php/blog/C10/`

Conversational Clarification is a *proactive interface* that asks clarifying questions of the user for specification refinement. In contrast, Program Navigation is a *reactive interface* that assists the user to explicitly correct any mistake made by the underlying PBE engine while picking a DSL program that matches the user's under-specified specification.

10. Discussion

We now discuss some FAQs related to PBE.

Comparison with Machine Learning: It is interesting to compare PBE and Machine learning (ML). While both involve example-based training and prediction on new unseen data, they differ significantly in how they operate and hence they have complementary strengths. For simple repetitive tasks that can be automated using small scripts, PBE is a better technology for the following reasons:

- PBE generates human readable and editable programs (unlike black box models produced by ML).
- PBE requires very few examples (unlike ML, which typically requires large amount of training data).
- PBE generates scripts that are supposed to work with perfect precision on any valid new input (unlike ML, which aims to generate models with high, but not necessarily 100%, precision).

For fuzzy tasks, such as speech translation or image recognition, ML based technologies are the only alternative (since no small script that can be generated by a PBE technology can automate such sophisticated and fuzzy tasks).

However, machine learning can be used as an integral part of a PBE methodology in various ways. First, it can be used to learn ranking functions for selecting among those programs (generated by the search algorithm) that are consistent with the user specification (§8.3). Second, it can even be used to drive the search process based on learned biases about which production rules in a DSL are more likely to yield an intended program given the user specification [21]. Third, it can be used in allowing DSLs with probabilistic semantics, for handling noise in input data or for modeling semantic background knowledge such as names or non-schematized datetimes [29].

Limitations of a given PBE tool: A user might wonder whether or not a given task can be automated by a PBE tool for that task domain. If the underlying synthesizer is complete (for instance, one that is based on witness functions that are precise), then the following characterization can be used to answer this question: If the task that the user intends to perform can be described by a program in the underlying DSL, then the synthesizer will discover some program that can perform the intended task after the user has provided a sufficient set of representative examples. (A good ranking scheme will though ensure that most common tasks can be automated with very few examples.) On the other hand, if the task that the user intends to perform cannot be described by a program in the underlying DSL, then the system will eventually fail to find a program after the user has provided sufficient set of examples (which cannot be described by any program in the underlying DSL).

Deductive Synthesis vs. Inductive Synthesis: *Deductive synthesis* refers to the process of synthesis using deductive *techniques*. Deductive synthesis has been traditionally used for synthesis from formal logical specifications. *Inductive synthesis* refers to the process of synthesis from inductive or example-based *specifications*. Inductive synthesis has been traditionally performed using various techniques including enumerative search and constraint-based techniques. Deductive synthesis and inductive synthesis are not two ends of a spectrum, but rather they belong to two different dimensions [7], one that relates to the synthesis technique, and the other that relates to the form of the user specification. The methodology described in this article combines both deductive synthesis and inductive synthesis in that it performs synthesis using deductive techniques from inductive specifications. Recent work by Osera et.al. [23] and Feser et.al. [6] also belong to this category.

11. Conclusion

The programming languages research community has traditionally catered to the needs of professional programmers in the continuously evolving technical industry. However, there is a new opportunity that knocks our doors. The recent IT revolution has resulted in the masses having access to personal computing devices. More than 99% of these computer users are non-programmers and are today limited to being passive consumers of the software that is made available to them. The PBE paradigm can empower these users to more effectively leverage computers for their daily tasks by allowing them to create small scripts using examples.

A killer application, which has appropriately driven research in PBE forward in the last few years, is that of data wrangling. Data is the new oil. While the digital revolution resulted in massive digitization of human generated data, the past few years have seen an explosive growth in machine generated data, thanks to cloud computing and IoT. Data scientists are estimated to spend around 80% time wrangling data before it is brought into a form where they can apply machine learning techniques to draw appropriate insights from. PBE has enabled faster and easier data wrangling.

Building useful end-to-end PBE systems requires **cross-disciplinary inspiration**. Logical reasoning techniques of the kind developed in the *Formal methods* community can drive development of efficient search algorithms and heuristics (§7). Language design principles from the community of *Programming Languages* can inspire creation of useful DSLs (§5). *Machine learning* techniques are useful in ranking (§8). The field of *Human-computer interaction* plays a significant role in designing user interaction models (§9).

There are two big opportunities going forward. One is to enable **by-example interaction for any relevant feature in any software** such as filtering by example, grouping by example, sorting by example, formatting by example, etc. Such a software can then proudly model itself with the logo "PBE inside". In order to enable this, we need to empower expert developers (who can write DSLs, but are not necessarily expert in developing search techniques) to be able to build such functionalities without further help from researchers. The search methodology described in this article, which packages various search strategies inside a modular framework and is parameterizable by DSLs, shall facilitate such a future of industrialization of development of PBE technologies.

The other big opportunity is to define the **next generation of programming experience** that goes beyond composing syntactically correct sequence of instructions to realize a particular task. This new paradigm shall facilitate interactive programming using *multi-modal natural input* from the user. While this article has focused on techniques for handling example-based specification, it turns out that natural language is a better fit for certain class of tasks such as spreadsheet queries [11] and smartphone scripts [16]. More generally, the new paradigm shall allow expressing intent using combination of various means [25] such as examples, demonstrations, natural language, keywords, and sketching of a partial program [30]. Such a new paradigm shall also require designing new forms of debugging support involving active learning.

Acknowledgments

Thanks to the Marktoberdorf Summer School organizers for inviting me to present lectures on *programming by examples*, which inspired this draft. These lectures were also accompanied by a wonderful hands-on tutorial, given by Alex Polozov, illustrating use of FlashMeta framework [24] to create an individual PBE synthesizer. Additional thanks to Alex for providing valuable feedback on this draft. Thanks to all my collaborators whose passion and leadership has made this vision possible.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[2] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, 2015.

[3] N. Bjørner. Taking satisfiability to next level with Z3. In *IJCAR*, 2012.

[4] S. Cheema, S. Buchanan, S. Gulwani, and J. J. L. Jr. A practical framework for constructing structured drawings. In *IUI*, 2014.

[5] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[6] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[7] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[9] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

[10] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[11] S. Gulwani and M. Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.

[12] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.

[13] T. Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.

[14] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, 2000.

[15] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *PLDI*, 2014.

[16] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[17] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *PLDI*, 2015.

[18] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[19] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *UIST*, 2015.

[20] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, 2013.

[21] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[22] T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2), 1982.

[23] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[24] O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *OOPSLA*, 2015.

[25] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, 2015.

[26] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[27] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.

[28] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV*, 2015.

[29] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *POPL*, 2016.

[30] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[31] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[32] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, 2013.

[33] K. Yessenov, S. Tulsiani, A. K. Menon, R. C. Miller, S. Gulwani, B. W. Lampson, and A. Kalai. A colorful approach to text processing by example. In *UIST*, 2013.