Programming by Examples: Applications, Algorithms, and Ambiguity Resolution

Sumit Gulwani

Microsoft Corporation, Redmond, WA, USA
sumitg@microsoft.com

Abstract. 99% of computer end users do not know programming, and struggle with repetitive tasks. Programming by Examples (PBE) can revolutionize this landscape by enabling users to synthesize intended programs from example based specifications. A key technical challenge in PBE is to search for programs that are consistent with the examples provided by the user. Our efficient search methodology is based on two key ideas: (i) Restriction of the search space to an appropriate domain-specific language that offers balanced expressivity and readability (ii) A divide-and-conquer based deductive search paradigm that inductively reduces the problem of synthesizing a program of a certain kind that satisfies a given specification into sub-problems that refer to sub-programs or sub-specifications.Another challenge in PBE is to resolve the ambiguity in the example based specification. We will discuss two complementary approaches: (a) machine learning based ranking techniques that can pick an intended program from among those that satisfy the specification, and (b) active-learning based user interaction models. The above concepts will be illustrated using FlashFill, FlashExtract, and FlashRelate—PBE technologies for data manipulation domains. These technologies, which have been released inside various Microsoft products, are useful for data scientists who spend 80% of their time wrangling with data. The Microsoft PROSE SDK allows easy construction of such technologies.

1 Introduction

Program Synthesis [4] is the task of synthesizing a program that satisfies a given specification. The traditional view of program synthesis has been to synthesize programs from logical specifications that relate the inputs and outputs of the program. Programming by Examples (PBE) [6] is a sub-field of program synthesis, where the specification consists of input-output examples, or more generally, output properties over given input states. PBE has emerged as a favorable paradigm for two reasons: (i) the example-based specification in PBE makes it more tractable than general program synthesis. (ii) Example-based specifications are much easier for the users to provide in many scenarios.

2 Applications

PBE has been applied to various domains [3, 15], and some recent applications include parsing [14], refactoring [17], and query construction [20]. However, the killer application of PBE today is in the broad space of *data wrangling*, which refers to the tedious process of converting data from one form to another. The data wrangling pipelines includes tasks related to extraction, transformation, and formatting.

Extraction: A first step in a data wrangling pipeline is often that of ingesting or extracting tabular data from semi-structured formats such as text/log files, web pages, and XML/JSON 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 relevant data. The FlashExtract technology allows extracting structured (tabular or hierarchical) data out of semi-structured documents from examples [12]. 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. The FlashExtract technology ships 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 test. The FlashExtract technology also ships in Azure OMS (Operations Management Suite), where it enables extracting *custom fields* from log files.

Transformation: The *Flash Fill* feature, released in Excel 2013 and beyond, is a PBE technology for automating syntactic string transformations of the kind such as converting "FirstName LastName" into "LastName, FirstName" [5]. PBE can also facilitate more sophisticated string transformations that require lookup into other tables [21]. PBE is also a very natural fit for automating transformations of other data types such as numbers [22] and dates [24].

Formatting: Another useful application of PBE is in the space of formatting data tables. This can be useful in converting semi-structured tables found commonly in spreadsheets into proper relational tables [2], or for re-pivoting the underlying hierarchical data that has been locked into a two-dimensional tabular format [10]. PBE can also be useful in automating repetitive formatting in a powerpoint slide deck such as converting all red colored text into green, or switching the direction of all horizontal arrows [19].

3 Algorithms

Our methodology for designing and developing PBE algorithms involves three key insights: domain-specific languages, deductive search, and a framework that provides rich reusable machinery.

Domain-specific Language: A key idea in program synthesis is to restrict the search space to an underlying domain-specific language (DSL) [7, 1]. The DSL should be expressive enough to represent a wide variety of tasks in the underlying task domain, but also restricted enough to allow efficient search. We have designed many functional domain-specific languages for this purpose, each of which is 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) [6].

Deductive Search: A simple search strategy is to enumerate all programs in order of increasing size [27]. Another commonly used search strategy is to reduce the search problem to constraint solving via an appropriate reduction and then leverage off-the-shelf SAT/SMT constraint solvers [25, 26, 8]. None of these search strategies work effectively for our domains: the underlying DSLs are too big for an enumerative strategy to scale, and involve operators that are too sophisticated for existing constraint solvers to reason about.

Our synthesis algorithms employ a novel deductive search methodology [18] that is based on standard algorithmic paradigm of divide-and-conquer. The key idea is to recursively reduce the problem of synthesizing a program expression e of a certain kind and that satisfies a certain 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. In contrast to enumerative search, this search methodology is top-down—it fixes the top-part of an expression and then searches for its sub-expressions. Enumerative search is usually bottom-up—it enumerates smaller sub-expressions before enumerating larger expressions.

Framework: Developing a synthesis algorithm for a specific domain is an expensive process: The design of the algorithm requires domain-specific insights. A robust implementation requires non-trivial engineering. Furthermore any extensions or modifications to the underlying DSL are not easy.

The divide-and-conquer strategy underneath the various synthesis algorithms can be refactored out inside a framework. Furthermore, since the reduction logic depends on the logical properties of the top-level operator, these properties can be captured modularly by the framework for re-use inside synthesizers for others DSLs that use that operator. Our PROSE framework [18] builds over these ideas and has facilitated development of industrial-strength PBE implementations for various domains.

4 Ambiguity Resolution

Examples are an ambiguous form of specification; there are often many programs that are consistent with the specification provided by a user. A challenge is to

identify an intended program that has the desired behavior on the various inputs that the user cares about. Tessa Lau presented a critical discussion of PBE systems in 2009 noting that PBE systems are not yet widespread due to lack of usability and confidence in such systems [11]. We present two complementary techniques for increasing usability and confidence of a PBE system.

Ranking: Our synthesis algorithms generate the set of all/most programs in the underlying DSL that are consistent with the specification provided by the user. We rank these programs and pick the top-ranked program. Ranking is a function of both program features and data features. Program features typically capture simplicity and size of a program. Data features are over the data that is generated by the program when executed on various inputs. Weights over these features can be learned using machine learning techniques in an offline manner [23].

User Interaction models: In case the ranking does not pick an intended program, or even otherwise, we need appropriate user interaction models that can provide the equivalent of debugging experience in standard programming environments. We can allow the user to navigate between all programs synthesized by the underlying synthesizer (in an efficient manner) and to pick an intended program [16]. Another complementary technique can be to ask questions to the user as in active learning. These questions can be generated based on the differences in the results produced by executing the multiple synthesized programs on the available inputs [16].

5 Conclusion and Future Work

The programming languages research community has traditionally catered to the needs of professional programmers in the continuously evolving technical industry. The widespread access to computational devices has brought a new opportunity, that of enabling non-programmers to create small programs for automating their repetitive tasks. PBE becomes a very valuable paradigm in this setting.

It is interesting to compare PBE with Machine learning (ML) since both involve example-based training and prediction on new unseen data. PBE learns from very few examples, while ML typically requires large amount of training data. The models generated by PBE are human-readable and editable programs unlike many black-box models produced by ML. On the other hand, ML is better suited for fuzzy/noisy tasks.

There are many interesting future directions. The next generation of programming experience shall be built around *multi-modal specifications* that are natural and easy for the user to provide. While this article has focused on example-based specifications, natural language-based specifications can complement example-based specifications and might even be a better fit for various class of tasks such as spreadsheet queries [9] and smartphone scripts [13]. Furthermore, the specifications may be provided iteratively, implying the need for incremental

synthesis algorithms. Another interesting future direction is to build systems that learn user preferences based on past user interactions across different programming sessions. (For instance, the underlying ranking can be dynamically updated). This can pave the way for personalization and learning across users.

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. A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
4. S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
5. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
6. S. Gulwani. Programming by examples (and its applications in data wrangling). In J. Esparza, O. Grumberg, and S. Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.
7. S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.
8. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
9. S. Gulwani and M. Marron. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.
10. W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.
11. T. Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.
12. V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *PLDI*, 2014.
13. V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.
14. A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *PLDI*, 2015.
15. H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.
16. 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.
17. N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, 2013.
18. O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *OOPSLA*, 2015. https://microsoft.github.io/prose/.
19. M. Raza, S. Gulwani, and N. Milic-Frayling. Programming by example using least general generalizations. In *AAAI*, 2014.
20. Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.

21. R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.
22. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.
23. R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV*, 2015.
24. R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *POPL*, 2016.
25. A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.
26. S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.
27. 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.