

Accessible Programming using Program Synthesis

by

Rishabh Singh

Bachelor of Technology (Honors), Computer Science and Engineering,
Indian Institute of Technology, Kharagpur (2008)

Master of Science, Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Rishabh Singh, MMXIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part.

Author
August 28, 2014

Certified by
Armando Solar-Lezama
Associate Professor without Tenure
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by
Sumit Gulwani
Principal Researcher
Microsoft Research, Redmond
Thesis Supervisor

Accepted by
Leslie A. Kolodziej
Professor of Electrical Engineering
Chair, Department Committee on Graduate Students

ACCESSIBLE PROGRAMMING USING PROGRAM SYNTHESIS

RISHABH SINGH

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Ph.D.)
at the
Massachusetts Institute of Technology

August 2014

Dedicated to the loving memory of my father,
Mr. Amala Prasad Singh.

1960 – 2002

ABSTRACT

New computing platforms have greatly increased the demand for programmers, but learning to program remains a big challenge. Program synthesis techniques have the potential to revolutionize programming by making it more accessible. In this dissertation, I present three systems, AutoProf, FlashFill, and Storyboard Programming Tool (SPT), that work towards making programming more accessible to a large class of people, namely students and end-users. The AutoProf (Automated Program Feedback) system provides automated feedback to students on introductory programming assignments. It has been successfully piloted on thousands of student submissions from an edX course and is currently being integrated on the MITx and edX platforms. The FlashFill system helps spreadsheet end-users perform semantic string transformations, number transformations, and table lookup transformations using few input-output examples. A part of the FlashFill system is shipping in Microsoft Excel 2013 and was quoted as one of the top features in Excel by many press articles. Finally, the Storyboard Programming Tool helps students write data structure manipulations using visual examples similar to the ones used in textbooks and classrooms. It has been used to synthesize many textbook manipulations over linked list, binary search trees, and graphs.

These systems are enabled by new Program Synthesis techniques. Unlike traditional program synthesis approaches where the primary goal was to derive provably correct programs from a complete specification, these synthesis techniques are designed around natural specification mechanisms and intuitive interaction models. Each system relies on a different synthesis technique, but the techniques can be structured and understood in terms of four major components:

- **Specification Mechanism:** The way users specify the functional behavior of their intended tasks to the system. Our synthesis techniques support more natural and intuitive forms of specification such as input-output examples, reference implementation, intermediate states etc.
- **Hypothesis Space:** The hypothesis space defines the space of possible programs the system searches over to synthesize the desired program. The hypothesis space can be fixed or parametrized (user-defined) with additional user inputs. The fixed hypothesis spaces are defined using domain-specific languages (DSL) that exploit the domain knowledge to efficiently structure the hypothesis space, which enables the systems to represent a huge set of expressions in these languages succinctly. The parametric

hypothesis spaces are defined using intuitive user inputs that allows users to easily define and control the space of possible programs.

- **Synthesis Algorithm:** We develop new constraint-based and version-space algebra based synthesis algorithms to efficiently learn programs from a large hypothesis space that conform to the specification.
- **User Interaction Model:** Finally, the user interaction model determines how users refine their intent and provide additional insights to the system for converging to the desired task.

We describe the three systems based on the four components, and present experimental evaluation to show the effectiveness of these systems in practice. We also present related work and some future directions to build upon these techniques to make programming accessible to an even larger class of people.

Thesis Advisors: Armando Solar-Lezama (MIT CSAIL) and Sumit Gulwani (Microsoft Research, Redmond)

Thesis Committee: Rob Miller (MIT CSAIL), Sumit Gulwani (Microsoft Research, Redmond), Armando Solar-Lezama (MIT CSAIL)

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013

Rishabh Singh and Sumit Gulwani. Learning Semantic String Transformations from Examples. *PVLDB*, 5(8):740–751, 2012

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet Data Manipulation Using Examples. *Communications of the ACM Research Highlight*, 55(8):97–105, August 2012

Rishabh Singh and Sumit Gulwani. Synthesizing Number Transformations from Input-Output Examples. In *CAV*, pages 634–651, 2012

Rishabh Singh and Armando Solar-Lezama. SPT: Storyboard Programming Tool. In *CAV*, pages 738–743, 2012

Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011

ACKNOWLEDGMENTS

I would first like to thank my advisors, Armando Solar-Lezama and Sumit Gulwani. I could not have possibly dreamt to have worked on a better problem with better advisors, and I feel very fortunate to have them as my colleagues and friends. I am thankful to them for spending countless hours teaching me how to be a better researcher, a better writer, and a better presenter.

From the very beginning of my graduate student life, Armando showed interest in all of my ideas irrespective of how naive they were. I still remember during my very first meeting with him when I was discussing about speeding-up refinement in Model Checking, he gave some excellent suggestions and said “Lets solve this problem, and then work on Synthesis”. He was always available to meet with me, sometimes we met 6 times a day! During paper deadlines, we would stay up together until 7 am and keep making changes 15 minutes after the deadline. Armando taught me that everything is possible if we believe in it and focus on it with all our resources. At one of our meetings with the edX team, the AutoProf demo was taking around 2 minutes to generate feedback, which was a little too much for edX both in terms of server costs and waiting time. Armando said we can improve the backend engine, which I thought could not be optimized any further. After about two weeks, sure enough we were able to get it down to 10 seconds! Armando gave me full freedom to pursue my ideas, and gave me just enough advice so that I keep my focus on the larger goal. In addition to being an excellent advisor, he was also always there to give me great advice even for personal matters.

I first met Sumit at the Oregon Summer School in 2009, and fell in love with his passion and vision for using synthesis for helping end-users and students. I am thankful for him to take time off his very busy schedule for having phone/email discussions with me even after I got back to MIT, which laid the foundations of our work in synthesis. I then got a chance to spend three memorable summers at Microsoft Research, Redmond to work on the FlashFill system. Sumit’s excitement, encouragement, and support made my work in the FlashFill system one of the most exciting and enjoyable research experiences. I thank him for treating me like a family member. I still remember how Sumit would graciously drive me for lunch, gym, and other fun places, and we would have great research discussions outside of office. Sumit taught me the importance of working on important problems and finding practical solutions that can have big impact. His passion for research is truly infectious and inspirational. It was not uncommon to see him in the office at 9 AM and 2 AM on the

same day! Whenever I have questions regarding research or personal matters, I can always count on him to help me and give me the best advice.

I would like to thank Rob for being on my thesis committee. I recently started collaborating with Rob and it has been a refreshing experience to get to know how great HCI research is performed. Even during this short time, I have learnt immensely from him. Rob taught me that building a good enough system is not good enough, we as researchers should always strive to build the best possible system and then make it better. The discussions with Rob on the AutoProf system were instrumental in shaping the feedback that we are currently deploying on the MITx platform. Another thing I really liked while working with Rob was the way he manages his research group – the interesting group meeting setup, pairing students for research, and writing papers in google docs with Dropbox. I had decided if I become a Professor, I would try to manage my group as Rob does.

I would also like to thank my undergraduate advisor Andrey Rybalchenko for exposing me to the beautiful world of formal methods research. His enthusiasm and passion for research, and the motto “Believe to Achieve” has been inspiring me every day.

The Lambda-1337 family: I would like to say a big thank you to the lambda-1337 group: Aleks, Eunsuk, Jean, Joe, Kuat, Sachi, and Sasa. We all came to MIT together roughly six years back and they deserve as much credit as anyone for helping me finish my thesis. They always supported me, made me feel like a family, and never let me miss home at any moment. I looked forward to coming to office every day to spend time with them. I would like to thank Aleks for all the great discussions we had specially during our walk from MIT to home on every topic ranging from research to politics. I thank Eunsuk for helping me ease into the MIT culture, for all kinds of great discussions, and supporting me in all my decisions. Jean for being my fiercest competitor, pushing me to become better every day, and always supporting me as a great friend. Joe for all the great advice about everything in life, research discussions, and being the best Shalimar buffet lunch companion. Kuat for always making me think about the other skeptic side of my research, which helped me a lot in improving my work. Sachi for all the fun activities we did from cricket to movies to competing in Age of Empires. Sasa for teaching me how to remain positive in every situation, great research discussions, and teaching me how not to drown while swimming in the deep pool. Finally, Marcie for her support, motivating me to keep getting better, and keeping me updated with the latest music. It was a great privilege to share these last six years with you guys and I hope to spend many more years together.

MIT-PL and CAP Groups: I would like to thank all the faculty members of the MIT-PL group: Adam Chlipala, Arvind, Armando

Solar-Lezama, Charles Lieserson, Daniel Jackson, Martin Rinard, Rob Miller, and Saman Amarsinghe. They always supported my research with excellent suggestions, and most importantly taught me everything about academia and how to be a good researcher through various panels and discussions. I would also like to thank my Computer-Aided Programming (CAP) group members: Alvin, Evan, Jean, Kuat, Rohit, Santiago, Xiaokang, Zenna, and Zhilei. Thanks for the great group meetings, giving great feedback on my papers and talks, and all the great discussions outside of meetings.

Collaborators: I was fortunate enough to get to collaborate with great researchers: Rajeev Alur, Randy Davis, Dimitra Giannakopoulou, Elena Glassman, Sumit Gulwani, Philip Guo, John Guttag, William Harris, Rebecca Krosnick, Aleksandar Milicevic, Rob Miller, Joseph Near, Corina Pasareanu, Derek Rayside, Martin Rinard, Dany Rouhana, Andrey Rybalchenko, Andrew Sabisch, Jeremy Scott, Rohit Singh, Armando Solar-Lezama, Emily Su, Zhilei Xu, Ben Zorn. I learnt a lot about how to do great research and improve myself as a researcher while working with each one of them.

Friends: My roommates Rahul and Saurav, who came together with me from IIT Kharagpur, made my stay very enjoyable. I would also like to thank all the friends at MIT who made my stay a lot of fun: Jackie, Oshani, Harshad, Neha, Ted, Santiago, Andrew, Mike, Charith. My friends Aws and Loris, whom I met at Microsoft Research, for always supporting me and having great discussions.

Family: Finally, I would like to thank my amazingly loving and supporting parents, my sister Richa, and my brother Rohit. I was fortunate enough to spend last three years with Rohit when he also joined MIT for his PhD. I would like to thank him specially for all his help and support, endless encouragement and motivation, and teaching me about startups. I would also like to thank Deeti for all her endless support and understanding my crazy schedule during the PhD.

Funding: I would like to thank Microsoft Research for the PhD fellowship and the NSF for the Expeditions project ExCAPE (CCF NSF 1139056) and the Human-Centered Software Synthesis grant (NSF 1116362) that funded my work, and gave me full freedom to pursue my ideas.

I dedicate this thesis to my dad, whom I wish was here to see this work. It is only because of him I am here writing this dissertation. Thanks for everything you did for me.

CONTENTS

1	INTRODUCTION	1
1.1	AutoProf	4
1.2	FlashFill	5
1.3	Storyboard Programming	7
1.4	Key Contributions	8
2	PROGRAM SYNTHESIS COMPONENTS	10
2.1	Specification Mechanism	10
2.2	Hypothesis Space	13
2.3	Synthesis Algorithm	20
2.4	User Interaction Model	22
3	AUTOPROF	24
3.1	AutoProf Feedback Example	26
3.2	Specification Mechanism	30
3.3	Hypothesis Space of Corrections	30
3.3.1	EML: Error Model Language	31
3.3.2	Rewriting Student Solution using an Error Model	35
3.4	Synthesis Algorithm	37
3.4.1	Translation of mPY programs to SKETCH	38
3.4.2	CEGISMIN: Incremental Solving for the Minimize holes	41
3.4.3	Mapping SKETCH solution to generate feedback	44
3.5	User Interaction Model	44
3.6	Implementation and Experiments	44
3.6.1	Implementation	45
3.6.2	Benchmarks	45
3.6.3	Experiments	46
3.7	Capabilities and Limitations	50
4	SEMANTIC STRING TRANSFORMATIONS IN FLASHFILL	52
4.1	Motivating Example	53
4.2	Hypothesis Space: Lookup Transformations	55
4.2.1	Lookup Transformation Language L_t	55
4.3	Synthesis Algorithm	58
4.3.1	Data Structure for Set of Expressions in L_t	58
4.3.2	Synthesis Algorithm for L_t	60
4.3.3	Ranking	64
4.4	User Interaction Model	64
4.5	Semantic Transformations	65
4.5.1	Semantic Transformation Language L_u	67
4.5.2	Synthesis Algorithm	69
4.5.3	Synthesis Algorithm for L_u	70
4.5.4	Ranking	74
4.6	Standard Data Types	74

4.7	Experimental Evaluation	77
4.8	Ranking in FlashFill	80
4.8.1	Motivating Examples	80
4.8.2	Learning the Ranking Function	83
4.8.3	Efficiently Ranking a Set of L_a Expressions	86
4.8.4	Case Study: FlashFill	93
5	STORYBOARD PROGRAMMING TOOL	100
5.1	Example Manipulations with SPT	102
5.1.1	In-place Linked List Reversal	102
5.1.2	Linked List Deletion	106
5.2	Specification Mechanism: Storyboard	107
5.3	Hypothesis Space	111
5.3.1	A Simple Pointer Language L_p	111
5.3.2	Abstract Semantics of L_p	112
5.4	Constraint-based Synthesis Algorithm	117
5.4.1	Computing Least Fixed Points	118
5.4.2	Dealing with sets of abstract shapes	119
5.4.3	Termination	119
5.5	User Interaction Model	120
5.6	Experimental Evaluation	120
6	RELATED WORK	123
6.1	Program Synthesis	123
6.2	Programming By Example and Demonstrations	125
6.3	Computer-aided Education and Grading	128
6.4	Automated Program Repair	130
6.5	Query Synthesis in Databases	131
6.6	Ranking in Program Synthesis	132
7	FUTURE WORK	135
8	CONCLUSIONS	139
	BIBLIOGRAPHY	140

LIST OF FIGURES

Figure 1	The four major components of new program synthesis techniques. 2	
Figure 2	The characterization of AutoProf, FlashFill, and SPT based on the four components of the new program synthesis approach. 3	
Figure 3	The reference implementation for computeDeriv. 4	4
Figure 4	A simplified error model. 4	
Figure 5	(a) An incorrect student attempt for computeDeriv, and (b) The corresponding feedback generated by the AutoProf system. 5	
Figure 6	A transformation that requires lookup and join operations on multiple tables, and then syntactic manipulations on the results. 6	
Figure 7	Doubly linked list deletion example in SPT. 7	7
Figure 8	A semantic transformation in Flashfill to convert dates from one format to another specified using a concrete input-output example. 10	
Figure 9	A concrete input-output example of linked list of length 4 specifying the desired behavior of the list reverse manipulation in SPT. 11	
Figure 10	An abstract input-output example of linked list of an unbounded length specifying the desired behavior of the list reverse manipulation in SPT. The Figure also shows the inductive definition of the abstract list node (shown using ellipsis). 12	
Figure 11	A scenario consisting of an intermediate state for binary search tree insert manipulation in SPT. 12	
Figure 12	The reference implementation for the evaluatePoly problem specified by an instructor in AutoProf. The problem asks students to evaluate a polynomial on a value x , where the polynomial coefficients are represented using a Python list <code>poly</code> . 13	
Figure 13	The loop skeleton for the list reverse implementation in SPT. 15	
Figure 14	Syntax for a general abstract language L_a for a version-space algebra based Programming-By-Example system. 16	

Figure 15	A data structure D_a for succinctly representing a large set of L_a expressions. 17	
Figure 16	Semantics of the data structure D_a used to succinctly represent a large number of L_a expressions. 18	
Figure 17	Set-based sharing of position pair (substring) expressions in FlashFill. 19	
Figure 18	Path-based sharing of concatenate expressions for the transformation $Rob \rightarrow Mr.$ Rob in FlashFill. 20	
Figure 19	The general synthesis algorithm for learning a program in the hypothesis space that conforms to a given set of input-output examples. 21	
Figure 20	Solving an exists forall quantified constraint using the Synthesis and Verification phases of the CounterExample Guided Inductive Synthesis Algorithm (CEGIS). 21	
Figure 21	The reference implementation for <code>computeDeriv</code> . 26	
Figure 22	Three very different student submissions ((a), (b), and (c)) for the <code>computeDeriv</code> problem and the corresponding feedback generated by our tool ((d), (e), and (f)) for each one of them using the same reference implementation. 27	
Figure 23	The architecture of our AutoProf system. 30	
Figure 24	The syntax of a simple Python-like language <code>mPy</code> . 31	
Figure 25	The syntax of language \widetilde{mPy} for succinct representation of a large number of <code>mPy</code> programs. 32	
Figure 26	The $\llbracket \cdot \rrbracket$ function that translates an <code>mPy</code> program to a weighted set of <code>mPy</code> programs. 33	
Figure 27	The error model \mathcal{E} for the <code>computeDeriv</code> problem. The default choices that do not change the original expression are added by default by the translation function. 34	
Figure 28	The $\mathcal{T}_{\mathcal{E}_1}$ method for error model \mathcal{E}_1 . 35	
Figure 29	Application of $\mathcal{T}_{\mathcal{E}_1}$ (abbreviated \mathcal{T}) on expression $(x[i] < y[j])$. 36	
Figure 30	The resulting \widetilde{mPy} program after applying correction rules to program in Figure 22(a). 37	
Figure 31	The <code>MultiType</code> struct for encoding Python types. 38	
Figure 32	The <code>addMT</code> function for adding two <code>MultiType</code> a and b. 39	
Figure 33	The translation rules for converting \widetilde{mPy} set-exprs to corresponding <code>SKETCH</code> function bodies. 41	

Figure 34	The sketch generated for $\widetilde{\text{mPy}}$ program in Figure 30. 42
Figure 35	An example of big conceptual error for a student's attempt for (a) <code>evalPoly</code> and (b) <code>hangman2-str</code> problems. 47
Figure 36	The number of incorrect student submissions that require different number of corrections (in log scale). 48
Figure 37	The number of incorrect student submissions corrected by addition of correction rules to the error models. 49
Figure 38	The performance of <code>computeDeriv</code> error model on other benchmark problems. 50
Figure 39	A transformation that requires syntactic manipulations on multiple lookup results. 54
Figure 40	The syntax of lookup transformation language L_t . 55
Figure 41	A lookup transformation that requires joining two tables. 56
Figure 42	The syntax of the data structure D_t for succinctly representing a set of expressions from language L_t . 58
Figure 43	The semantics of the data structure D_t for succinctly representing a set of expressions from language L_t . 58
Figure 44	The reachability graph of nodes in Ex. 4.2.3. 59
Figure 45	The <code>GenerateStr_t</code> procedure for generating the set of all expressions (of depth at most k) in language L_t that are consistent with a given input-output example. 60
Figure 46	The <code>Intersect_t</code> procedure for intersecting sets of expressions from language L_t . The <code>Intersect_t</code> procedure returns \emptyset in all other cases. 61
Figure 47	The general synthesis algorithm for version-space algebra based synthesis algorithm. 62
Figure 48	A lookup transformation that requires concatenating input strings before performing selection from a table. 68
Figure 49	A nested syntactic and lookup transformation. It requires concatenating results of multiple lookup transformations, each of which involves a selection operation that indexes on some substring of the input. 68
Figure 50	A partial Dag representation of the set of expressions in Example 4.5.3. 71

Figure 51	User interface of our programming-by-example Excel add-in. (a) and (b) are the screenshots before and after clicking the Apply button. 75	
Figure 52	Formatting dates using examples. 76	
Figure 53	(a) Number of expressions consistent with given i-o examples and (b) Size of data structure (in terms of size of grammar derivation, where each terminal symbol contributes a unit size) to represent all consistent expressions. 78	
Figure 54	(a) Running time (in seconds) to learn the program and (b) Size of the data structure D_u before and after performing Intersect_u . 79	
Figure 55	Adding Mr. title to the input names. 81	
Figure 56	Initials from input Firstnames. 82	
Figure 57	Extracting city names from addresses. 82	
Figure 58	Algorithm for finding the highest ranked expression amongst a set of fixed arity expressions. 89	
Figure 59	Algorithm for finding the highest ranked expression amongst a set of associative expressions represented as a DAG. 92	
Figure 60	The DAG data structure for representing all programs induced programs in Example 4.8.1. 93	
Figure 61	The set of features for ranking position pair expression $\text{SubStr}(\sigma_i, \{\tilde{p}_j\}_j, \{\tilde{p}_k\}_k)$, where $\tilde{p}_j = \text{pos}(r_1^l, r_2^l, c^l)$, $\tilde{p}_k = \text{pos}(r_1^r, r_2^r, c^r)$. The table also shows the abstraction function, abstract dimension, project dimension, and projection set for features. 95	
Figure 62	The features for ranking expressions on DAG edges, where s denotes the string obtained from executing the DAG edge expression and i denotes the input substring that matches with s . 96	
Figure 63	The set of associative features for ranking a set of $\text{Concatenate}(a_1, \dots, a_n)$ expressions together with their binary operator and numerical feature. 97	
Figure 64	Comparison of LearnRank with the Baseline scheme for a random 30-70 partition. 98	
Figure 65	The running times for FlashFill without ranking (NoRanking) and with LearnRank. 99	
Figure 66	Doubly linked list deletion example in SPT. 100	
Figure 67	Graphical description of linked-list reverse 102	
Figure 68	The storyboard scenarios for in-place linked list reversal. 103	

Figure 69	Unfold and fold predicate definitions for mid summary node. 104
Figure 70	Control flow sketch for list reversal. Each number corresponds to an unknown block of code. 105
Figure 71	(a) The synthesized implementation of list reverse, and (b) cleaned up version of (a) with false conditionals removed. 105
Figure 72	Textual description of linked list deletion storyboard shown in Figure 73. 106
Figure 73	The storyboard consisting of four visual scenarios describing the input and output state descriptions for linked list deletion. 106
Figure 74	Two possible unfold definitions for summary node front. 107
Figure 75	(a) The loop skeleton and (b) the synthesized implementation for linked list deletion. 108
Figure 76	An abstract list representing infinite concrete lists 108
Figure 77	State configuration for a singly linked list 109
Figure 78	The syntax for a simple pointer language L_p . 111
Figure 79	Abstract semantics of L_p statements. 112
Figure 80	Abstract semantics of L_p conditionals. 113
Figure 81	Unfold and fold operations for different data structure manipulations 115
Figure 82	unfold in 3-valued shape analysis 116
Figure 83	Red-black tree fixInvariant storyboard. 122
Figure 84	Learning transformations on unstructured, non-uniform, and ambiguous data type strings. 135

LIST OF TABLES

Table 1	The percentage of student attempts corrected and the time taken for correction for the benchmark problems. 48
Table 2	The average number of examples to learn test tasks for 10 runs of different training partitions. 98
Table 3	Experimental results for case studies 122

INTRODUCTION

The unprecedented ubiquity of computational devices has resulted in a big increase in the demand for programmers to build systems and services on top of these devices. Unfortunately, programming these devices has not become much easier. One still needs to write a program in a step-by-step manner detailing every single step of the execution in an arcane programming language, which makes it challenging to meet the increasing programmer demand. The goal of this dissertation is to build systems to democratize the programming experience to a large class of users, who are not programmers, but have access to these devices and services, and want to program them to perform useful custom tasks.

We can try to achieve the goal of democratizing programming using two approaches. First, we can build systems that allow for natural and intuitive interaction mechanisms for users to specify their intended tasks so that even non-programmers can perform programming tasks. Second, we can build systems to help users learn traditional programming. In this dissertation, I present three systems that take both of these approaches to make programming more accessible to a large class of users, namely students and end-users.

The AutoProf (Automated Program Feedback) system [112] provides automated feedback to students on introductory programming assignments and has been successfully piloted on thousands of student submissions from an online edX course “Introduction to Computer Science and Programming (6.00x)”. It is currently being integrated on the MITx and edX platforms. The FlashFill system [49, 56] helps spreadsheet end-users perform semantic string transformations [108], number transformations [107], and table lookup transformations [108] using few input-output examples. A part of the FlashFill system was shipped in Microsoft Excel 2013 and was quoted as one of the top features in Excel this year by many press articles [1, 2, 3, 6, 7]. Finally, the Storyboard Programming Tool (SPT) [109, 111] helps students write data structure manipulations using visual examples similar to the ones used in textbooks and classrooms. It has been used to synthesize many textbook manipulations over linked list, binary search trees, and graphs.

These systems from the domains of automated grading, programming by example, and visual programming are enabled by new program synthesis techniques. Program synthesis has been an intriguing area of research from a long time, whose ultimate goal has been to automatically synthesize programs from high-level specifications [82].

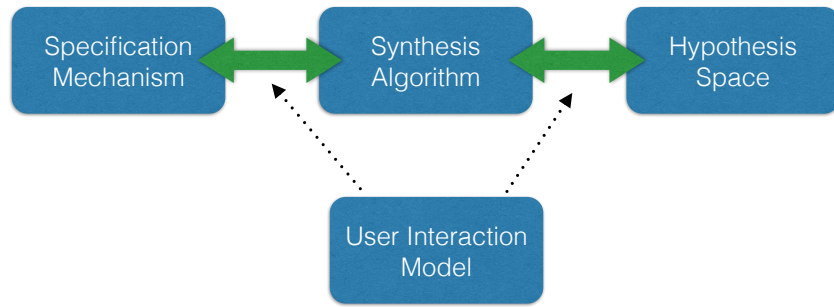


Figure 1: The four major components of new program synthesis techniques.

Traditionally, it has been used to synthesize small but provably correct complex programs from a *complete* specification. One major pitfall for these traditional program synthesis approaches has been the expectation of a complete specification. Unfortunately, often times writing a complete specification is at least as hard as writing the program in first place. The interaction mechanism in some of the traditional interactive synthesis systems [23, 116] also requires deep expertise in deduction and theorem proving. The systems we need for helping non-programmers do not conform to the view of traditional synthesis approaches. Instead, these systems demand new synthesis approaches to support different forms of more intuitive specifications and interaction models. These systems embrace the fact that writing complete specifications is difficult and allow for specification mechanisms that are easier and more natural for non-programmers.

These three systems depend on different program synthesis techniques, but the techniques are structured around four major components (Figure 1). These components provide a systematic way to characterize the systems based on new program synthesis techniques.

- **Specification Mechanism:** Specification mechanism is the way users specify the functional behavior of their intended tasks to the system. Writing a complete specification of the intended task is challenging even for programmers and we can not expect non-programmers to provide such specifications. Therefore, our synthesis technique supports simpler and more intuitive specification mechanisms such as input-output examples (abstract and concrete), reference implementations, and intermediate states. These specifications are inherently ambiguous which makes the synthesis techniques more challenging.
- **Hypothesis Space:** Traditional program synthesis approaches have aimed at synthesizing programs in Turing-complete languages, which makes the synthesis process more complex and less scalable. The hypothesis space in new synthesis techniques can either be fixed or parameterized (user-defined). For fixed hypothesis spaces, we identify *efficient* subsets of these Turing-

Characterization of AutoProf, FlashFill, and SPT based on New Program Synthesis Components			
	AutoProf	FlashFill	SPT
Specification Mechanism	Concrete I/O Examples	Reference Implementation	Abstract I/O Examples, Concrete I/O Examples
Hypothesis Space	Student Submission + Error Model (in mPy)	DSL for Semantic Transformations	DSL for Pointer Assignments + Loop Skeleton
Synthesis Algorithm	Constraint-based Synthesis	Version-space Algebra based Synthesis	Abstract Interpretation + Constraint-based Synthesis
User Interaction Model	Correction Rules, Feedback Levels	Additional examples, Ambiguity Highlighting	Intermediate States, Partial Program

Figure 2: The characterization of AutoProf, FlashFill, and SPT based on the four components of the new program synthesis approach.

complete languages to create Domain-specific languages (DSL) that exploit the domain knowledge for efficiently structuring the hypothesis space. The key idea in designing these domain-specific languages is to make them expressive enough to be able to encode majority of the intended tasks, but at the same time keep them concise enough for learning. For parametric hypothesis spaces, the synthesis techniques allow for additional user inputs to enable users to easily define and control the search-space for all possible programs.

- **Synthesis Algorithm:** The synthesis algorithm efficiently learns a program in the hypothesis space that conform to the specification. Since specification mechanisms such as input-output examples are inherently incomplete and ambiguous, and our hypothesis space is expressive, there are typically a large number of programs in the hypothesis space that conform to the specification. The synthesis techniques use constraints and novel data-structures to represent this large number of programs (10^{15}) succinctly, and use corresponding constraint-based algorithms and divide-and-conquer algorithms for efficiently learning expressions (succinctly represented in these representations) that conform to the specification.
- **User Interaction Model:** Finally, a major component of usability of these systems is the user interaction model. The user interaction model determines how the system guides the users in cases where the provided specification is ambiguous or incomplete, and how a user then refines the intent and provides additional information for the system to converge to the intended task.

The four components for AutoProf, FlashFill, and SPT systems are shown in Figure 2. We now briefly describe the three systems together with their characterization based on these four components.

```

def computeDeriv_list_int(poly_list_int):
    result = []
    for i in range(len(poly_list_int)):
        result += [i * poly_list_int[i]]
    if len(poly_list_int) == 1:
        return result      # return [0]
    else:
        return result[1:]  # remove the leading 0

```

Figure 3: The reference implementation for computeDeriv.

$$\begin{aligned}
 \text{range}(a_1, a_2) &\rightarrow \text{range}(a_1, a_2 + 1) \\
 a_0 \geq a_1 &\rightarrow a_0 > a_1
 \end{aligned}$$

Figure 4: A simplified error model.

1.1 AUTOProf

There has been a big interest recently to teach programming to hundreds of thousands of people worldwide through MOOCs. One major challenge, however, in these courses is how to replicate the personalized feedback provided in traditional classrooms in the online setting.

The AutoProf [113] system for Python provides feedback for introductory programming exercises to students by telling them exactly what is wrong with their solution and how to correct it. The specification mechanism for the tool is a reference implementation that specifies the desired functional behavior of a student solution. For example, the reference implementation for the computeDeriv problem is shown in Figure 3. The computeDeriv problem is taken from an introductory programming course on edX that asks students to write a python function that computes the derivative of a polynomial whose coefficients are represented using a Python list.

The hypothesis space in AutoProf is defined by the combination of a student program and the error model. An error model is a set of rewrite rules that captures common mistakes that students make while solving a given problem. For example, a simple error model for the computeDeriv problem is shown in Figure 4. The error model for this example captures two very specific mistakes: incrementing the second argument of the range function, and changing the greater than equal operator to greater than. In general an error model would have many more correction rules, but these two rules suffices to find mistakes in the incorrect student submission shown in Figure 5.

Given the reference implementation and the error model, the synthesis algorithm symbolically searches over the space of all possible

rewrites to a student solution and finds a corrected solution, which is functionally equivalent to the reference implementation and which requires minimum number of rewrites. This set of rewrites induces a large space of corrections (10^{15}), and we need to check each one of them for functional equivalence. We encode this large solution space using constraints and use an iterative minimization algorithm to efficiently solve them. The user interaction model of AutoProf for teachers is to allow them to incrementally add rewrite rules in the error model for providing feedback on student mistakes that are not captured by the current error model.

We have evaluated AutoProf on thousands of student submissions from the Introduction to Programming course (6.00x) offered on edX in Fall 2012. These exercises cover a range of Python data types including integers, lists, strings, and dictionaries, and programming idioms such as comparisons, iteration, and recursion. The tool was able to generate feedback for 64% of all incorrect student submissions and took about 9 seconds on average per submission. The system is currently being integrated on the MITx and edX platforms.

<pre> 1 def computeDeriv(poly): 2 length = int(len(poly)-1) 3 i = length 4 deriv = range(1,length) 5 if len(poly) == 1: 6 deriv = [0] 7 else: 8 while i >= 0: 9 new = poly[i] * i 10 i -= 1 11 deriv[i] = new 12 return deriv </pre>	<p>The program requires 2 changes:</p> <ul style="list-style-type: none"> • In the expression range(1, length) in line 4, increment length by 1. • In the comparison expression (i >= 0) in line 8, change operator >= to >.
--	---

Figure 5: (a) An incorrect student attempt for `computeDeriv`, and (b) The corresponding feedback generated by the AutoProf system.

1.2 FLASHFILL

FlashFill is an interactive programming-by-example system for spreadsheets. Spreadsheets have millions of users with diverse backgrounds from traders to graphic designers. They are not professional programmers but often need to create one-off applications to support business needs. These end-users routinely struggle with transformations on data over strings, numbers, and tables, but can easily specify their intent using examples [47]. The FlashFill project was started by Sumit Gulwani at Microsoft Research with an observation that

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
S30	Stroller	30%
B56	Bib	45%
D32	Diapers	35%
W98	Wipes	40%
A46	Aspirator	30%
...

CostRec		
Id	Date	Price
S30	12/2010	\$145.67
S30	11/2010	\$142.38
B56	12/2010	\$3.56
D32	1/2011	\$21.45
W98	4/2009	\$5.12
A46	2/2010	\$2.56
...

Figure 6: A transformation that requires lookup and join operations on multiple tables, and then syntactic manipulations on the results.

programming syntactic string transformations using examples can be reduced to synthesizing programs in a domain-specific language of strings [49]. We developed a more general programming-by-example (PBE) methodology to allow end-users to automate repetitive tasks [56] and used it to extend the FlashFill system to support *semantic* string transformations, namely table lookup transformations [108] and number transformations [107]. Another important aspect of our work has been to make FlashFill usable in practice by developing a novel methodology to use machine learning for learning a ranking function for PBE systems [106]. The ranking function is used to efficiently disambiguate between the huge number of learnt expressions so that users only need to provide minimal number of input-output examples.

The specification mechanism for FlashFill is one or more tables of data in a spreadsheet together with the desired output strings for a few of the table rows. For example, consider a post taken from an Excel help forum (Figure. 6). An Excel user wanted to compute the selling price of an item (Output) from its name (Input v_1) and selling date (Input v_2) using the MarkupRec and CostRec tables. 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. The user provides the desired intent by providing outputs for the first two rows, and the system then generates the output (shown in bold) for other rows by learning the desired transformation consisting of table lookups and syntactic string transformations.

The hypothesis space for semantic transformations is defined by new domain-specific languages of lookup transformations and number transformations. These languages are then combined with the syntactic string transformation language to obtain an expressive combined language that can express tasks such as the one shown in Figure. 6. A key idea in the design of these domain-specific languages is that they are composed of specific forms of expressions, namely fixed-arity expressions and associative expressions, that allow for sharing a large number of expressions succinctly. This allows us to efficiently structure the hypothesis space of possible expressions.

The synthesis algorithm learns a large number of expressions (10^{30}) in these languages that conform to the examples in polynomial time, where the expressions are represented succinctly using novel data structures. These expressions are then ranked using a ranking function (learnt using machine learning) and the top-ranked expression is then run on the table rows to compute the corresponding output strings. The user interaction model of FlashFill is to allow users to provide additional input-output examples in case the system generates some undesired outputs. It can also highlight ambiguous input cells for which there are multiple highly-ranked outputs.

A part of the FlashFill system (string transformations with ranking) was shipped in Microsoft Excel 2013. The initial response for the FlashFill feature has been quite encouraging. It was quoted as one of the top features in Excel 2013 by many press articles [2] and there are currently more than 100 user-created YouTube videos about using it.

1.3 STORYBOARD PROGRAMMING

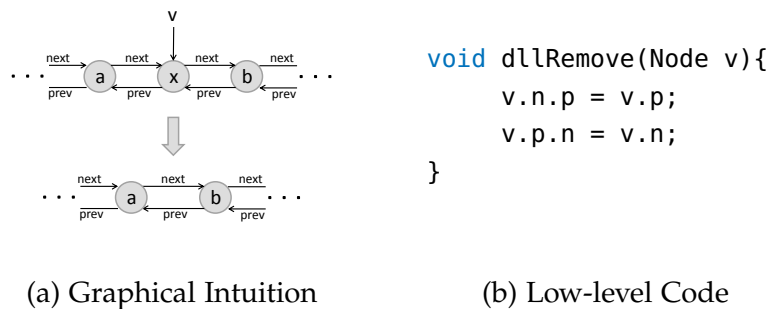


Figure 7: Doubly linked list deletion example in SPT.

Students learning to program find it challenging when there is a large gap between the abstractions at which algorithms are taught and explained in classrooms and the abstractions at which they are required to be programmed. One domain where this gap is rather large is data-structure manipulations, which are typically described using high-level visual diagrams. Their translation to low-level pointer manipulating code, however, is non-intuitive, tedious, and error-prone.

The Storyboard Programming Tool (SPT) [109, 111] is a new programming interface that aims to bridge this gap between the high-level visual insights and the corresponding low-level code. SPT combines programming-by-example with constraint-based synthesis. The specification mechanism in SPT lets a user specify the high-level insights in several different forms such as concrete input-output examples, abstract input-output examples, and intermediate states. For example, a user can specify the deletion manipulation in a doubly linked list by providing an abstract input-output example as shown in Figure 7(a). The hypothesis space of SPT is a domain-specific language of pointer assignments together with conditionals and loops (with bounded pointer dereferences). This language is parameterized by a loop skeleton, which helps to further constrain the hypothesis space as well as prevent the synthesis system from synthesizing solutions with arbitrary structure.

The synthesis algorithm in SPT combines constraints with abstract-interpretation based shape analysis to encode both the synthesis and verification problems as a constraint satisfaction problem whose solution defines the low-level pointer code. The synthesized low-level code for doubly linked list deletion is shown in Figure 7(b). Finally, SPT’s user interaction model allows users to provide additional input-output examples, intermediate data structure states, and additional information regarding the expected size of the synthesized code in case the default bounds are not enough. We have used SPT to successfully synthesize several traditional textbook data structure manipulations such as insertion, deletion, reversal, rotation, and traversal over linked lists, binary search trees, and graphs.

1.4 KEY CONTRIBUTIONS

This dissertation makes the following key contributions:

- **Generality of the Program Synthesis approach:** We show problems from very different domains of automated feedback generation for introductory programming assignments (AutoProf), spreadsheet data transformation using examples (FlashFill), and visual programming of data structure manipulations (SPT) can be encoded and efficiently solved using the new program synthesis approaches.
- **Multimodal Specification Mechanisms:** The presented systems embrace the fact that it is hard for non-programmers to provide complete functional specification of their desired intent, and allow for many different forms of specifications that are natural and intuitive. Some examples of such specifications include input-output examples (abstract and concrete), reference implementations, intermediate states, partial traces, and asser-

tions. The SPT system has now also been extended to allow for ink-based and voice-based specifications [100].

- **Fixed and Parametric Hypothesis Space:** The hypothesis space in these program synthesis approaches can either be fixed or parametric. For fixed hypothesis spaces, domain-specific languages are used to define the hypothesis space. These DSLs are expressive enough to capture most tasks in the domain, but at the same time are concise enough for efficient learning. The DSLs are composed of two classes of expressions, namely fixed-arity expressions and associative expressions, that allow for efficient structuring of the hypothesis space. The parametrized hypothesis space enables users to define and control the space of possible programs using intuitive inputs.
- **New Synthesis Algorithms:** We develop new constraint-based synthesis algorithms that combine abstract interpretation with the CEGIS algorithm (SPT), and allow for minimization constraints (CEGISMIN in AutoProf). We also develop version-space algebra based synthesis algorithms in FlashFill that exploit the sharing amongst DSL sub-expressions to efficiently learn a huge set of programs in several DSLs.
- **Machine Learning to build Synthesizers:** We present a general ranking technique to predict a correct program from a large set of programs that are induced from an incomplete and ambiguous specification. Our ranking technique uses gradient descent for learning the ranking function with the goal of optimizing a loss function that ranks any correct program over all incorrect programs. We use machine learning to build synthesizers instead of using it directly to learn the programs. Machine learning is ideal to learn relatively less complex functions from a large amount of data, whereas Program synthesis is more suitable to learn more complex structures such as programs from very few examples. Our approach presents a novel complementary combination of machine learning techniques with the program synthesis techniques.
- **Encouraging initial impact of the Systems:** Some of the presented systems have been practically deployed in the real-world and are already having some initial impact. A part of the FlashFill system was shipped in Microsoft Excel 2013 and was quoted as one of the top features in Excel 2013 by many press articles. The AutoProf system was successfully piloted over thousands of students submissions from an introductory programming course on edX, and is currently being integrated on the MITx and edX platforms.

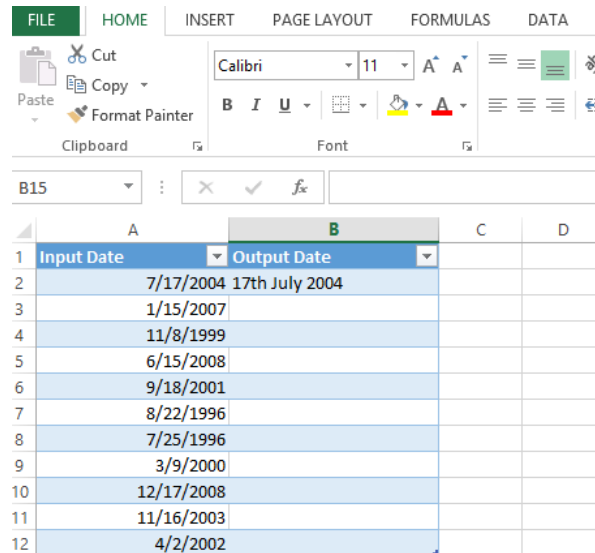
PROGRAM SYNTHESIS COMPONENTS

This chapter describes in more detail the four components of the new program synthesis techniques. We formalize the specification constraint obtained from different specification mechanisms, the hypothesis space, and the synthesis algorithm that learns the intended program in the hypothesis space that is consistent with the specification constraint. We then describe some user interaction models that users can use to refine their specification to provide additional information to the synthesis system.

Let P be the unknown program, H be hypothesis space of all possible programs, I be a set of inputs, and ϕ be the functional specification. The goal of the synthesis algorithm is to learn the unknown program by solving the synthesis constraint $\Theta \equiv \exists P \in H \forall \text{inp} \in I : \phi(P, \text{inp})$, i.e. the functional specification holds for all inputs inp from the set I .

2.1 SPECIFICATION MECHANISM

The new synthesis approaches support simpler and intuitive specification mechanisms such as input-output examples (concrete and abstract), reference implementations, and intermediate states (partial traces). We briefly describe each one of these mechanisms and the corresponding synthesis constraints that are obtained from them.



	A	B	C	D
1	Input Date	Output Date		
2	7/17/2004	17th July 2004		
3	1/15/2007			
4	11/8/1999			
5	6/15/2008			
6	9/18/2001			
7	8/22/1996			
8	7/25/1996			
9	3/9/2000			
10	12/17/2008			
11	11/16/2003			
12	4/2/2002			

Figure 8: A semantic transformation in Flashfill to convert dates from one format to another specified using a concrete input-output example.

Concrete Input-Output Examples: The concrete input-output examples are one of the simplest and commonly used specification mechanism for programming-by-example techniques. The concrete examples describe exactly how the input and output states should look like without abstracting away any details. For example, consider the example of a semantic string transformation in Flashfill shown in Figure 8. The date transformation from one date format to another is specified using an example with an input string “7/17/2004” and the corresponding output string “17th July 2004”. Another example of using concrete input-output examples to specify data structure manipulations in SPT is shown in Figure 9. The desired behavior of the list reverse manipulation is specified on a linked list of length 4 (with four concrete locations a, b, c, and d). In general, a user can provide a set of concrete input-output examples $C_I = \{(i_1, o_1), (i_2, o_2), \dots, (i_k, o_k)\}$. The functional specification in the synthesis constraint is $\phi_C(P, \text{inp}) \equiv ((\text{inp} == i_1) \implies (P(i_1) == o_1)) \wedge \dots \wedge ((\text{inp} == i_k) \implies (P(i_k) == o_k))$.

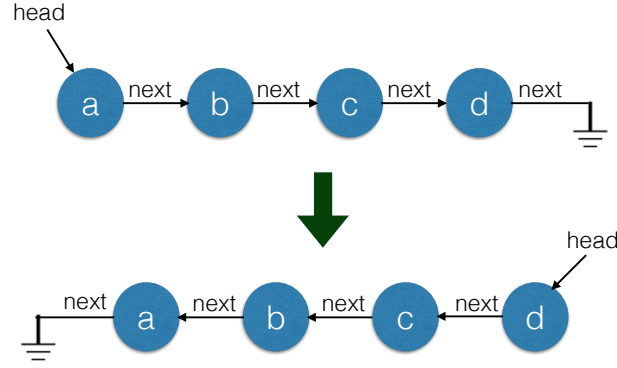


Figure 9: A concrete input-output example of linked list of length 4 specifying the desired behavior of the list reverse manipulation in SPT.

Abstract Input-Output Examples: The abstract input-output examples allow users to specify the behavior of the desired program on potentially infinite number of concrete input-output examples. For example, the abstract input-output example shown in Figure 10 shows the behavior of the list reverse manipulation on an abstract list (of unbounded length). The input list consists of two concrete nodes a and b, and an abstract node specified by the ellipsis. The ellipsis denotes a list segment of unbounded length and is defined inductively as shown on the right in the Figure. The abstract examples let users specify constraints on the desired program without enumerating all (potentially infinite) concrete examples. In general, a user can provide a set of abstract input-output examples $A_I = \{(i_1^\#, o_1^\#), (i_2^\#, o_2^\#), \dots, (i_k^\#, o_k^\#)\}$. The functional specification in the generated synthesis constraint is $\phi_C(P^\#, \text{inp}) \equiv ((\text{inp} == i_1^\#) \implies (P^\#(i_1^\#) \subseteq o_1^\#)) \wedge \dots \wedge ((\text{inp} == i_k^\#) \implies (P^\#(i_k^\#) \subseteq o_k^\#))$, where the set of inputs I corresponds to the set of input-output examples, i.e. $I =$

$\{i_1^\#, i_2^\#, \dots, i_k^\#\}$, and $P^\#$ denotes an abstraction of program P constructed using an appropriate abstract domain.

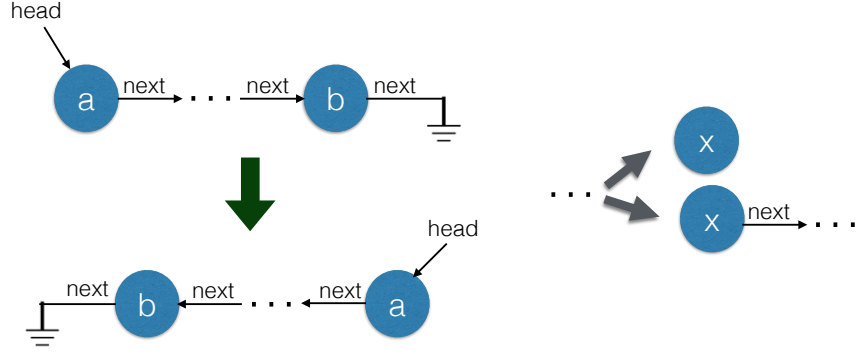


Figure 10: An abstract input-output example of linked list of an unbounded length specifying the desired behavior of the list reverse manipulation in SPT. The Figure also shows the inductive definition of the abstract list node (shown using ellipsis).

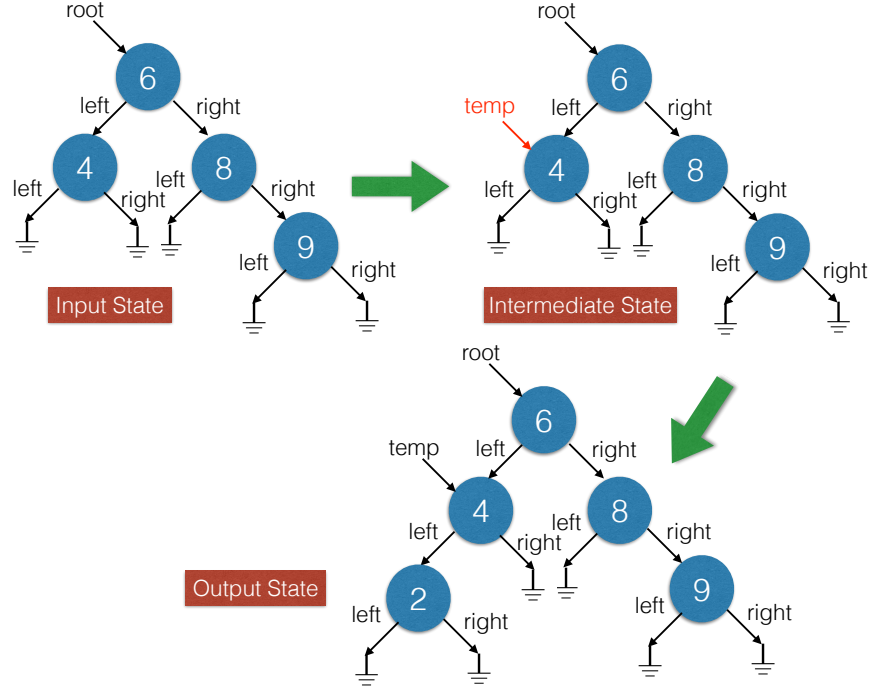


Figure 11: A scenario consisting of an intermediate state for binary search tree insert manipulation in SPT.

Intermediate States (Partial Traces): The intermediate states provide additional constraints to the synthesis system about the desired program and help scale the synthesis algorithm to larger programs. For example, the scenario for binary search tree insertion in Figure 11 shows an intermediate state that corresponds to the case where the insertion location of the new node has been found (at the end of the bst search loop). The function specification predicate is $\phi_{\text{int}}(P, i) \equiv$

$P_l(i) = j$, where l denotes the location of the intermediate state j and i denotes the input state.

Reference Implementation: A user can also specify the desired functional behavior of the unknown program P using a reference implementation R . For example, an instructor can specify the desired functionality of the `evaluatePoly` problem in the AutoProf system as shown in Figure 12. The `evaluatePoly` problem asks students to write a Python program to compute the value of a polynomial on a given value x , where the coefficients of the polynomial are represented using a Python list `poly`. The functional specification in the synthesis constraint obtained is $\phi_R(P, \text{inp}) \equiv P(\text{inp}) == R(\text{inp}) \wedge \text{Asserts}(P, \text{inp})$, where the $\text{Asserts}(P, \text{inp})$ predicate denotes whether any assertion was violated on the input `inp`.

```
def evaluatePoly(poly, x):
    result = 0
    for i in range(len(poly)):
        result += poly[i] * (x ** i)
    return result
```

Figure 12: The reference implementation for the `evaluatePoly` problem specified by an instructor in AutoProf. The problem asks students to evaluate a polynomial on a value x , where the polynomial coefficients are represented using a Python list `poly`.

2.2 HYPOTHESIS SPACE

The hypothesis space defines the space of all possible programs that is searched over by the synthesis algorithm to find the desired program. The hypothesis space can be fixed apriori for a system or it can be parameterized (user-defined) with additional user inputs. At one extreme end are the fixed hypothesis spaces, where the key idea is to define the space using a Domain-specific Language (DSL) that exploits the domain knowledge for efficiently structuring the space of possible programs. The DSLs have two properties: (i) they are expressive enough so that most desired tasks in the domain can be expressed, and (ii) they are concise enough to enable efficient learning of expressions that conform to different forms of specifications. At the other extreme end are the user-defined hypothesis spaces, which are completely defined using user inputs. We present few intuitive mechanisms for users to provide the inputs to easily define and control the hypothesis space.

The hypothesis space for the AutoProf system is completely user-defined. An instructor provides an error model consisting of a set of rewrite rules that corresponds to common mistakes that students are making on a given problem. The error model defines the space

of possible corrections searched over by the AutoProf system on a student solution. The instructor can further enrich the error model by adding new rewrite rules that expands the hypothesis space of possible corrections.

The hypothesis space for FlashFill is fixed and is defined using a new domain-specific language for semantic transformations L_u , which is obtained by combining table lookup transformation language L_t and syntactic string transformation language L_s [49]. The language L_u is expressive enough to encode transformations over strings that involve performing syntactic transformations over the outputs of lookups and joins on its constituent substrings. The restriction on the columns in the conditional predicates of the lookup transformation language L_t is that they should together constitute a candidate key of the table. A key restriction in the string language L_s is that the concatenation of substrings can only happen at the outermost expression level.

The hypothesis space for SPT is defined using a combination of the previous two approaches of defining hypothesis spaces. A student provides a loop skeleton input to describe structural constraints over the desired program, but the system imposes constraints on the kinds of statements and conditionals with which the skeleton can be filled using a domain-specific language. The DSL for SPT is a simple pointer manipulation language with loops and conditionals. The three key restrictions in the language are (i) no memory allocation of new nodes is allowed, (ii) there is at most one pointer dereference per expression, and (iii) there are only top-level (singly nested) conditional statements allowed. The third restriction precludes the occurrence of an `if` statement inside another `if` statement. The pointer language is expressive enough to express most data structure manipulations over linked lists and binary search trees, but at the same time efficiently learnable because of these restrictions.

We now describe few mechanisms with which a user can easily provide additional inputs for parametric hypothesis spaces.

Loop Skeleton: A user can specify the skeleton of the desired program using a loop skeleton S . This helps the synthesis system to constrain the hypothesis space of possible programs and helps in preventing the synthesizer from synthesizing solutions with arbitrary structure. The loop skeleton for the list reverse program in SPT is shown in Figure 13. It specifies that the list reverse implementation should have a single while loop, with a set of unknown states before the loop, in the loop body, and after the loop (denoted using the `??` construct). The construct `**` denotes an unknown comparison predicate over pointer dereferences. The constraint generated from the loop skeleton are $H \equiv S(c)$, where c denotes a parametric completion of the loop skeleton S .

Error Models: The hypothesis space of corrections searched over by the AutoProf system is parameterized by an error model \mathcal{E} . An

```

Node llReverse(Node head){
    ??
    while(**){
        ??
    }
    ??
}

```

Figure 13: The loop skeleton for the list reverse implementation in SPT.

error model is defined using a set of rewrite rules that corresponds to potential corrections for common mistakes that students are making on a given problem. For example, a simple error model below captures corrections that involve modifying the return value and the range iteration values.

$$\begin{aligned}
 \text{return } a &\rightarrow \text{return } [0] \\
 \text{range}(a_1, a_2) &\rightarrow \text{range}(a_1 + 1, a_2) \\
 a_0 == a_1 &\rightarrow \text{False}
 \end{aligned}$$

The correction rule $\text{return } a \rightarrow \text{return } [0]$ states that a return statement in a student program can be optionally replaced with a return statement that returns the list $[0]$. The hypothesis space generated from the error model is given by $H \equiv \mathcal{T}_\varepsilon(S)$, where S denotes a student submission and \mathcal{T}_ε denotes the transformation function that rewrites S using the error model ε .

Succinct Representation of Hypothesis Space

Since the specification mechanisms we consider in the new synthesis approaches are often incomplete and ambiguous, there are typically a huge number of programs in the hypothesis space that conform to the specification. For supporting efficient incremental learning, we need a mechanism to represent this large set of programs succinctly. We use two main approaches to succinctly represent this large number of consistent programs: (i) Constraint-based representation and (ii) Version-space algebra based representation.

The hypothesis space for AutoProf is represented using expressions from a language $\widetilde{\text{MPY}}$. The $\widetilde{\text{MPY}}$ language consists of set-expressions that can encode a large number of MPY programs succinctly, where MPY is a simple Python-like imperative language that encodes an efficiently learnable subset of Python. An example $\widetilde{\text{MPY}}$ expression is $\{x, y\} \{+, -, *, \} \{1, 2\}$ that succinctly represents a set of 32 different MPY expressions $\{x + 1, x + 2, x - 1, \dots, y/2\}$. The $\widetilde{\text{MPY}}$ expressions are in turn encoded using constraints.

The parametric hypothesis space of SPT is also represented succinctly using a constraints-based representation. The constraints succinctly encode the non-deterministic semantics of the abstract program traces as well as multimodal specifications.

On the other hand, the hypothesis space of a set of L_t expressions in FlashFill are succinctly represented using a data structure based on version-space algebra. This version-space algebra based representation exploits the domain knowledge of semantic string transformations to efficiently structure the hypothesis space by sharing common subexpressions at multiple different levels.

We now describe an abstract language L_a that captures two major kinds of expressions that allow for succinct representation in version-space algebra used in PBE systems, namely *fixed arity* expressions and *associative* expressions. We then present the syntax and semantics of the version-space algebra based data structure that can succinctly represent a large number of expressions in this language.

$$\begin{aligned} \text{Expr } e &:= v \mid c \mid e_f \mid e_h \\ \text{Fixed Arity Expr } e_f &:= f(e_1, \dots, e_n) \\ \text{Associative Expr } e_h &:= h(e_1, \dots, e_k) \end{aligned}$$

Figure 14: Syntax for a general abstract language L_a for a version-space algebra based Programming-By-Example system.

An Abstract Language L_a for Version-space Algebra based PBE Systems

An abstract language L_a that captures the major kinds of expression sharing in domain-specific languages of several version-space algebra based PBE systems [49, 108, 107, 58, 56] is shown in Figure 14. The top-level expression e in L_a can either be a constant c , a variable v , a fixed arity expression e_f , or an associative expression e_h . A fixed arity expression e_f applies a function f to a fixed number of arguments to compute $f(e_1, \dots, e_n)$. An associative expression e_h applies an associative operator h to an unbounded number of arguments (e_1, \dots, e_k) to compute $h(e_1, h(e_2, h(e_3, \dots, e_k) \dots))$, denoted as $h(e_1, \dots, e_k)$ for notational convenience.

Definition 2.2.1 (Fixed Arity Expression). Let f be any constructor for n independent expressions ($n \geq 1$). We use the notation $f(e_1, \dots, e_n)$ to denote a fixed arity expression with n arguments.

Example 2.2.1. The position pair expression in the FlashFill language $\text{SubStr}(v_i, p_1, p_2)$ is an example of fixed arity expression that represents the left and right position logic expressions p_1 and p_2 independently. The Boolean expression predicate $(C_1 = e_t \wedge \dots \wedge C_k = e_t)$

for a candidate key of size k in the lookup transformation language L_t is another fixed-arity expression of arity k with independent values for the column predicates. The decimal and exponential number formatting expressions $\text{Dec}(u, \eta_1, f)$ and $\text{Exp}(u, \eta_1, f, \eta_2)$ in the number transformation language [107] are also examples of fixed arity expressions with independent values for integer (η_1), fraction (f), and exponent (η_2) formatting.

Definition 2.2.2 (Associative Expression). Let h be a binary associative constructor for independent expressions. We use the simplified notation $h(e_1, \dots, e_k)$ to denote the expanded associative expression $h(e_1, h(e_2, h(e_3, \dots, h(e_{k-1}, e_k) \dots)))$ for any $k \geq 1$ (where $h(e)$ simply denotes e).

Example 2.2.2. The $\text{Concatenate}(f_1, \dots, f_n)$ expression in FlashFill is an example of an associative expression with Concatenate as the binary associative constructor. The top-level select expression $e_t := \text{Select}(C, T, C_i = e_t)$ in the lookup transformation language L_t and the associative program $\text{Assoc}(F, s_0, s_1)$ in the table layout transformation language [58] are also instances of an associative expression.

Associative expressions involve applying an associative operator with input and output type T to an unbounded sequence of expressions of type T . The associative expressions differ from the fixed arity expressions in two ways: (i) they have unbounded arity, and (ii) their input and output types are restricted to be the same.

$$\begin{aligned} \text{Union Expr } s &:= \{c_i, v_j, \dots, s_f, s_h\} \\ \text{Join Expr } s_f &:= f(s_1, \dots, s_n) \\ \text{DAG Expr } s_h &:= \text{Dag}(\tilde{\eta}, \eta^s, \eta^t, W) \\ &\quad \text{where } W : (\eta_1, \eta_2) \rightarrow s, |\tilde{\eta}| = k + 1 \end{aligned}$$

Figure 15: A data structure D_a for succinctly representing a large set of L_a expressions.

Data Structure D_a for Representing a Set of L_a Expressions

The data structure D_a to succinctly represent a large number of L_a expressions is shown in Figure 15. The Union Expression s represents a set of top-level expressions as an explicit set since there is no sharing amongst the expressions at the top level. The Join Expression s_f represents a set of fixed arity expressions by maintaining independent sets for its arguments e_1, \dots, e_n . The DAG expression s_h represents a set of associative expressions using a DAG Dag , where the edges in the graph correspond to a set of expressions s and each path from the

$$\begin{aligned}
\llbracket c \rrbracket_\sigma &:= c \\
\llbracket v \rrbracket_\sigma &:= \sigma(v) \\
\llbracket s \rrbracket_\sigma &:= \{e_j \mid e_j \in \llbracket e_i \rrbracket_\sigma, e_i \in s\} \\
\llbracket s_f \rrbracket_\sigma &:= \{f(e_1, \dots, e_n) \mid e_i \in \llbracket s_i \rrbracket_\sigma\} \\
\llbracket s_h \rrbracket_\sigma &:= \{h(e_1, \dots, e_k) \mid (\eta_1, \eta_2, \dots, \eta_{k+1}) \in \tilde{\eta}, \\
&\quad \eta_1 = \eta^s, \eta_{k+1} = \eta^t, e_i \in \llbracket W(\eta_i, \eta_{i+1}) \rrbracket_\sigma\}
\end{aligned}$$

Figure 16: Semantics of the data structure D_a used to succinctly represent a large number of L_a expressions.

start node η^s to the end node η^t denotes an associative expression. The semantics of the data structure is shown in Figure 16.

JOIN EXPRESSIONS (SET-BASED SHARING) There can often be a huge number of fixed-arity expressions that are consistent with a given example(s). Consider the input-output example pair (u, v) . Suppose v_1, v_2, v_3 are values such that $v = f(v_1, v_2, v_3)$. Suppose E_1, E_2 , and E_3 are sets of expressions that are respectively consistent with the input-output pairs (u, v_1) , (u, v_2) , and (u, v_3) . Then, $f(e_1, e_2, e_3)$ is consistent with (u, v) for any $e_1 \in E_1$, $e_2 \in E_2$, and $e_3 \in E_3$. The number of such expressions is $|E_1| \times |E_2| \times |E_3|$. However, these can be succinctly represented using the data-structure $f(E_1, E_2, E_3)$, which denotes the set of expressions $\{f(e_1, e_2, e_3) \mid e_1 \in E_1, e_2 \in E_2, e_3 \in E_3\}$, using space that is proportional to $|E_1| + |E_2| + |E_3|$.

Example 2.2.3. The data structure of FlashFill for position pair expressions $\text{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}_k\}_k)$ represents the set of left and right position logic expressions $\{\tilde{p}_j\}_j$ and $\{\tilde{p}_k\}_k$ independently. Each position logic expression in turn is represented as a tuple (r_1, r_2, c) . The set-based sharing of position pair expressions for the transformation $\$145.67 \rightarrow 145.67$ is shown in Figure 17. The left and right position logic expressions are maintained as independent sets. The generalized Boolean conditions in the select expression $\text{Select}(C, T, B)$ of the lookup transformation language L_t also exhibit set-based sharing. A generalized conditional $\tilde{b} \in B$ is represented as a conjunction of generalized predicates \tilde{p}_i , where each \tilde{p}_i is an equality comparison of some column of a candidate key with a set of string (or node) choices $\tilde{b} = \bigwedge_{i=1}^k (C_i = \{s, \eta\})$. The data structure for representing a set of decimal and exponential number formatting expressions in the number transformation language $\text{Dec}(u, \tilde{\eta}_1, \tilde{f})$ and $\text{Exp}(u, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2)$ represents integer formats $(\tilde{\eta}_1)$, fractional formats (\tilde{f}) , and exponent formats $(\tilde{\eta}_2)$ as independent sets.

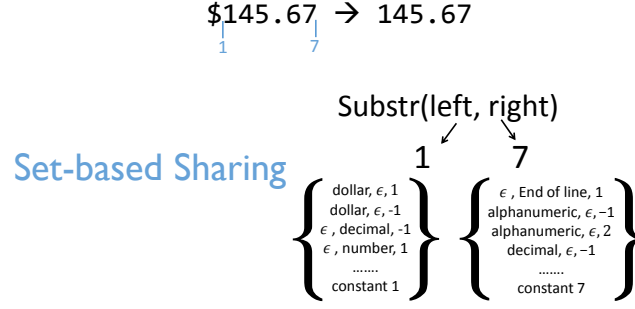


Figure 17: Set-based sharing of position pair (substring) expressions in FlashFill.

DAG EXPRESSIONS (PATH-BASED SHARING) There can often be a large number of associative expressions that are consistent with a given example(s). Consider the input-output example pair (u, v) . Suppose v_1, \dots, v_n be n values such that $v = h(v_1, \dots, v_n)$ and $e_{i,j}$ be an expression that evaluates to the value $v_{i,j} \equiv h(v_i, \dots, v_j)$ on input u ($1 \leq i < j \leq n$). Let $\sigma = [\sigma_0, \dots, \sigma_m]$ be a subsequence of $[0, \dots, n]$ such that $\sigma_0 = 0$ and $\sigma_m = n$ and e_σ be the expression $h(e'_1, \dots, e'_m)$, where $e'_i = e_{\sigma_{i-1}, \sigma_i}$. Note that the number of such subsequences σ is exponential in n , and for any subsequence σ , e_σ evaluates to $v_{1,n}$.

Such an exponential sized set of associative expressions can be represented succinctly as a DAG whose nodes correspond to $0, \dots, n$ and an edge between two nodes i and j corresponds to the value $v_{i,j}$ and is labeled with $e_{i,j}$. A path in the DAG from source node 0 to sink node n is some subsequence $[\sigma_1, \dots, \sigma_m]$ of $[0, \dots, n]$ where $\sigma_1 = 0$ and $\sigma_m = n$, and it represents the expression $F(e'_1, \dots, e'_m) = v$, where $e'_i = e_{\sigma_{i-1}, \sigma_i}$. The DAG data structure `dag` of FlashFill and the graph for generalized expression nodes for representing select expressions uses such path-based sharing for succinctly representing exponential number of expressions.

For example, the DAG representation of an associative expression (concatenate expression) for the transformation $\text{Rob} \rightarrow \text{Mr. Rob}$ is shown in Figure 18. For each index in the output string Mr. Rob , there exists a corresponding node in the DAG. An edge (i, j) between nodes i and j corresponds to an expression γ_{ij} in the DSL that generates the substring of the output string between indices i and j . For example in the Figure, the edge γ_1 between nodes 0 and 1 corresponds to a position pair expression that can generate the substring R . Any path in this DAG from the start node 0 to the end node 7 is a valid concatenate expression to generate the complete output string. There can potentially be an exponential number of paths from the start node to the end node, but they are represented succinctly in polynomial space using a DAG based representation.

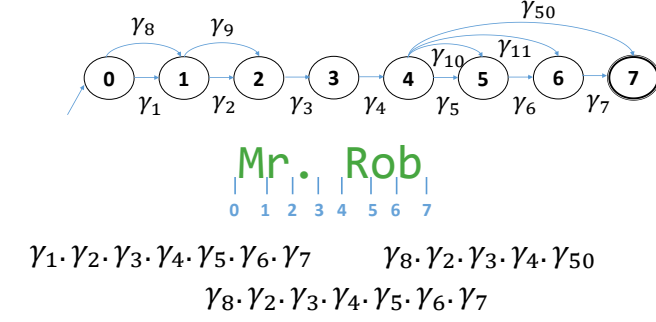


Figure 18: Path-based sharing of concatenate expressions for the transformation $\text{Rob} \rightarrow \text{Mr. Rob}$ in FlashFill.

2.3 SYNTHESIS ALGORITHM

The synthesis algorithm learns a program in the hypothesis space that conforms to the given specification. We first present a general synthesis algorithm that learns the set of all programs that conform to each individual input-output example, and intersects them to return the synthesized program that conforms to all examples. We then instantiate the components of the general synthesis algorithm with different constraint-based and version-space algebra based techniques to obtain the two corresponding instantiations: (i) Constraint-based synthesis algorithm, and (ii) Version-space algebra based synthesis algorithm. The synthesis algorithm in AutoProf is built on top of the constraint-based CEGIS (CounterExample Guided Inductive Synthesis) algorithm [117, 120] by adding support for minimality constraints. The synthesis algorithm for semantic transformations in FlashFill is based on version-space algebra, whereas the synthesis algorithm for SPT combines abstract interpretation [27, 28] with the CEGIS algorithm.

The general synthesis algorithm Synthesize is shown in Figure 19. The algorithm takes as input a set of pairs of input-output example states $\{(\sigma_1, s_1), \dots, (\sigma_n, s_n)\}$. It first learns the set of all programs P that are consistent with the first input-output example (σ_1, s_1) using the LearnProg procedure. It then iterates over the remaining examples to learn the set of programs for each input-output example, and intersects them to compute the common set of conforming programs using the Intersect procedure. The algorithm maintains a loop invariant that in the k^{th} loop iteration, the set of programs P consists of all the programs that are consistent with the first k input-output examples. Finally, the algorithm uses the Choose function to select a program from the set P to return as the synthesized program.

We obtain different synthesis algorithms based on different instantiations of the components of the general synthesis algorithm. The four components of the general synthesis algorithm are: 1) Selection of input-output examples, 2) LearnProg, 3) Intersect, and 4) Choose.

```

Synthesize( $((\sigma_1, s_1), \dots, (\sigma_n, s_n))$ )
   $P := \text{LearnProg}(\sigma_1, s_1);$ 
  for  $i = 2$  to  $n$ :
     $P' := \text{LearnProg}(\sigma_i, s_i);$ 
     $P := \text{Intersect}(P, P');$ 
  return Choose( $P$ );

```

Figure 19: The general synthesis algorithm for learning a program in the hypothesis space that conforms to a given set of input-output examples.

For the constraint-based synthesis algorithm, the input-output examples are selected using the counter-examples generated by a Verifier, the LearnProg procedure encodes the execution semantics of the input-output example as constraints, the Intersect procedure is simply a conjunction of constraints, whereas the Choose function is a non-deterministic function that randomly selects any program from the solution set of the final constraint P . For the version-space algebra based synthesis algorithm, the input-output examples are provided by the user, the LearnProg procedure is implemented as an explicit divide-and-conquer search to learn programs in a custom data structure, the Intersect procedure intersects the two data structures, and finally the Choose function is implemented using a ranking function that returns the highest ranked program from the set of all conforming programs (represented succinctly using the data structure).

We now present a brief description of the CEGIS algorithm to solve the synthesis constraint and how it selects the input-output examples.

Counter-example Guided Inductive Synthesis (CEGIS) Algorithm

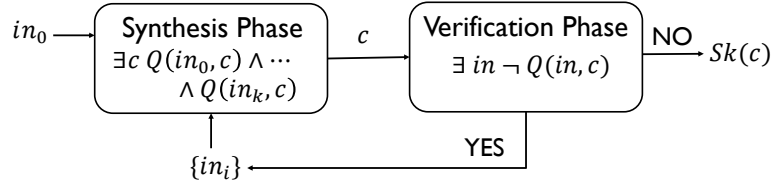


Figure 20: Solving an exists forall quantified constraint using the Synthesis and Verification phases of the CounterExample Guided Inductive Synthesis Algorithm (CEGIS).

Often times the hypothesis space of programs is parameterized using a vector of variables c , whose values define the unknown program $P \equiv Sk(c)$. Let $Q(inp, c) \equiv \Phi(P, inp)$ so that our synthesis constraint now becomes $\exists c. \forall inp. Q(inp, c)$. For solving such doubly quantified constraints, CEGIS solves an inductive synthesis problem of the form $\exists c. Q(inp_0, c) \wedge Q(inp_1, c) \cdots \wedge Q(inp_k, c)$, where $\{inp_0 \cdots, inp_k\}$ is a small set of representative inputs. If the equation above is unsatisfiable, the original equation will be unsatisfiable as well. If the equation

provides a solution, we can verify that solution by solving the following equation $\exists \text{inp } \neg Q(\text{inp}, c)$. The algorithm, shown in Figure 20, consists of two phases: synthesis phase and verification phase. The algorithm first starts with a random assignment of inputs inp_0 and solves for the constraint $\exists c Q(\text{inp}_0, c)$. If no solution exists, then it reports that the sketch can not be synthesized. Otherwise, it passes on the solution c to the verification phase to check if the solution works for all inputs using the constraint $\exists \text{inp } \neg Q(\text{inp}, c)$. If the verifier can not find a counterexample input, then the sketch $\text{Sk}(c)$ is returned as the desired solution. Otherwise, the verifier finds an input inp_1 which is then added to the synthesis phase. The synthesis phase now solves for the constraint $\exists c Q(\text{inp}_0, c) \wedge Q(\text{inp}_1, c)$. This loop between the synthesis and verification phases continues until either the synthesis or the verification constraint becomes unsatisfiable. The algorithm returns “no solution” when the synthesis constraint becomes unsatisfiable whereas it returns the sketch solution as the desired solution when the verification constraint becomes unsatisfiable.

2.4 USER INTERACTION MODEL

The specification mechanisms in these synthesis approaches are inherently ambiguous and incomplete, therefore we need some mechanism from the systems to help users refine their specification for achieving their intended goal. In this section, we describe some general mechanisms that the users can use to refine their intent in cases of under-constrained and over-constrained specification.

Incomplete (Under-constrained) Specification

Distinguishing Input The input-output examples provided by the users are often under-constrained such that there exists multiple different programs that are consistent with the specification. In such cases, the system generates a new distinguishing input inp [65] and two programs P_1 and P_2 , such that both P_1 and P_2 conform to the set of provided examples but generate different outputs on inp , i.e. $P_1(\text{inp}) \neq P_2(\text{inp})$. The system then queries the user to either select $P_1(\text{inp})$ or $P_2(\text{inp})$ as the desired output or provide some other output for the input inp . With this additional input-output example, the system restarts the synthesis process. In SPT and FlashFill, we use this technique for asking users to provide additional examples.

Multiple Outputs In version-space algebra based synthesis system, an advantage of learning the set of all conforming programs is that we can run them on new inputs to check if there are multiple different outputs being generated. FlashFill highlights such cells where the learnt programs generate multiple different outputs, which a user can

inspect and either select one of those outputs as the correct output, or can provide a new output. The system can then use this input-output example as an additional example and re-learn the transformation.

Refining Error Models With hundreds of thousands of student assignments, it can become prohibitively expensive to go over each student assignment to find the correction rules for providing corresponding feedback. The AutoProf system guides the instructor to inspect only a few student submissions that could not be corrected with the current error model. After the addition of new correction rules, AutoProf re-runs the synthesis process and returns another student assignment where the correction process with the updated error model fails and can not produce the desired feedback.

Conflicting (Over-constrained) Specification

In some cases, the specification can be over-constrained such that there does not exist any implementation that conforms to the provided specification. This can happen because of two reasons: (i) either the intended transformation lies outside the hypothesis space, or (ii) the specification is inconsistent. The synthesis system can provide some guidance to users for explaining why the synthesis process failed to help them debug the specification. SPT finds the minimal set of examples that are conflicting and asks the user to check if the outputs are correct. In FlashFill, we compute clusters of examples based on the size of the intersection set of programs, and it can query the users to check for clusters of unusually small size.

Timeout of Synthesis Process

The synthesis algorithm can sometimes take too long to generate the desired programs. In such cases, a user can refine their specification and provide additional insights to make the synthesis process more scalable. In SPT, a user can provide additional intermediate states, as well as more information in the loop skeleton with partially filled statements and conditionals. In FlashFill, a user can break-down a complex transformation into a sequence of simpler transformations. In AutoProf, an instructor can provide additional information in terms of specialized correction rules, and smaller bounds on the input values, loop unrolling, and recursion inlining to reduce the search space of corrections.

There has been a lot of interest recently in making quality education more accessible to students worldwide using information technology. Several education initiatives such as EdX, Coursera, and Udacity are racing to provide online courses on various college-level subjects ranging from computer science to psychology. These courses, also called massive open online courses (MOOC), are typically taken by thousands of students worldwide, and present many interesting scalability challenges. Specifically, we consider the challenge of providing personalized feedback for programming assignments in introductory programming courses.

The two methods most commonly used by MOOCs to provide feedback on programming problems are: (i) test-case based feedback and (ii) *peer-feedback* [37]. In test-case based feedback, the student program is run on a set of test cases and the failing test cases are reported back to the student. This is also how the 6.00x course (Introduction to Computer Science and Programming) offered by MITx currently provides feedback for Python programming exercises. The feedback of failing test cases is however not ideal; especially for beginner programmers who find it difficult to map the failing test cases to errors in their code. This is reflected by the number of students who post their submissions on the discussion board to seek help from instructors and other students after struggling for hours to correct the mistakes themselves. In fact, for the classroom version of the Introduction to Programming course (6.00) taught at MIT, the teaching assistants are required to manually go through each student submission and provide feedback describing what is wrong with the submission and how to correct it. This manual feedback by teaching assistants is simply prohibitive for the number of students in the online class setting.

The second approach of peer-feedback is being suggested as a potential solution to this problem [132]. For example in 6.00x, students routinely answer each other's questions on the discussion forums. This kind of peer-feedback is helpful, but it is not without problems. For example, we observed several instances where students had to wait for hours to get any feedback, and in some cases the feedback provided was too general or incomplete, and even wrong in a few cases. Some courses have experimented with more sophisticated peer evaluation techniques [73] and there is an emerging research area that builds on recent results in crowd-powered systems [16, 80] to provide more structure and better incentives for improving the feedback quality. However, peer-feedback has some inherent limitations, such as

the time it takes to receive quality feedback and the potential for inaccuracies in feedback, especially when a majority of the students are themselves struggling to learn the material.

We present the AutoProf tool that employs an automated technique to provide feedback for introductory programming assignments. The approach leverages program synthesis technology to automatically determine minimal fixes to the student's solution that will make it match the behavior of a reference solution written by the instructor. This technology makes it possible to provide students with precise feedback about what they did wrong and how to correct their mistakes. The problem of providing automatic feedback appears to be related to the problem of automated bug fixing, but it differs from it in the following two significant respects:

- **The complete specification is known.** An important challenge in automatic debugging is that there is no way to know whether a fix is addressing the root cause of a problem, or simply masking it and potentially introducing new errors. Usually the best one can do is check a candidate fix against a test suite or a partial specification [41]. While providing feedback on the other hand, the solution to the problem is known, and it is safe to assume that the instructor already wrote a correct reference implementation for the problem.
- **Errors are predictable.** In a homework assignment, everyone is solving the same problem after having attended the same lectures, so errors tend to follow predictable patterns. This makes it possible to use a *model-based* feedback approach, where the potential fixes are guided by a model of the kinds of errors students typically make for a given problem.

These simplifying assumptions, however, introduce their own set of challenges. For example, since the complete specification is known, AutoProf now needs to reason about the equivalence of the student solution with the reference implementation. Also, in order to take advantage of the predictability of errors, AutoProf needs to be parameterized with models that describe the classes of errors. And finally, these programs can be expected to have higher density of errors than production code, so techniques which correct bugs one path at a time [69] will not work for many of these problems that require coordinated fixes in multiple places.

Our feedback generation technique handles all of these challenges. The AutoProf tool can reason about the semantic equivalence of student programs with reference implementations written in a fairly large subset of Python, so the instructor does not need to learn a new formalism to write specifications. The tool also provides an *error model* language that can be used to write an error model: a very high level description of potential corrections to errors that students might

```

def computeDeriv_list_int(poly_list_int):
    result = []
    for i in range(len(poly_list_int)):
        result += [i * poly_list_int[i]]
    if len(poly_list_int) == 1:
        return result      # return [0]
    else:
        return result[1:]  # remove the leading 0

```

Figure 21: The reference implementation for computeDeriv.

make in the solution. When the system encounters an incorrect solution by a student, it symbolically explores the space of all possible combinations of corrections allowed by the error model and finds a correct solution requiring a *minimal* set of corrections.

We have evaluated our approach on thousands of student solutions on programming problems obtained from the 6.00x submissions and discussion boards, and from the 6.00 class submissions. These problems constitute a major portion of first month of assignment problems. Our tool can successfully provide feedback on over 64% of the incorrect solutions.

3.1 AUTOPROF FEEDBACK EXAMPLE

In order to illustrate the kind of feedback generated by AutoProf, consider the problem of computing the derivative of a polynomial whose coefficients are represented as a list of integers. This problem is taken from week 3 problem set of 6.00x (PS3: Derivatives). Given the input list `poly`, the problem asks students to write the function `computeDeriv` that computes a list `poly'` such that

$$\text{poly}' = \begin{cases} [i \times \text{poly}[i] \mid 0 < i < \text{len}(\text{poly})] & \text{if } \text{len}(\text{poly}) > 1 \\ [0] & \text{if } \text{len}(\text{poly}) = 1 \end{cases}$$

For example, if the input list `poly` is `[2, -3, 1, 4]` (denoting the function $f(x) = 4x^3 + x^2 - 3x + 2$), the `computeDeriv` function should return `[-3, 2, 12]` (denoting the derivative $f'(x) = 12x^2 + 2x - 3$). The reference implementation for the `computeDeriv` function is shown in Figure 21. This problem teaches concepts of conditionals and iteration over lists. For this problem, students struggled with many low-level Python semantics issues such as the list indexing and iteration bounds. In addition, they also struggled with conceptual issues such as missing the corner case of handling lists consisting of single element (denoting constant function).

One challenge in providing feedback for student submissions is that a given problem can be solved by using many different algorithms. Figure 22 shows three very different student submissions

Three different student submissions for computeDeriv	
Student Program	AutoProf Feedback
<pre> 1 def computeDeriv(poly): 2 deriv = [] 3 zero = 0 4 if (len(poly) == 1): 5 return deriv 6 for e in range(0, len(poly)): 7 if (poly[e] == 0): 8 zero += 1 9 else: 10 deriv.append(poly[e]*e) 11 return deriv </pre> <p>(a)</p>	<p>The program requires 3 changes:</p> <ul style="list-style-type: none"> • In the return statement return deriv in line 5, replace deriv by [0]. • In the comparison expression (poly[e] == 0) in line 7, change (poly[e] == 0) to False. • In the expression range(0, len(poly)) in line 6, increment 0 by 1. <p>(d)</p>
<pre> 1 def computeDeriv(poly): 2 idx = 1 3 deriv = list([]) 4 plen = len(poly) 5 while idx <= plen: 6 coeff = poly.pop(1) 7 deriv += [coeff * idx] 8 idx = idx + 1 9 if len(poly) < 2: 10 return deriv </pre> <p>(b)</p>	<p>The program requires 1 change:</p> <ul style="list-style-type: none"> • In the function computeDeriv, add the base case at the top to return [0] for len(poly)=1. <p>(e)</p>
<pre> 1 def computeDeriv(poly): 2 length = int(len(poly)-1) 3 i = length 4 deriv = range(1, length) 5 if len(poly) == 1: 6 deriv = [0] 7 else: 8 while i >= 0: 9 new = poly[i] * i 10 i -= 1 11 deriv[i] = new 12 return deriv </pre> <p>(c)</p>	<p>The program requires 2 changes:</p> <ul style="list-style-type: none"> • In the expression range(1, length) in line 4, increment length by 1. • In the comparison expression (i >= 0) in line 8, change operator >= to !=. <p>(f)</p>

Figure 22: Three very different student submissions ((a), (b), and (c)) for the computeDeriv problem and the corresponding feedback generated by our tool ((d), (e), and (f)) for each one of them using the same reference implementation.

for the `computeDeriv` problem, together with the feedback generated by our tool for each submission. The student submission shown in Figure 22(a) is taken from the 6.00x discussion forum¹. The student posted the code in the forum seeking help and received two responses. The first response asked the student to look for the first if-block return value, and the second response said that the code should return `[0]` instead of empty list for the first if statement. There are many different ways to modify the code to return `[0]` for the case `len(poly)=1`. The student chose to change the initialization of the `deriv` variable from `[]` to the list `[0]`. The problem with this modification is that the result will now have an additional 0 in front of the output list for all input lists (which is undesirable for lists of length greater than 1). The student then posted the query again on the forum on how to remove the leading 0 from result, but unfortunately this time did not get any more response.

AutoProf generates the feedback shown in Figure 22(d) for the student program in about 40 seconds. During these 40 seconds, AutoProf searches over more than 10^7 candidate fixes and finds the fix that requires minimum number of corrections. There are three problems with the student code: first it should return `[0]` in line 5 as was suggested in the forum but was not specified how to make the change, second the if block should be removed in line 7, and third that the loop iteration should start from index 1 instead of 0 in line 6. The generated feedback consists of four pieces of information (shown in bold in the figure for emphasis):

- the location of the error denoted by the line number.
- the problematic expression in the line.
- the sub-expression which needs to be modified.
- the new modified value of the sub-expression.

The feedback generator is parameterized with a feedback-level parameter to generate feedback consisting of different combinations of the four kinds of information, depending on how much information the instructor is willing to provide to the student.

WORKFLOW In order to provide the level of feedback described above, AutoProf needs some information from the instructor. First, AutoProf needs to know what the problem is that the students are supposed to solve. The instructor provides this information by writing a reference implementation such as the one in Figure 21. Since Python is dynamically typed, the instructor also provides the types of

¹ https://www.edx.org/courses/MITx/6.00x/2012_Fall/discussion/forum/600x_ps3_q2/threads/5085f3a27d1d4225000000040

function arguments and return value. In Figure 21, the instructor specifies the type of input argument to be list of integers (`poly_list_int`) by appending the type to the name.

In addition to the reference implementation, AutoProf needs a description of the kinds of errors students might make. We have designed an error model language EML, which can describe a set of correction rules that denote the potential corrections to errors that students might make. For example, in the student attempt in Figure 22(a), we observe that corrections often involve modifying the return value and the range iteration values. We can specify this information with the following three correction rules:

$$\begin{aligned} \text{return } a &\rightarrow \text{return } [0] \\ \text{range}(a_1, a_2) &\rightarrow \text{range}(a_1 + 1, a_2) \\ a_0 == a_1 &\rightarrow \text{False} \end{aligned}$$

The correction rule $\text{return } a \rightarrow \text{return } [0]$ states that the expression of a return statement can be optionally replaced by `[0]`. The error model for this problem that we use for our experiments is shown in Figure 27, but we will use this simple error model for simplifying the presentation in this section. In later experiments, we also show how only a few tens of incorrect solutions can provide enough information to create an error model that can automatically provide feedback for thousands of incorrect solutions.

The rules define a space of candidate programs which AutoProf needs to search in order to find one that is equivalent to the reference implementation and that requires minimum number of corrections. We use constraint-based synthesis technology [117, 52, 122] to efficiently search over this large space of programs. Specifically, we use the SKETCH synthesizer that uses a SAT-based algorithm to complete program sketches (programs with holes) so that they meet a given specification. We extend the SKETCH synthesizer with support for *minimize* hole expressions whose values are computed efficiently by using incremental constraint solving. To simplify the presentation, we use a simpler language MPY (*miniPython*) in place of Python to explain the details of our algorithm. In practice, our tool supports a fairly large subset of Python including closures, higher order functions, and list comprehensions.

The architecture of AutoProf is shown in Figure 23. The solution strategy to find minimal corrections to a student’s solution is based on a two-phase translation to the Sketch synthesis language. In the first phase, the Program Rewriter uses the correction rules to translate the solution into a language we call $\widetilde{\text{MPY}}$; this language provides us with a concise notation to describe sets of MPY candidate programs, together with a cost model to reflect the number of corrections associated with each program in this set. In the second phase, this $\widetilde{\text{MPY}}$ program is translated into a sketch program by the Sketch Translator,

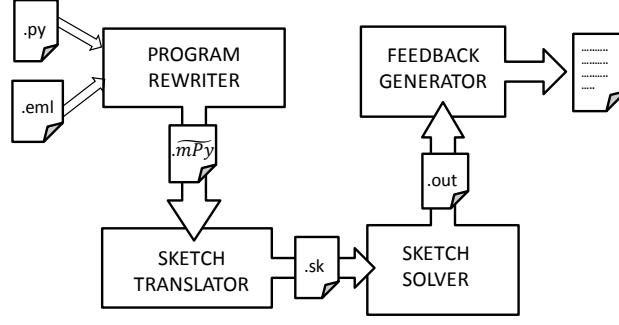


Figure 23: The architecture of our AutoProf system.

which is then solved using the SKETCH synthesis system. After the synthesizer finds a solution, the Feedback Generator extracts the choices made by the synthesizer and uses them to generate the corresponding feedback in natural language.

3.2 SPECIFICATION MECHANISM

The specification mechanism of the AutoProf tool consists of a reference implementation that specifies the correct functional behavior of a student submission. The reference implementation for the `computeDeriv` problem is shown in Figure 21. Since Python is a dynamic language and we use a statically typed synthesis system SKETCH for synthesizing corrections, the reference implementation also needs to specify the types of the function arguments and the return type. These types are specified by appending the corresponding types at the end of argument names and function name.

3.3 HYPOTHESIS SPACE OF CORRECTIONS

The hypothesis space of corrections searched over by the AutoProf tool is defined by the application of error model on a student implementation. We first describe EML, the Error Model Language to write error models and present its syntax and semantics. An error model consists of a collection of rewrite rules that correspond to potential corrections to common mistakes students are making on a given problem. We define the semantics of an error model over a simple Python-like imperative language `mPy` to obtain a large set of Python programs represented succinctly using the language `mPy`. Finally, we present a syntactic translation of an error model to a transformation function that transforms an `mPy` program to an `mPy` program.

3.3.1 EML: Error Model Language

The second input to the AutoProf tool is an error model that describes a set of correction rules that corresponds to the common mistakes students make for a given problem. The error models are described using the error model language EML. We describe the syntax and semantics of EML. An EML error model consists of a set of rewrite rules that captures the potential corrections for mistakes that students might make in their solutions. We define the rewrite rules over a simple Python-like imperative language mPY. A rewrite rule transforms a program element in mPY to a set of weighted mPY program elements. This weighted set of mPY program elements is represented succinctly as an $\widetilde{\text{mPY}}$ program element, where $\widetilde{\text{mPY}}$ extends the mPY language with set-exprs (sets of expressions) and set-stmts (sets of statements). The weight associated with a program element in this set denotes the cost of performing the corresponding correction. An error model transforms an mPY program to an $\widetilde{\text{mPY}}$ program by recursively applying the rewrite rules. We show that this transformation is deterministic and is guaranteed to terminate on *well-formed* error models.

$$\begin{aligned}
 \text{Arith Expr } a &:= n \mid [] \mid v \mid a[a] \mid a_0 \text{ op}_a a_1 \\
 &\mid [a_1, \dots, a_n] \mid f(a_0, \dots, a_n) \\
 &\mid a_0 \text{ if } b \text{ else } a_1 \\
 \text{Arith Op } \text{op}_a &:= + \mid - \mid \times \mid / \mid ** \\
 \text{Bool Expr } b &:= \text{not } b \mid a_0 \text{ op}_c a_1 \mid b_0 \text{ op}_b b_1 \\
 \text{Comp Op } \text{op}_c &:= == \mid < \mid > \mid \leq \mid \geq \\
 \text{Bool Op } \text{op}_b &:= \text{and} \mid \text{or} \\
 \text{Stmt Expr } s &:= v = a \mid s_0; s_1 \mid \text{while } b : s \\
 &\mid \text{if } b : s_0 \text{ else: } s_1 \\
 &\mid \text{for } a_0 \text{ in } a_1 : s \mid \text{return } a \\
 \text{Func Def. } p &:= \text{def } f(a_1, \dots, a_n) : s
 \end{aligned}$$

Figure 24: The syntax of a simple Python-like language mPY.

mPY AND $\widetilde{\text{mPY}}$ LANGUAGES The syntax of mPY and $\widetilde{\text{mPY}}$ languages is shown in Figure 24 and Figure 25 respectively. The purpose of $\widetilde{\text{mPY}}$ language is to represent a large collection of mPY programs succinctly. The $\widetilde{\text{mPY}}$ language consists of set-expressions (\tilde{a} and \tilde{b}) and set-statements (\tilde{s}) that represent a weighted set of corresponding mPY expressions and statements respectively. For example, the set expression $\{\boxed{n_0}, \dots, n_k\}$ represents a weighted set of constant integers where n_0 denotes the default integer value associated with cost 0 and

$$\begin{aligned}
\text{Arith set-expr } \tilde{a} &:= a \mid \{\boxed{\tilde{a}_0}, \dots, \tilde{a}_n\} \mid \tilde{a}[\tilde{a}] \mid \tilde{a}_0 \widetilde{\text{op}}_a \tilde{a}_1 \\
&\mid [\tilde{a}_0, \dots, \tilde{a}_n] \mid \tilde{f}(\tilde{a}_0, \dots, \tilde{a}_n) \\
\text{set-op } \widetilde{\text{op}}_x &:= \text{op}_a \mid \{\widetilde{\text{op}}_{x_0}, \dots, \widetilde{\text{op}}_{x_n}\} \\
\text{Bool set-expr } \tilde{b} &:= b \mid \{\boxed{\tilde{b}_0}, \dots, \tilde{b}_n\} \mid \text{not } \tilde{b} \mid \tilde{a}_0 \widetilde{\text{op}}_c \tilde{a}_1 \mid \tilde{b}_0 \widetilde{\text{op}}_b \tilde{b}_1 \\
\text{Stmt set-expr } \tilde{s} &:= s \mid \{\boxed{\tilde{s}_0}, \dots, \tilde{s}_n\} \mid \tilde{v} := \tilde{a} \mid \tilde{s}_0; \tilde{s}_1 \\
&\mid \text{while } \tilde{b} : \tilde{s} \mid \text{for } \tilde{a}_0 \text{ in } \tilde{a}_1 : \tilde{s} \\
&\mid \text{if } \tilde{b} : \tilde{s}_0 \text{ else} : \tilde{s}_1 \mid \text{return } \tilde{a} \\
\text{Func Def } \tilde{p} &:= \text{def } f(a_1, \dots, a_n) \tilde{s}
\end{aligned}$$

Figure 25: The syntax of language $\widetilde{\text{mPy}}$ for succinct representation of a large number of mPy programs.

all other integer constants (n_1, \dots, n_k) are associated with cost 1. The sets of composite expressions are represented succinctly in terms of sets of their constituent sub-expressions. For example, the composite expression $\{\boxed{a_0}, a_0 + 1\} \{<, \leq, >, \geq, ==, \neq\} \{\boxed{a_1}, a_1 + 1, a_1 - 1\}$ represents 36 mPy expressions.

Each mPy program in the set of programs represented by an $\widetilde{\text{mPy}}$ program is associated with a cost (weight) that denotes the number of modifications performed in the original program to obtain the transformed program. This cost allows AutoProf to search for corrections that require minimum number of modifications. The weighted set of mPy programs is defined using the $\llbracket \cdot \rrbracket$ function shown in Figure 26. The $\llbracket \cdot \rrbracket$ function on mPy expressions such as a returns a singleton set $\{(a, 0)\}$ consisting of the corresponding expression associated with cost 0. On set-expressions of the form $\{\boxed{\tilde{a}_0}, \dots, \tilde{a}_n\}$, the function returns the union of the weighted set of mPy expressions corresponding to the default set-expression $\llbracket \tilde{a}_0 \rrbracket$ and the weighted set of expressions corresponding to other set-expressions $(\tilde{a}_1, \dots, \tilde{a}_n)$, where each expression in $\llbracket \tilde{a}_i \rrbracket$ is associated with an additional cost of 1. On composite expressions, the function computes the weighted set recursively by taking the cross-product of weighted sets of its constituent sub-expressions and adding their corresponding costs. For example, the weighted set for composite expression $\tilde{x}[\tilde{y}]$ consists of an expression $x_i[y_j]$ associated with cost $c_{x_i} + c_{y_j}$ for each $(x_i, c_{x_i}) \in \llbracket \tilde{x} \rrbracket$ and $(y_j, c_{y_j}) \in \llbracket \tilde{y} \rrbracket$.

SYNTAX OF EML An EML error model consists of a set of correction rules that are used to transform an mPy program to an $\widetilde{\text{mPy}}$ program. A *correction rule* \mathcal{C} is written as a rewrite rule $L \rightarrow R$, where L and R denote a *program element* in mPy and $\widetilde{\text{mPy}}$ respectively. A program element can either be a term, an expression, a statement, a method

$$\begin{aligned}
\llbracket \mathbf{a} \rrbracket &= \{(a, 0)\} \\
\llbracket \boxed{\tilde{a}_0}, \dots, \tilde{a}_n \rrbracket &= \llbracket \tilde{a}_0 \rrbracket \cup \{(a, c+1) \mid (a, c) \in \llbracket \tilde{a}_i \rrbracket, 0 < i \leq n\} \\
\llbracket \tilde{a}_0[\tilde{a}_1] \rrbracket &= \{(a_0[a_1], c_0 + c_1) \mid (a_0, c_0) \in \llbracket \tilde{a}_0 \rrbracket, (a_1, c_1) \in \llbracket \tilde{a}_1 \rrbracket\} \\
\llbracket \tilde{a}_0 \widetilde{\text{op}}_x \tilde{a}_1 \rrbracket &= \{(a_0 \text{ op}_x a_1, c_0 + c_x + c_1) \mid (a_0, c_0) \in \llbracket \tilde{a}_0 \rrbracket, \\
&\quad (\text{op}_x, c_x) \in \llbracket \widetilde{\text{op}}_x \rrbracket, (a_1, c_1) \in \llbracket \tilde{a}_1 \rrbracket, x \in \{a, b, c\}\} \\
\llbracket \tilde{f}(\tilde{a}_1, \dots, \tilde{a}_n) \rrbracket &= \{(f(a_1, \dots, a_n), c_f + c_1 + \dots + c_n) \mid (f, c_f) \in \llbracket \tilde{f} \rrbracket, \\
&\quad (a_i, c_i) \in \llbracket \tilde{a}_i \rrbracket\} \\
\llbracket \text{op}_a \rrbracket &= \{(\text{op}_a, 0)\} \\
\llbracket \boxed{\widetilde{\text{op}}_{a_0}}, \dots, \widetilde{\text{op}}_{a_n} \rrbracket &= \llbracket \widetilde{\text{op}}_{a_0} \rrbracket \cup \{(\text{op}_a, c+1) \mid (\text{op}_a, c) \in \llbracket \widetilde{\text{op}}_{a_i} \rrbracket\} \\
\llbracket \mathbf{b} \rrbracket &= \{(b, 0)\} \\
\llbracket \boxed{\tilde{b}_0}, \dots, \tilde{b}_n \rrbracket &= \llbracket \tilde{b}_0 \rrbracket \cup \{(b, c+1) \mid (b, c) \in \llbracket \tilde{b}_i \rrbracket, 0 < i \leq n\} \\
\llbracket \mathbf{s} \rrbracket &= \{(s, 0)\} \\
\llbracket \boxed{\tilde{s}_0}, \dots, \tilde{s}_n \rrbracket &= \llbracket \tilde{s}_0 \rrbracket \cup \{(s, c+1) \mid (s, c) \in \llbracket \tilde{s}_i \rrbracket, 0 < i \leq n\} \\
\llbracket \tilde{v} := \tilde{a} \rrbracket &= \{(v := a, c_0 + c_1) \mid (v, c_0) \in \llbracket \tilde{v} \rrbracket, (a, c_1) \in \llbracket \tilde{a} \rrbracket\} \\
\llbracket \tilde{s}_0; \tilde{s}_1 \rrbracket &= \{(s_0; s_1, c_0 + c_1) \mid (s_0, c_0) \in \llbracket \tilde{s}_0 \rrbracket, (s_1, c_1) \in \llbracket \tilde{s}_1 \rrbracket\} \\
\llbracket \text{if } \tilde{b} \text{ then } \tilde{s}_0 \text{ else } \tilde{s}_1 \rrbracket &= \{(\text{if } b \text{ then } s_0 \text{ else } s_1, c_b + c_0 + c_1) \mid (b, c_b) \in \llbracket \tilde{b} \rrbracket, \\
&\quad (s_0, c_0) \in \llbracket \tilde{s}_0 \rrbracket, (s_1, c_1) \in \llbracket \tilde{s}_1 \rrbracket\} \\
\llbracket \text{while } \tilde{b} \text{ do } \tilde{s} \rrbracket &= \{(\text{while } b \text{ do } s, c_b + c_s) \mid (b, c_b) \in \llbracket \tilde{b} \rrbracket, (s, c_s) \in \llbracket \tilde{s} \rrbracket\} \\
\llbracket \text{return } \tilde{a} \rrbracket &= \{(\text{return } a, c) \mid (a, c) \in \llbracket \tilde{a} \rrbracket\}
\end{aligned}$$

Figure 26: The $\llbracket \cdot \rrbracket$ function that translates an $\widetilde{\text{mPy}}$ program to a weighted set of mPy programs.

or the program itself. The left hand side (L) denotes an $\widetilde{\text{MPY}}$ program element that is pattern matched to be transformed to an $\widetilde{\text{MPY}}$ program element denoted by the right hand side (R). The left hand side of the rule can use free variables whereas the right hand side can only refer to the variables present in the left hand side. The language also supports a special ' (prime) operator that can be used to *tag* sub-expressions in R that are further transformed recursively using the error model. The rules use a shorthand notation ?a (in the right hand side) to denote the set of all variables that are of the same type as the type of expression a and are in scope at the corresponding program location. We assume each correction rule is associated with cost 1, but it can be easily extended to different costs to account for different severity of mistakes.

$$\begin{aligned}
\text{INDR: } v[a] &\rightarrow v[\{a+1, a-1, ?a\}] \\
\text{INITR: } v = n &\rightarrow v = \{n+1, n-1, 0\} \\
\text{RANR: } \text{range}(a_0, a_1) &\rightarrow \text{range}(\{0, 1, a_0-1, a_0+1\}, \\
&\quad \{a_1+1, a_1-1\}) \\
\text{COMPR: } a_0 \text{ op}_c a_1 &\rightarrow \{\{a'_0-1, ?a_0\} \widetilde{\text{op}}_c \{a'_1-1, 0, 1, ?a_1\}, \\
&\quad \text{True, False}\} \\
&\quad \text{where } \widetilde{\text{op}}_c = \{<, >, \leq, \geq, ==, \neq\} \\
\text{RETR: } \text{return } a &\rightarrow \text{return}([0] \text{ if } \text{len}(a) == 1 \text{ else } a, \\
&\quad a[1:] \text{ if } (\text{len}(a) > 1) \text{ else } a)
\end{aligned}$$

Figure 27: The error model \mathcal{E} for the `computeDeriv` problem. The default choices that do not change the original expression are added by default by the translation function.

Example 3.3.1. The error model for the `computeDeriv` problem is shown in Figure 27. The `INDR` rewrite rule transforms the list access indices. The `INITR` rule transforms the right hand side of constant initializations. The `RANR` rule transforms the arguments for the range function; similar rules are defined in the model for other range functions that take one and three arguments. The `COMPR` rule transforms the operands and operator of the comparisons. The `RETR` rule adds the two common corner cases of returning `[0]` when the length of input list is 1, and the case of deleting the first list element before returning the list. Note that these rewrite rules define the corrections that can be performed optionally; the zero cost (default) case of not correcting a program element is added automatically as described in Section 3.3.

Definition 3.3.1. Well-formed Rewrite Rule : A rewrite rule $\mathcal{C} : L \rightarrow R$ is defined to be well-formed if all tagged sub-terms t' in R have a smaller size syntax tree than that of L .

The rewrite rule $\mathcal{C}_1 : v[a] \rightarrow \{(v[a])' + 1\}$ is not a well-formed rewrite rule as the size of the tagged sub-term $(v[a])$ of R is the same as that of the left hand side L . On the other hand, the rewrite rule $\mathcal{C}_2 : v[a] \rightarrow \{v'[a'] + 1\}$ is well-formed.

Definition 3.3.2. Well-formed Error Model : An error model \mathcal{E} is defined to be well-formed if all of its constituent rewrite rules $\mathcal{C}_i \in \mathcal{E}$ are well-formed.

3.3.2 Rewriting Student Solution using an Error Model

An error model \mathcal{E} is syntactically translated to a function $\mathcal{T}_{\mathcal{E}}$ that transforms an mPY program to an $\widetilde{\text{mPY}}$ program. The $\mathcal{T}_{\mathcal{E}}$ function first traverses the program element w in the default way, i.e. no transformation happens at this level of the syntax tree, and the function is called recursively on all of its top-level sub-terms t to obtain the transformed element $w_0 \in \widetilde{\text{mPY}}$. For each correction rule $\mathcal{C}_i : L_i \rightarrow R_i$ in the error model \mathcal{E} , the function contains a **Match** expression that matches the term w with the left hand side of the rule L_i (with appropriate unification of the free variables in L_i). If the match succeeds, it is transformed to a term $w_i \in \widetilde{\text{mPY}}$ as defined by the right hand side R_i of the rule after calling the $\mathcal{T}_{\mathcal{E}}$ function on each of its tagged sub-terms t' . Finally, the method returns the set of all transformed terms $\{\boxed{w_0}, \dots, w_n\}$.

```

 $\mathcal{T}_{\mathcal{E}_1}(w : \text{mPY}) : \widetilde{\text{mPY}} =$ 
  let  $w_0 = w[t \rightarrow \mathcal{T}_{\mathcal{E}_1}(t)]$  in (*  $t$  : a sub-term of  $w$  *)
  let  $w_1 = \text{Match } w \text{ with}$ 
     $v[a] \rightarrow v[[a + 1, a - 1]]$  in
  let  $w_2 = \text{Match } w \text{ with}$ 
     $a_0 \text{ op}_c a_1 \rightarrow \{\mathcal{T}_{\mathcal{E}_1}(a_0) - 1, 0\} \text{ op}_c$ 
     $\{\mathcal{T}_{\mathcal{E}_1}(a_1) - 1, 0\}$  in
   $\{\boxed{w_0}, w_1, w_2\}$ 

```

Figure 28: The $\mathcal{T}_{\mathcal{E}_1}$ method for error model \mathcal{E}_1 .

Example 3.3.2. Consider an error model \mathcal{E}_1 consisting of the following three correction rules:

$$\begin{aligned}
 \mathcal{C}_1 : v[a] &\rightarrow v[[a - 1, a + 1]] \\
 \mathcal{C}_2 : a_0 \text{ op}_c a_1 &\rightarrow \{a'_0 - 1, 0\} \text{ op}_c \{a'_1 - 1, 0\} \\
 \mathcal{C}_3 : v[a] &\rightarrow ?v[a]
 \end{aligned}$$

The transformation function $\mathcal{T}_{\mathcal{E}_1}$ for the error model \mathcal{E}_1 is shown in Figure 28.

$$\begin{aligned}
\mathcal{T}(x[i] < y[j]) &\equiv \{ \boxed{\mathcal{T}(x[i]) < \mathcal{T}(y[j])}, \{ \mathcal{T}(x[i]) - 1, 0 \} < \{ \mathcal{T}(y[j]) - 1, 0 \} \} \\
\mathcal{T}(x[i]) &\equiv \{ \boxed{\mathcal{T}(x)[\mathcal{T}(i)]}, x[\{i+1, i-1\}], y[i] \} \\
\mathcal{T}(y[j]) &\equiv \{ \boxed{\mathcal{T}(y)[\mathcal{T}(j)]}, y[\{j+1, j-1\}], x[j] \} \\
\mathcal{T}(x) &\equiv \{ \boxed{x} \} \quad \mathcal{T}(i) \equiv \{ \boxed{i} \} \quad \mathcal{T}(y) \equiv \{ \boxed{y} \} \quad \mathcal{T}(j) \equiv \{ \boxed{j} \} \\
\mathcal{T}(x[i] < y[j]) &\equiv \{ \{ \boxed{\boxed{x}[\boxed{i}]} , x[\{i+1, i-1\}], y[i] \} < \{ \boxed{\boxed{y}[\boxed{j}]} , y[\{j+1, j-1\}], x[j] \} \} , \\
&\quad \{ \{ \boxed{\boxed{x}[\boxed{i}]} , x[\{i+1, i-1\}], y[i] \} - 1, 0 \} < \{ \{ \boxed{\boxed{y}[\boxed{j}]} , y[\{j+1, j-1\}], x[j] \} - 1, 0 \} \}
\end{aligned}$$

Figure 29: Application of $\mathcal{T}_{\mathcal{E}_1}$ (abbreviated \mathcal{T}) on expression $(x[i] < y[j])$.

The recursive steps of application of $\mathcal{T}_{\mathcal{E}_1}$ function on expression $(x[i] < y[j])$ are shown in Figure 29. This example illustrates two interesting features of the transformation function:

- **Nested Transformations** : Once a rewrite rule $L \rightarrow R$ is applied to transform a program element matching L to R , the instructor may want to apply another rewrite rule on only a few sub-terms of R . For example, she may want to avoid transforming the sub-terms which have already been transformed by some other correction rule. The EML language facilitates making such distinction between the sub-terms for performing nested corrections using the ' (prime) operator. Only the sub-terms in R that are tagged with the prime operator are visited for applying further transformations (using the $\mathcal{T}_{\mathcal{E}}$ function recursively on its tagged sub-terms t'), whereas the non-tagged sub-terms are not transformed any further. After applying the rewrite rule \mathcal{C}_2 in the example, the sub-terms $x[i]$ and $y[j]$ are further transformed by applying rewrite rules \mathcal{C}_1 and \mathcal{C}_3 .
- **Ambiguous Transformations** : While transforming a program using an error model, it may happen that there are multiple rewrite rules that pattern match the program element w . After applying the rewrite rule \mathcal{C}_2 in the example, there are two rewrite rules \mathcal{C}_1 and \mathcal{C}_3 that pattern match the terms $x[i]$ and $y[j]$. After applying one of these rules (\mathcal{C}_1 or \mathcal{C}_3) to an expression $v[a]$, we cannot apply the other rule to the transformed expression. In such ambiguous cases, the $\mathcal{T}_{\mathcal{E}}$ function creates a copy of the transformed program element (w_i) for each ambiguous choice and then computes the union of all such elements to

```

def computeDeriv(poly):
    deriv = []
    zero = 0
    if ({len(poly) == 1}, False):
        return {deriv, [0]}
    for e in range ({0}, 1, len(poly)):
        if ({poly[e] == 0}, False):
            zero += 1
        else:
            deriv.append(poly[e]*e)
    return {deriv, [0]}

```

Figure 30: The resulting $\widetilde{\text{mPy}}$ program after applying correction rules to program in Figure 22(a).

obtain the transformed program element. This semantics of handling ambiguity of rewrite rules also matches naturally with the intent of the instructor. If the instructor wanted to perform both transformations together on array accesses, she could have provided a combined rewrite rule such as $v[a] \rightarrow ?v[[a + 1, a - 1]]$.

For the student program shown in Figure 22(d), the three correction rules in the error model described in Section 3.1 induce a space of 32 different candidate programs as shown in Figure 30. This candidate space is fairly small, but the number of candidate programs grow exponentially with the number of correction places in the program and with the number of correction choices in the rules. The error model that we use in our experiments induces a space of more than 10^{12} candidate programs for some of the benchmark problems. In order to search this large space efficiently, the program is translated to a sketch program.

Theorem 3.3.1. Given a well-formed error model \mathcal{E} , the transformation function $\mathcal{T}_{\mathcal{E}}$ always terminates.

Proof. From the definition of well-formed error model, each of its constituent rewrite rule is also well-formed. Hence, each application of a rewrite rule reduces the size of the syntax tree of terms that are required to be visited further for transformation by $\mathcal{T}_{\mathcal{E}}$. Therefore, the $\mathcal{T}_{\mathcal{E}}$ function terminates in a finite number of steps. \square

3.4 SYNTHESIS ALGORITHM

In the previous section, we described the transformation of an mPy program to an $\widetilde{\text{mPy}}$ program based on an error model. We now present the translation of an $\widetilde{\text{mPy}}$ program into a SKETCH program [117] and present the CEGISMIN algorithm for solving the SKETCH program to

```

struct MultiType{
    int val, flag;
    bit bval;
    MTString str; MTTuple tup;
    MTDict dict; MTLList lst;
}

struct MTLList{
    int len;
    MultiType[len] lVals;
}

```

Figure 31: The MultiType struct for encoding Python types.

compute the minimum number of corrections to a student program (represented as an $\widetilde{\text{MPY}}$ program) such that it becomes functionally equivalent with the teacher's reference implementation.

3.4.1 Translation of $\widetilde{\text{MPY}}$ programs to SKETCH

The SKETCH [117] synthesis system allows programmers to write programs while leaving fragments of it unspecified as *holes*; the contents of these holes are filled up automatically by the synthesizer such that the program conforms to a specification provided in terms of a reference implementation. The synthesizer uses the CEGIS algorithm [118] to efficiently compute the values for holes and uses bounded symbolic verification techniques for performing equivalence check of the two implementations.

There are two key aspects in the translation of an $\widetilde{\text{MPY}}$ program to a SKETCH program: (i) the translation of Python-like constructs in $\widetilde{\text{MPY}}$ to SKETCH, and (ii) the translation of set-expr choices in $\widetilde{\text{MPY}}$ to SKETCH functions. The first aspect is specific to the Python language. SKETCH supports high-level features such as closures and higher-order functions which simplifies the translation, but it is statically typed whereas $\widetilde{\text{MPY}}$ programs (like Python) are dynamically typed. The translation models the dynamically typed variables and operations over them using struct types in SKETCH in a way similar to the union types. The second aspect of the translation is the modeling of set-expressions in $\widetilde{\text{MPY}}$ using ?? (holes) in SKETCH, which is language independent.

HANDLING DYNAMIC TYPING OF $\widetilde{\text{MPY}}$ VARIABLES The dynamic variable types in the $\widetilde{\text{MPY}}$ language are modeled using the MultiType struct defined in Figure 31. The MultiType struct consists of a flag field that denotes the dynamic type of a variable and currently supports the following set of types: {INTEGER, BOOL, TYPE, LIST, TUPLE, STRING, DICTIONARY}. The val and bval fields store the value of an integer and a Boolean variable respectively, whereas the str, tup, dict, and lst fields store the value of string, tuple, dictionary, and list variables respectively. The MTLList struct consists of a field len that denotes the length of the list and a field lVals of type array of MultiType

that stores the list elements. For example, the integer value 5 is represented as the value `MultiType(val=5, flag=INTEGER)` and the constant list value `[1,2]` is represented as the value `MultiType(lst=new MTList(len=2,lVals={oneMT, twoMT}), flag=LIST)`, where `oneMT = MultiType(val=1, flag=INTEGER)` and `twoMT = MultiType(val=2, flag=INTEGER)`. The $\widetilde{\text{MPY}}$ expressions and statements are transformed to `SKETCH` functions that perform the corresponding transformations according to Python semantics over `MultiType`. For example, the Python statement `(a = b)` is translated to `assignMT(a, b)`, where the `assignMT` function assigns `MultiType b` to `a`. Similarly, the binary add expression `(a + b)` is translated to `binOpMT(a, b, ADD_OP)` that in turn calls the function `addMT(a,b)` to add `a` and `b` as shown in Figure 32.

```
MultiType addMT(MultiType a, MultiType b){
    assert a.flag == b.flag; // same types can be added
    if(a.flag == INTEGER)    // add for integers
        return new MultiType(val=a.val+b.val, flag = INTEGER);
    if(a.flag == LIST){      // add for lists
        int newLen = a.lst.len + b.lst.len;
        MultiType[newLen] newLVals = a.lst.lVals;
        for(int i=0; i<b.lst.len; i++)
            newLVals[i+a.lst.len] = b.lst.lVals[i];
        return new MultiType(lst = new MTList(lVals=newLVals, len=
            newLen), flag=LIST);}
    ... ..
}
```

Figure 32: The `addMT` function for adding two `MultiType a` and `b`.

TRANSLATION OF $\widetilde{\text{MPY}}$ SET-EXPRESSIONS The second key aspect of this translation is the translation of expression choices in $\widetilde{\text{MPY}}$. The `SKETCH` construct `??` denotes an unknown integer hole that can be assigned any constant integer value by the synthesizer. The expression choices in $\widetilde{\text{MPY}}$ are translated to functions in `SKETCH` that based on the unknown hole values return either the default expression or one of the other expression choices. The function bodies obtained by the application of translation function (Φ) on some of the interesting $\widetilde{\text{MPY}}$ constructs are shown in Figure 33. The `SKETCH` construct `??` (called *hole*) is a placeholder for a constant value, which is filled up by the `SKETCH` synthesizer while solving the constraints to satisfy the given specification.

The singleton sets consisting of an $\widetilde{\text{MPY}}$ expression such as $\{a\}$ are translated simply to the corresponding expression itself. A set-expression of the form $\{\boxed{\tilde{a}_0}, \dots, \tilde{a}_n\}$ is translated recursively to the if expression `if (??) $\Phi(\tilde{a}_0)$ else $\Phi(\{\tilde{a}_1, \dots, \tilde{a}_n\})$` , which means that the synthesizer can optionally select the default set-expression

$\Phi(\tilde{a}_0)$ (by choosing $??$ to be true) or select one of the other choices $(\tilde{a}_1, \dots, \tilde{a}_n)$. The set-expressions of the form $\{\tilde{a}_0, \dots, \tilde{a}_n\}$ are similarly translated but with an additional statement for setting a fresh variable choice_k if the synthesizer selects the non-default choice \tilde{a}_0 .

The translation rules for the assignment statements $(\tilde{a}_0 := \tilde{a}_1)$ results in `if` expressions on both left and right sides of the assignment. The `if` expression choices occurring on the left hand side are desugared to individual assignments. For example, the left hand side expression `if (??) x else y := 10` is desugared to `if (??) x := 10 else y := 10`. The infix operators in $\widetilde{\text{MPY}}$ are first translated to function calls and are then translated to `sketch` using the translation for set-function expressions. The remaining $\widetilde{\text{MPY}}$ expressions are similarly translated recursively.

For example, the set-statement `return {deriv}[0]`; (line 5 in Figure 30) is translated to `return modRetVal0(deriv)`, where the `modRetVal0` function is defined as:

```
MultiType modRetVal0(MultiType a){
    if(??) return a; // default choice
    choiceRetVal0 = True; // non-default choice
    MTList list = new MTList(lVals={new MultiType(val=0,
        flag=INTEGER)}, len=1);
    return new MultiType(lst=list, type = LIST);
}
```

TRANSLATING FUNCTION CALLS The translation of function calls for recursive problems and for problems that require writing a function that uses other sub-functions is parameterized by three options: 1) use the student's implementation of sub-functions, 2) use the teacher's implementation of sub-functions, and 3) treat the sub-functions as uninterpreted functions.

GENERATING THE DRIVER FUNCTIONS The translation phase also generates a `SKETCH` harness that compares the outputs of the translated student and reference implementations on all inputs of a bounded size. The `SKETCH` synthesizer supports the equivalence checking of functions whose input arguments and return values are over `SKETCH` primitive types such as `int`, `bit` and arrays. Therefore, after the translation of $\widetilde{\text{MPY}}$ programs to `SKETCH` programs, we need additional driver functions to integrate the functions over `MultiType` input arguments and return value to the corresponding functions over `SKETCH` primitive types. The driver functions first converts the input arguments over primitive types to corresponding `MultiType` variables using library functions such as `computeMTFromInt`, and then calls the translated $\widetilde{\text{MPY}}$ function with the `MultiType` variables. The returned `MultiType` value is translated back to primitive types using library functions such as `computeIntFromMT`. The driver function for student's

$$\begin{aligned}
\Phi(\{a\}) &= a \\
\Phi(\{\boxed{\tilde{a}_0}, \dots, \tilde{a}_n\}) &= \text{if } (??) \Phi(\tilde{a}_0) \text{ else } \Phi(\{\tilde{a}_1, \dots, \tilde{a}_n\}) \\
\Phi(\{\tilde{a}_0, \dots, \tilde{a}_n\}) &= \text{if } (??) \{\text{choice}_k = 1; \Phi(\tilde{a}_0)\} \\
&\quad \text{else } \Phi(\{\tilde{a}_1, \dots, \tilde{a}_n\}) \\
\Phi(\tilde{a}_0[\tilde{a}_1]) &= \Phi(\tilde{a}_0)[\Phi(\tilde{a}_1)] \\
\Phi(\{f(\tilde{a}_1, \dots, \tilde{a}_n)\}) &= f(\Phi(\tilde{a}_1), \dots, \Phi(\tilde{a}_n)) \\
\Phi(\{\boxed{\tilde{f}_0}, \dots, \tilde{f}_n\}(\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_n)) &= \text{if } (??) \Phi(\tilde{f}_0(\tilde{a}_1, \dots, \tilde{a}_n)) \\
&\quad \text{else } \Phi(\{\tilde{f}_1, \dots, \tilde{f}_n\}(\tilde{a}_1, \dots, \tilde{a}_n)) \\
\Phi(\{\tilde{f}_0, \dots, \tilde{f}_n\}(\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_n)) &= \text{if } (??) \{\text{choice}_k = 1; \Phi(\tilde{f}_0(\tilde{a}_1, \dots, \tilde{a}_n))\} \\
&\quad \text{else } \Phi(\{\tilde{f}_1, \dots, \tilde{f}_n\}(\tilde{a}_1, \dots, \tilde{a}_n)) \\
\Phi(\tilde{a}_0 = \tilde{a}_1) &= \Phi(\tilde{a}_0) := \Phi(\tilde{a}_1) \\
\Phi(\tilde{s}_0; \tilde{s}_1) &= \Phi(\tilde{s}_0); \Phi(\tilde{s}_1) \\
\Phi(\text{if } \tilde{b} : \tilde{s}_0 \text{ else} : \tilde{s}_1) &= \text{if } (\Phi(\tilde{b})) \{\Phi(\tilde{s}_0)\} \text{ else } \{\Phi(\tilde{s}_1)\} \\
\Phi(\text{while } \tilde{b} : \tilde{s}) &= \text{while } (\Phi(\tilde{b})) \{\Phi(\tilde{s})\} \\
\Phi(\text{return } \tilde{a}) &= \text{return } \Phi(\tilde{a})
\end{aligned}$$

Figure 33: The translation rules for converting $\widetilde{\text{MPY}}$ set-exprs to corresponding `SKETCH` function bodies.

programs also consists of additional statements of the form `if(choicek) totalCost++;` and the statement `minimize(totalCost)`, which tells the synthesizer to compute a solution to the Boolean variables `choicek` that minimizes the `totalCost` variable.

For example in case of the `computeDeriv` function, with bounds of $n = 4$ for both the number of integer bits and the maximum length of input list, the harness matches the output of the two implementations for more than 2^{16} different input values as opposed to 10 test-cases used in 6.00x. Even though `AutoProf` searches over a much larger space of inputs, it can only check for inputs upto a bounded size. This tradeoff can lead it to declare some student programs correct when they are not, but this can be easily taken care of by complementing `AutoProf` with traditional test cases to perform an additional result verification step. The sketch generated for $\widetilde{\text{MPY}}$ program in Figure 30 is shown in Figure 34.

3.4.2 CEGISMIN: Incremental Solving for the Minimize holes

We can use the standard CEGIS algorithm in `SKETCH` [117] to solve the resulting sketch files, but since the algorithm searches for any possible solution, it often comes up with corrections that requires a lot of changes. Ideally, we would like to compute minimal changes since that would most likely correspond to the approach that student had

```

#include <multiTypeLib.c>
int totalCost = 0;
bit choiceComp0=0, choiceComp1=0;
bit choiceIdx0=0, choiceRetVal0=0, choiceRetVal1=0;

#define zeroMT new MultiType(val=0, type = INTEGER)
#define oneMT new MultiType(val=1, type = INTEGER)
#define emptyListMT = new MultiType(lst = new MTList(lVals={}, len=0),
    type = LIST)

MultiType computeDeriv(MultiType poly){
    MultiType deriv, zero;
    assignMT(deriv, emptyListMT);
    assignMT(zero, zeroMT);
    if(getBoolValue(modComp0(compareMT(len(poly), oneMT, COMP_EQ))))
        return modRetVal0(deriv);
    MultiType iterList = range2(modIdx0(zeroMT), len(poly));
    for(int i=0; i< iterList.lst.len; i++){
        MultiType e = iterList.lst.lVals[i];
        if(getBoolValue(modComp1(compareMT(sub(poly,e), zeroMT, COMP_EQ))))
            assignMT(zero, binOpMT(zero, oneMT, ADD_OP));
        else
            append(deriv, binOpMT(sub(poly,e), e, MUL_OP));
    }
    return modRetVal1(deriv);
}

MultiType modComp0(MultiType b){
    if(??) return b;
    choiceComp0 = 1;
    return FalseMT();
}

MultiType modIdx0(MultiType a){
    if(??) return a;
    choiceIdx0 = 1;
    return incrementOne(a);
}

MultiType modRetVal0(MultiType a){
    if(??) return a;
    choiceRetVal0 = 1;
    MTList list = new MTList(lVals={zeroMT}, len =1);
    return new MultiType(lst=list, type = LIST);
}

int[N] computeDeriv_driver(int N, int[N] poly) implements
    computeDeriv_teacher_driver{
    MultiType polyMT = createMTFromArray(N, poly);
    MultiType resultMT = computeDeriv(polyMT);
    int[N] result = getArrayFromMT(resultMT);
    if(choiceComp0) totalCost = totalCost + 1;
    if(choiceComp1) totalCost = totalCost + 1;
    if(choiceIdx0) totalCost = totalCost + 1;
    if(choiceRetVal0) totalCost = totalCost + 1;
    if(choiceRetVal1) totalCost = totalCost + 1;
    minimize(totalCost);
    return result;
}

```

Figure 34: The sketch generated for $\widetilde{\text{mPy}}$ program in Figure 30.

Algorithm 1 CEGISMIN Algorithm for Minimize expression

```

1:  $\sigma_0 \leftarrow \sigma_{\text{random}}, \quad i \leftarrow 0, \quad \Phi_0 \leftarrow \Phi, \quad \phi_p \leftarrow \text{null}$ 
2: while (True)
3:    $i \leftarrow i + 1$ 
4:    $\Phi_i \leftarrow \text{Synth}(\sigma_{i-1}, \Phi_{i-1})$  /* Synthesis Phase */
5:   if ( $\Phi_i = \text{UNSAT}$ ) /* Synthesis Fails */
6:     if ( $\Phi_{\text{prev}} = \text{null}$ ) return UNSAT_SKETCH
7:     else return PE(P,  $\phi_p$ )
8:   choose  $\phi \in \Phi_i$ 
9:    $\sigma_i \leftarrow \text{Verify}(\phi)$  /* Verification Phase */
10:  if ( $\sigma_i = \text{null}$ ) /* Verification Succeeds */
11:    ( $\text{minHole}, \text{minHoleValue}$ )  $\leftarrow \text{getMinHoleValue}(\phi)$ 
12:     $\phi_p \leftarrow \phi$ 
13:     $\Phi_i \leftarrow \Phi_i \cup \{\text{encode}(\text{minHole} < \text{minHoleVal})\}$ 

```

in mind. But adding minimality constraints to SAT-based approaches is challenging. We can try to use binary search or MAX-SAT solvers to solve for minimality constraints, but we found them to be too inefficient in practice to solve the sketches for these problems.

We extend the CEGIS algorithm to obtain the CEGISMIN algorithm shown in Algorithm 1 for efficiently solving sketches that include a minimization constraint. The input state of the sketch program is denoted by σ and the sketch constraint store is denoted by Φ . Initially, the input state σ_0 is assigned a random input state value and the constraint store Φ_0 is assigned the constraint set obtained from the sketch program. The variable ϕ_p stores the previous satisfiable hole values and is initialized to null. In each iteration of the loop, the synthesizer first performs the inductive synthesis phase where it shrinks the constraints set Φ_{i-1} to Φ_i by removing behaviors from Φ_{i-1} that do not conform to the input state σ_{i-1} . If the constraint set becomes unsatisfiable, it either returns the sketch completed with hole values from the previous solution if one exists, otherwise it returns UNSAT. On the other hand, if the constraint set is satisfiable, then it first chooses a conforming assignment to the hole values and goes into the verification phase where it tries to verify the completed sketch. If the verifier fails, it returns a counter-example input state σ_i and the synthesis-verification loop is repeated. If the verification phase succeeds, instead of returning the result as is done in the CEGIS algorithm, the CEGISMIN algorithm computes the value of minHole from the constraint set ϕ , stores the current satisfiable hole solution ϕ in ϕ_p , and adds an additional constraint $\{\text{minHole} < \text{minHoleVal}\}$ to the constraint set Φ_i . The synthesis-verification loop is then repeated with this additional constraint to find a conforming value for the minHole variable that is smaller than the current value in ϕ .

3.4.3 Mapping SKETCH solution to generate feedback

Each correction rule in the error model is associated with a feedback message, e.g. the correction rule for variable initialization $v = n \rightarrow v = \{n + 1\}$ in the `computeDeriv` error model is associated with the message “Increment the right hand side of the initialization by 1”. After the SKETCH synthesizer finds a solution to the constraints, the tool maps back the values of unknown integer holes to their corresponding expression choices. These expression choices are then mapped to natural language feedback using the messages associated with the corresponding correction rules, together with the line numbers. If the synthesizer returns UNSAT, the tool reports that the student solution can not be fixed.

3.5 USER INTERACTION MODEL

An instructor first starts with a basic error model corresponding to the set of common mistakes that students typically make across *all* programming problems. This lets AutoProf provide common generic feedback for some of the student submissions. The instructor then chooses one of the incorrect student submissions for which AutoProf could not generate feedback, and adds corresponding correction rules to the model so that in the next iteration AutoProf would generate feedback for such mistakes. This iterative process of incrementally adding correction rules to the error model helps instructors avoid the prohibitively expensive step of going through each student solution, and analyze only a handful of student submission to construct a relatively thorough problem-specific error model.

AutoProf also allows instructors to specify the cost value corresponding to each correction rule. This cost metric can be updated iteratively as more rules are added to the error model. An instructor can also specify the level of feedback detail that AutoProf should generate for each correction rule. For some corrections, the instructor may wish to provide a very generic feedback, whereas for some other corrections, the instructor may wish to provide exact feedback corresponding to how to the fix and line number.

3.6 IMPLEMENTATION AND EXPERIMENTS

We now briefly describe some of the implementation details of AutoProf, and then describe the experiments we performed to evaluate AutoProf over the benchmark problems.

3.6.1 *Implementation*

The frontend of AutoProf is implemented in Python itself and uses the Python `ast` module to convert a Python program to a `SKETCH` program. The backend system that solves the sketch is implemented as a wrapper over the `SKETCH` system that is extended with the `CEGISMIN` algorithm. The feedback generator, implemented in Python, parses the output generated by the backend system and translates it to corresponding high level feedback in natural language. The error models in AutoProf are currently written in terms of rewrite rules over the Python AST. In addition to the Python AutoProf, we have also developed a prototype for the C# language, which we built on top of the Microsoft Roslyn compiler framework. The C# prototype supports a smaller subset of the language relative to the Python tool but nevertheless it was useful in helping us evaluate the potential of our technique on a different language.

3.6.2 *Benchmarks*

We created our benchmark set with problems taken from the Introduction to Programming course at MIT (6.00) and the edX version of the class (6.00x) offered in 2012. Our benchmark set includes most problems from the first four weeks of the course. We only excluded (i) a problem that required more detailed floating point reasoning than what we currently handle, (ii) a problem that required file i/o which we currently do not model, and (iii) a handful of trivial finger exercises. To evaluate the applicability to C#, we created a few programming exercises² on `PEX4FUN` [126] that were based on loop-over-arrays and dynamic programming from an AP level exam³. A brief description of each benchmark problem follows:

- `prodBySum-6.00` : Compute the product of two numbers `m` and `n` using only the sum operator.
- `oddTuples-6.00` : Given a tuple `l`, return a tuple consisting of every other element of `l`.
- `compDeriv-6.00` : Compute the derivative of a polynomial `poly`, where the coefficients of `poly` are represented as a list.
- `evalPoly-6.00` : Compute the value of a polynomial (represented as a list) at a given value `x`.
- `compBal-stdin-6.00` : Print the values of monthly installment necessary to purchase a car in one year, where the inputs `car price` and `interest rate` (compounded monthly) are provided from `stdin`.

² <http://pexforfun.com/learnbeginningprogramming>

³ AP exams allow high school students in the US to earn college level credit.

- `compDeriv-6.00x` : `compDeriv` problem from the EdX class.
- `evalPoly-6.00x` : `evalPoly` problem from the EdX class.
- `oddTuples-6.00x` : `oddTuples` problem from the EdX class.
- `iterPower-6.00x` : Compute the value m^n using only the multiplication operator, where m and n are integers.
- `recurPower-6.00x` : Compute the value m^n using recursion.
- `iterGCD-6.00x` : Compute the greatest common divisor (gcd) of two integers m and n using an iterative algorithm.
- `hangman1-str-6.00x` : Given a string `secretWord` and a list of guessed letters `lettersGuessed`, return `True` if all letters of `secretWord` are in `lettersGuessed`, and `False` otherwise.
- `hangman2-str-6.00x` : Given a string `secretWord` and a list of guessed letters `lettersGuessed`, return a string where all letters of `secretWord` that have not been guessed yet (i.e. not present in `lettersGuessed`) are replaced by the letter `'_'`.
- `stock-market-I(C#)` : Given a list of stock prices, check if the stock is stable, i.e. if the price of stock has changed by more than \$10 in consecutive days on less than 3 occasions over the duration.
- `stock-market-II(C#)` : Given a list of stock prices and a start and end day, check if the difference between the maximum and minimum stock prices over the duration from start and end day is less than \$20.
- `restaurant_rush (C#)` : A variant of maximum contiguous subset sum problem.

3.6.3 Experiments

We now present various experiments we performed to evaluate AutoProf on the benchmark problems.

PERFORMANCE Table 1 shows the number of student attempts corrected for each benchmark problem as well as the time taken by AutoProf to provide the feedback. The experiments were performed on a 2.4GHz Intel Xeon CPU with 16 cores and 16GB RAM. The experiments were performed with bounds of 4 bits for input integer values and maximum length 4 for input lists. For each benchmark problem, we first removed the student attempts with syntax errors to get the Test Set on which we ran AutoProf. We then separated the attempts which were correct to measure the effectiveness of AutoProf

```
def evaluatePoly(poly, x):
    result = 0
    for i in list(poly):
        result += i*x**poly.index(i)
    return result
```

(a)

```
def getGuessedWord(secretWord, lettersGuessed):
    for letter in lettersGuessed:
        secretWord = secretWord.replace(letter, '_')
    return secretWord
```

(b)

Figure 35: An example of big conceptual error for a student's attempt for (a) evalPoly and (b) hangman2-str problems.

on the incorrect attempts. AutoProf was able to provide appropriate corrections as feedback for 64% of all incorrect student attempts in around 10 seconds on average. The remaining 36% of incorrect student attempts on which AutoProf could not provide feedback fall in one of the following categories:

- **Completely incorrect solutions:** We observed many student attempts that were empty or performing trivial computations such as printing strings and variables.
- **Big conceptual errors:** A common error we found in the case of eval-poly-6.00x was that a large fraction of incorrect attempts (260/541) were using the list function index to get the index of a value in the list (e.g. see Figure 35(a)), whereas the index function returns the index of first occurrence of the value in the list. Another example of this class of error for the hangman2-str problem is shown in Figure 35(b), where the solution replaces the guessed letters in the secretWord by '_' instead of replacing the letters that are not yet guessed. The correction of some other errors in this class involves introducing new program statements or moving statements from one program location to another. These errors can not be corrected with the application of a set of local correction rules.
- **Unimplemented features:** Our implementation currently lacks a few of the complex Python features such as pattern matching on list enumerate function and lambda functions.
- **Timeout:** In our experiments, we found less than 5% of the student attempts timed out (set as 4 minutes).

Benchmark	Median (LOC)	Total Attempts	Syntax Errors	Test Set	Correct	Incorrect Attempts	Generated Feedback	Average Time(in s)	Median Time(in s)
prodBySum-6.00	5	1056	16	1040	772	268	218 (81.3%)	2.49s	2.53s
oddTuples-6.00	6	2386	1040	1346	1002	344	185 (53.8%)	2.65s	2.54s
compDeriv-6.00	12	144	20	124	21	103	88 (85.4%)	12.95s	4.9s
evalPoly-6.00	10	144	23	121	108	13	6 (46.1%)	3.35s	3.01s
compBal-stdin-6.00	18	170	32	138	86	52	17 (32.7%)	29.57s	14.30s
compDeriv-6.00x	13	4146	1134	3012	2094	918	753 (82.1%)	12.42s	6.32s
evalPoly-6.00x	15	4698	1004	3694	3153	541	167 (30.9%)	4.78s	4.19s
oddTuples-6.00x	10	10985	5047	5938	4182	1756	860 (48.9%)	4.14s	3.77s
iterPower-6.00x	11	8982	3792	5190	2315	2875	1693 (58.9%)	3.58s	3.46s
recurPower-6.00x	10	8879	3395	5484	2546	2938	2271 (77.3%)	10.59s	5.88s
iterGCD-6.00x	12	6934	3732	3202	214	2988	2052 (68.7%)	17.13s	9.52s
hangman1-str-6.00x	13	2148	942	1206	855	351	171 (48.7%)	9.08s	6.43s
hangman2-str-6.00x	14	1746	410	1336	1118	218	98 (44.9%)	22.09s	18.98s
stock-market-I(C#)	20	52	11	41	19	22	16 (72.3%)	7.54s	5.23s
stock-market-II(C#)	24	51	8	43	19	24	14 (58.3%)	11.16s	10.28s
restaurant rush (C#)	15	124	38	86	20	66	41 (62.1%)	8.78s	8.19s

Table 1: The percentage of student attempts corrected and the time taken for correction for the benchmark problems.



Figure 36: The number of incorrect student submissions that require different number of corrections (in log scale).

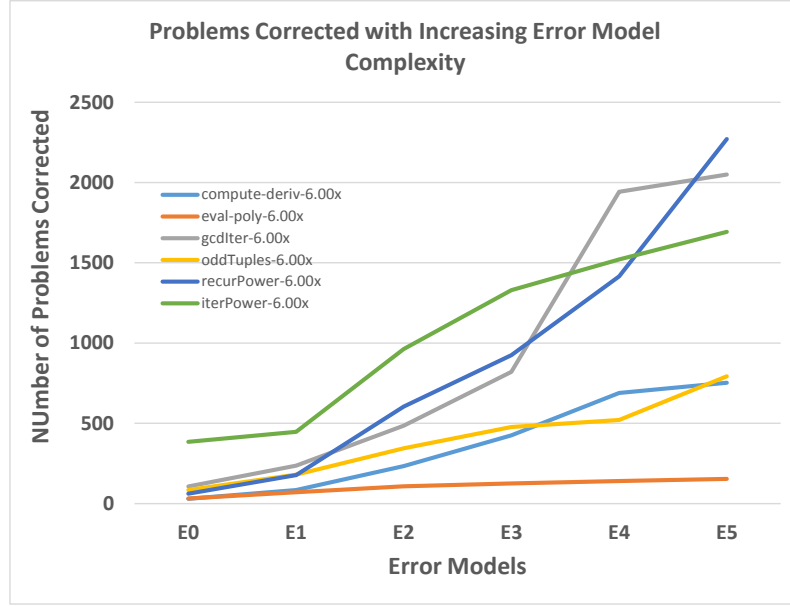


Figure 37: The number of incorrect student submissions corrected by addition of correction rules to the error models.

NUMBER OF CORRECTIONS The number of student submissions that require different number of corrections are shown in Figure 36 (on a logarithmic scale). We observe from the figure that a significant fraction of the problems require 3 and 4 coordinated corrections, and to provide feedback on such attempts, we need a technology like ours that can symbolically encode the outcome of different corrections on all input values.

REPETITIVE MISTAKES In this experiment, we check our hypothesis that students make similar mistakes while solving a given problem. The graph in Figure 37 shows the number of student attempts corrected as more rules are added to the error models of the benchmark problems. As can be seen from the figure, adding a single rule to the error model can lead to correction of hundreds of attempts. This validates our hypothesis that different students indeed make similar mistakes when solving a given problem.

GENERALIZATION OF ERROR MODELS In this experiment, we check the hypothesis that the correction rules generalize across problems of similar kind. The result of running the compute-deriv error model on other benchmark problems is shown in Figure 38. As expected, it does not perform as well as the problem-specific error models, but it still fixes a fraction of the incorrect attempts and can be useful as a good starting point to specialize the error model further by adding more problem-specific rules.

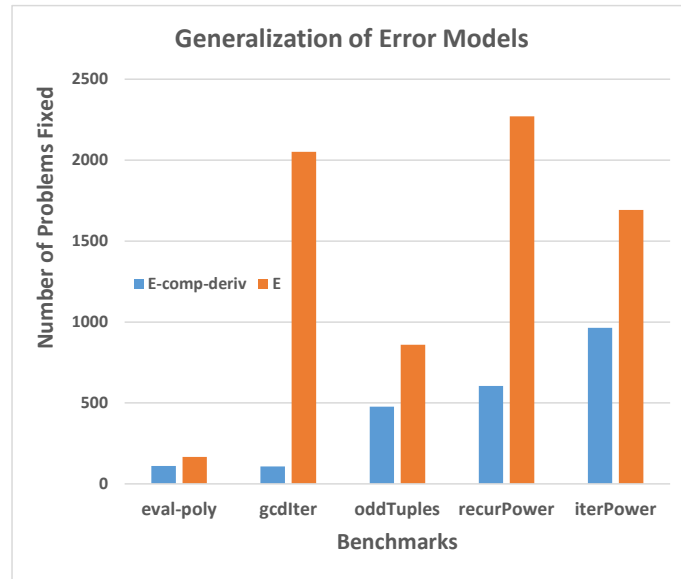


Figure 38: The performance of computeDeriv error model on other benchmark problems.

3.7 CAPABILITIES AND LIMITATIONS

AutoProf supports a fairly large subset of Python types and language features, and can currently provide feedback on a large fraction (64%) of student submissions in our benchmark set. In comparison to the traditional test-cases based feedback techniques that test the programs over a few dozens of test-cases, AutoProf typically performs the equivalence check over more than 10^6 inputs. Programs that print the output to console (e.g. compBal-stdin) pose an interesting challenge for test-cases based feedback tools. Since beginner students typically print some extra text and values in addition to the desired outputs, the traditional tools need to employ various heuristics to discard some of the output text to match the desired output. AutoProf lets instructors provide correction rules that can optionally drop some of the print expressions in the program, and then AutoProf finds the required print expressions to eliminate so that a student is not penalized much for printing additional values.

Now we briefly describe some of the limitations of AutoProf. One limitation is in providing feedback on student attempts that have big conceptual errors (see Section 3.6.3), which can not be fixed by application of a set of local rewrite rules. Correcting such programs typically requires a large global rewrite of the student solution, and providing feedback in such cases is an open question. Another limitation of AutoProf is that it does not take into account structural requirements in the problem statement since it focuses only on functional equivalence. For example, some of the assignments explicitly ask students to use bisection search or recursion, but AutoProf can

not distinguish between two functionally equivalent solutions, e.g. it can not distinguish between a bubble sort and a merge sort implementation of the sorting problem.

For some problems, the feedback generated by AutoProf is too low-level. For example, a suggestion provided by AutoProf in Figure 22(d) is to replace the expression `poly[e]==0` by `False`, whereas a higher level feedback would be a suggestion to remove the corresponding block inside the comparison. Deriving the high-level feedback from the low-level suggestions is mostly an engineering problem as it requires specializing the message based on the context of the correction.

The scalability of the technique also presents a limitation. For some problems that use large constant values, AutoProf currently replaces them with smaller teacher-provided constant values such that the correct program behavior is maintained. We also currently need to specify bounds for the input size, the number of loop unrollings and recursion depth as well as manually provide specialized error models for each problem. The problem of discovering these optimizations automatically by mining them from the large corpus of datasets is also an interesting research question. AutoProf also currently does not support some of the Python language features, most notably classes and objects, which are required for providing feedback on problems from later weeks of the class.

SEMANTIC STRING TRANSFORMATIONS IN FLASHFILL

The IT revolution over the past few decades has resulted in two significant advances: digitization of massive amounts of data and accessibility of computational devices to massive proportions of the population. 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 pros, marketing managers, underwriters, compliance officers, and even mail room clerks—they are not professional programmers, but they need to create small, *often one-off*, applications to support business functions [46].

Unfortunately, the programming experience since inception has focused mostly on serving the needs of a select class of few million skilled users. In particular, spreadsheet systems like Microsoft Excel allow sophisticated users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts using a variety of programming languages like Visual Basic, or .NET. Since end-users are not proficient in programming, they find it too difficult to write desired macros or scripts.

The combination of the above-mentioned technical trends and lack of a satisfactory solution has led to a marketplace of hundreds of advertisement-driven help-forums¹, some of which contain millions of posts from end-users soliciting help for scripts to manipulate data in their spreadsheets. The experts respond to these requests after some time. From an extensive case study of such spreadsheet help-forums, we observed the following two things.

- **Semantic String Transformations:** Several of the requested scripts/macros were for manipulating strings that need to be interpreted as more than a sequence of characters, e.g., as a column entry from some relational table, or as some standard data type such as number, date, time, or currency. We describe the systematic design of a semantic transformation language for manipulating such strings.
- **Input-Output Examples based Interaction Model:** End-users used input-output examples as the most common and natural way of expressing intent to experts on the other side of the help-forums. An expert provides a program/transformation that is consistent with

¹ <http://www.excelforum.com/>, <http://www.ozgrid.com/forum/>, <http://www.mrexcel.com/>, <http://www.exceltip.com/>

those examples. If the end-user is not happy with the result of the program on any other new input in the spreadsheet, the interaction is repeated with an extended set of input-output examples. This is the natural interface that our tool provides to the end-user. We describe the systematic design of an (inductive) synthesis algorithm that can learn desired scripts in our transformation language from very few examples.

We observe that most *semantic transformations* can be expressed as a combination of *lookup transformations* and *syntactic transformations*. We use this observation to present a systematic design of the transformation language for performing semantic string transformations. We first present an expressive language for lookup transformations and then extend it by adding syntactic transformations [49].

We also describe a systematic design of the synthesis algorithm for the semantic string transformation language, which can synthesize a set of semantic transformations that are consistent with the given set of input-output examples. We first describe a synthesis algorithm for the lookup transformation language L_t , and then extend it to a synthesis algorithm for the extension of L_t with syntactic transformations. We present the evaluation of our prototype on a large collection of benchmarks obtained from help-forums, books, mailing lists and Excel product team. The experimental results on our benchmark examples show that our algorithm is scalable and can learn desired transformations from very few examples.

4.1 MOTIVATING EXAMPLE

Consider the example posted by a user on an Excel help-forum shown in Figure 39.

Example 4.1.1. A shopkeeper wanted to compute the selling price of an item (Output) from its name (Input v_1) and selling date (Input v_2) using the MarkupRec and CostRec tables as shown in Figure 6. 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.

The user expresses her intent by giving a couple of examples (i.e., the first two rows). Our tool then automates the repetitive task (i.e., fills in the bold entries). We highlight below the key technical challenges involved.

EXPRESSIVE TRANSFORMATION LANGUAGE The program inferred by our system for automating the repetitive task involves both lookup and syntactic operations. In particular, note that we need lookup operations for (i) obtaining the markup percentage from an item name in MarkupRec table (Stroller \rightarrow 30%) and (ii) for obtaining the pur-

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			CostRec		
Id	Name	Markup	Id	Date	Price
S30	Stroller	30%	S30	12/2010	\$145.67
B56	Bib	45%	S30	11/2010	\$142.38
D32	Diapers	35%	B56	12/2010	\$3.56
W98	Wipes	40%	D32	1/2011	\$21.45
A46	Aspirator	30%	W98	4/2009	\$5.12
...	A46	2/2010	\$2.56
		

Figure 39: A transformation that requires syntactic manipulations on multiple lookup results.

chase price of item in CostRec table after performing a join operation between the two tables on the item Id column (Stroller,12/2010 \rightarrow \$145.67). Observe that the string 12/2010 used for performing the second lookup is obtained by performing a syntactic transformation (namely, a substring operation) on the input string 10/12/2010. After performing the lookups, we need a syntactic transformation (namely, a concatenate operation) to concatenate the lookup outputs with constant strings like +,0.,* in a particular order to generate the final output string. We present an expressive transformation language that combines lookup and syntactic transformations in a nested manner.

SUCCINCT REPRESENTATION OF LARGE NUMBER OF CONSISTENT TRANSFORMATIONS The number of expressions in the expressive lookup transformation language L_t that are consistent with a given example can potentially be very large. For example, for the first i-o example (Stroller, 10/12/2012 \rightarrow \$145.67+0.30*145.67), there are a large number of transformations that can generate the output string. In general, every substring in the output string can potentially be either a constant string, or a substring of an input string, or the result of a lookup operation. For example, the substring 30 in the output string can either be a constant string or a string obtained by performing a lookup operation in the MarkupRec or CostRec table. Some of the possible lookup transformations to obtain the string 30 include selecting the Markup column entry in the MarkupRec table where the item Name in the corresponding row is either one of the constant strings Stroller or Aspirator, or it matches the input string v_1 . Another

valid lookup transformation is to select the last two characters from the item Id column (S30) in MarkupRec or CostRec table with various ways to select the first row by constraining the item Name or Date columns respectively. We thus have a large number of possible transformations for each substring of the output string and explicit enumeration of all such choices becomes infeasible. A key technical contribution of our system is a data structure that can succinctly represent an exponential number of such transformations in polynomial space, and an algorithm that can compute such transformations in polynomial time. The key idea is to share common sub-expressions and compute/maintain choices for independent sub-expressions independently.

RANKING Our synthesis algorithm learns the set of all consistent transformations for each example and then intersects these sets to obtain the common transformations. The number of examples required to converge to the desired transformation may be large. To enable learning of the desired transformation from very few examples, we perform a ranking of these learned transformations that gives preference to transformations that are smaller (Occam’s razor principle) and that use fewer constants (to enforce generalization).

4.2 HYPOTHESIS SPACE: LOOKUP TRANSFORMATIONS

In this section, we present a lookup transformation language L_t that can model transformations that involve mapping a tuple of strings to another string using (possibly nested) lookup operations over a given database of relational tables. We present the syntax and semantics of L_t and then present few examples of L_t programs that can express tasks taken from Excel help forums.

4.2.1 Lookup Transformation Language L_t

$$\begin{array}{ll}
 \text{Expression } e_t & ::= v_i \\
 & \mid \text{Select}(C, T, b) \\
 \text{Boolean Cond } b & ::= p_1 \wedge \dots \wedge p_n \\
 \text{Predicate } p & ::= C = s \\
 & \mid C = e_t
 \end{array}$$

Figure 40: The syntax of lookup transformation language L_t .

The syntax of our expression language L_t for lookup transformations is defined in Figure 40. An expression e_t is either an input

string variable v_i , or a select expression denoted by $\text{Select}(C, T, b)$, where T is a relational table identifier and C is a column identifier of the table. The Boolean condition b is an ordered conjunction of predicates $p_1 \wedge \dots \wedge p_n$ where predicate p is an equality comparison between the content of some column of the table with a constant or an expression. We place a restriction on the columns present in these conditional predicates namely that these columns together constitute a *candidate key* of the table. The main idea behind this restriction is that we want to express queries that produce a single output as opposed to a set of outputs. The ordering of predicates results in an efficient *intersection* algorithm as described in Section. 4.3.2.

The language L_t has expected semantics. The expression $\text{Select}(C, T, b)$ returns the table entry $T[C, r]$, where r is the only row that satisfies condition b (as condition b is over candidate keys of the table). If there exists no row r whose columns satisfy b , the expression returns the empty string ϵ . The predicate $C = e_t$ is evaluated for row r by first evaluating the expression e_t and then comparing the returned string $\llbracket e_t \rrbracket \sigma$ with the string $T[C, r]$.

We now present few posts taken from Excel help-forums where the desired transformation can be represented in L_t .

Example 4.2.1. An Excel user was working on two tables: *CustData* and *Sale*. The user wanted to map names of customers to the selling price using address and street number columns as common columns between the two tables and posted the example shown in Figure 41 on a help-forum.

Input v_1	Output
Peter Shaw	110
Gary Lamb	225
Mike Henry	2015
Sean Riley	495

CustData		
Name	Addr	St
Sean Riley	432	15th
Peter Shaw	24	18th
Mike Henry	432	18th
Gary Lamb	104	12th
...

Sale			
Addr	St	Date	Price
24	18th	5/21	110
104	12th	5/23	225
432	18th	5/20	2015
432	15th	5/24	495
...

Figure 41: A lookup transformation that requires joining two tables.

The transformation can be expressed in L_t as $\text{Select}(\text{Price}, \text{Sale}, \text{Addr} = \text{Select}(\text{Addr}, \text{CustData}, \text{Name} = v_1) \wedge \text{St} = \text{Select}(\text{St}, \text{CustData}, \text{Name} = v_1))$.

Example 4.2.2. An Excel user was struggling with the task of copying the data from one table to another table based on a common column part-number. The first table Parts consisted of two columns PartNumber and Price of some items. The second table also consisted of the same two columns partNumber and Price but the price column was empty and the user wanted to fill this table using the Parts table. The user managed to perform the copying using PHP by importing data into tables, using scripts to join them and exporting it back to CSV. An expert later replied to the post with the macro : `=INDEX(C2:C4,MATCH(G3,B2:B4,0),1)`. In our framework, the user can perform the task by just providing one example as shown below. The strings in bold are the outputs produced by our framework.

Input v_1		Parts	
Output		PartNumber	Price
AXY-0031-UX	48	ISD-3234-PY	21
ISD-3234-PY	21	VWX-3023-HS	105
VWX-3023-HS	105	IIW-2010-QE	99
		AXY-0031-UX	48
	

The desired transformation can be expressed in the language L_t as: `Select(Price, Parts, PartNumber = v_1)`.

The surface syntax of L_t allows sharing of sub-expressions (which is the key principle used in data structure D_t described in Section 4.3.1). To appreciate this, consider the following example, which is also our running example in this section.

Example 4.2.3. Consider m tables T_1 to T_m , each containing three columns C_1 , C_2 , and C_3 with C_1 being the primary key. Suppose table T_i contains a row (s_i, s_{i+1}, s_{i+2}) . Now given an input-output example $s_1 \rightarrow s_m$, we want to compute all expressions in L_t that are consistent with it.

Consider the case of $m = 4$. Let $e \equiv \text{Select}(C_2, T_1, C_1 = v_1)$ that produces string s_2 . Two possible expressions in L_t to obtain output s_4 from input s_1 are: (i) $e_1 \equiv \text{Select}(C_3, T_2, C_1 = e)$ (corresponding to path $s_1 \rightarrow s_2 \rightarrow s_4$) and (ii) $e_2 \equiv \text{Select}(C_2, T_3, C_1 = \text{Select}(C_2, T_2, C_1 = e))$ (corresponding to path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$). The expressions e_1 and e_2 share the common sub-expression e which corresponds to obtaining the intermediate string s_2 . This sharing of sub-expression leads to a path-based sharing of set of select expressions to represent an exponential number of such expressions in polynomial space.

$$\begin{aligned}
\tilde{e}_t &:= (\tilde{\eta}, \eta^t, \text{Progs}) \\
&\quad \text{where } \text{Progs} : \tilde{\eta} \rightarrow 2^{\tilde{f}} \\
\tilde{f} &:= v_i \mid \text{Select}(C, T, B) \\
B &:= \{\tilde{b}_i\}_i \\
\tilde{b} &:= \tilde{p}_1 \wedge \dots \wedge \tilde{p}_n \\
\tilde{p} &:= C = s \mid C = \eta \\
&\quad \mid C = \{s, \eta\}
\end{aligned}$$

Figure 42: The syntax of the data structure D_t for succinctly representing a set of expressions from language L_t .

$$\begin{aligned}
\llbracket (\tilde{\eta}, \eta^t, \text{Progs}) \rrbracket &= \{e_t \mid e_t \in \llbracket \tilde{f} \rrbracket, \tilde{f} \in \text{Progs}[\eta^t]\} \\
\llbracket v_i \rrbracket &= \{v_i\} \\
\llbracket \text{Select}(C, T, \{\tilde{b}_i\}_i) \rrbracket &= \{\text{Select}(C, T, b) \mid b \in \llbracket \tilde{b}_i \rrbracket\} \\
\llbracket \tilde{p}_1 \wedge \dots \wedge \tilde{p}_n \rrbracket &= \{p_1 \wedge \dots \wedge p_n \mid p_j \in \llbracket \tilde{p}_j \rrbracket\} \\
\llbracket C = s \rrbracket &= \{C = s\} \\
\llbracket C = \eta \rrbracket &= \{C = e_t \mid e_t \in \llbracket \text{Progs}[\eta] \rrbracket\} \\
\llbracket C = \{s, \eta\} \rrbracket &= \llbracket C = s \rrbracket \cup \llbracket C = \eta \rrbracket
\end{aligned}$$

Figure 43: The semantics of the data structure D_t for succinctly representing a set of expressions from language L_t .

4.3 SYNTHESIS ALGORITHM

Since the language L_t is quite expressive, there are typically a large number of expressions that are consistent with a set of few input-output examples. We first present a data structure D_t to succinctly represent a large set of expressions in the language. We then present an efficient synthesis algorithm to learn a set of transformations in L_t from a set of input-output examples, such that each of the learned transformations when run on the given inputs produces the corresponding outputs.

4.3.1 Data Structure for Set of Expressions in L_t

The set of expressions in language L_t that are consistent with a given input-output example can be exponential in the number of *reachable* table entries. We represent this set succinctly using the data structure D_t , which is described in Figure 42. The data structure consists of a generalized expression \tilde{e}_t , generalized Boolean condition \tilde{b} , and generalized predicate \tilde{p} (which respectively denote a set of L_t expres-

sions, a set of Boolean conditions b , and a set of predicates p). The formal semantics $\llbracket \cdot \rrbracket$ of the data structure is described in Figure 43. The generalized expression \tilde{e}_t is represented using a tuple $(\tilde{\eta}, \eta^t, \text{Progs})$ where $\tilde{\eta}$ is a set of nodes containing a distinct target node η^t (representing the output string), and $\text{Progs} : \tilde{\eta} \rightarrow 2^{\tilde{f}}$ maps each node $\eta \in \tilde{\eta}$ to a set consisting of input variables v_i or generalized select expressions $\text{Select}(C, T, B)$. A generalized select expression takes a set of generalized Boolean conditions \tilde{b}_i as the last argument, where each \tilde{b}_i corresponds to some candidate key of table T . A generalized conditional \tilde{b} is a conjunction of generalized predicates \tilde{p}_i , where each \tilde{p}_i is an equality comparison of the j^{th} column of the corresponding candidate key with a constant string s or some node $\tilde{\eta}$ or both. There are two key aspects of this data structure which are explained below using some worst-case examples.

Use of intermediate nodes in $\tilde{\eta}$ for sharing: Consider the problem in Example 4.2.3. The set of all transformations in L_t that are consistent with the example $s_1 \rightarrow s_m$ can be represented succinctly using our data structure as: $\{\{\eta_1, \dots, \eta_m\}, \eta_m, \text{Progs}\}$, where $\text{Progs}[\eta_i] = \{\text{Select}(C_2, T_{i-1}, \{C_1 = \{s_{i-1}, \eta_{i-1}\}\}), \text{Select}(C_3, T_{i-2}, \{C_1 = \{s_{i-2}, \eta_{i-2}\}\})\}$, $\text{Progs}[\eta_1] = \{v_1\}$, and $\text{Progs}[\eta_2] = \{\text{Select}(C_2, T_1, \{C_1 = \{s_1, \eta_1\}\})\}$. The node η_i essentially corresponds to the string s_i . Figure 44 illustrates how the various nodes can be reached or computed from one another. Let $N(i)$ denote the number of expressions

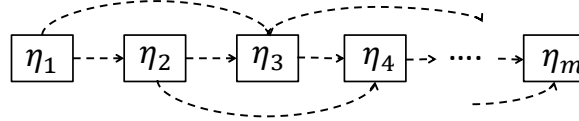


Figure 44: The reachability graph of nodes in Ex. 4.2.3.

represented succinctly by $\text{Progs}[\eta_i]$. We have $N(i) = 2 + N(i-1) + N(i-2)$, implying that $N(i) = \Theta(2^i)$. Observe how the data structure makes use of the set of nodes $\{\eta_1, \dots, \eta_m\}$ to succinctly represent $\Theta(2^m)$ transformations in $O(m)$ space.

Exploiting CNF form of boolean conditions: The second key aspect of our representation is exploiting the CNF form of boolean conditions to succinctly represent a huge set \tilde{b} of conditionals. Consider a table T with $n+1$ columns C_1, \dots, C_{n+1} , where the first n columns constitute a primary key of the table and the table contains an entry $(s_1, s_2, \dots, s_n, t)$. Consider the input-output example $(s_1, s_2, \dots, s_m) \rightarrow t$ with $s_1 = s_2 = \dots = s_{\max(m,n)}$. The number of transformations that are consistent with the given input-output example are $(m+1)^n$ because for indexing into each column C_i of the table, we have $m+1$ choices namely constant s_1 and the input string variables v_1, \dots, v_m . This huge set of transformations can be represented succinctly in $O(n+m)$ space using our data structure as

$(\{\eta_1, \eta_2\}, \eta_2, \text{Progs})$, where $\text{Progs}[\eta_1] = \{v_1, \dots, v_m\}$, and $\text{Progs}[\eta_2] = \{\text{Select}(C_{n+1}, T, \tilde{b})\}$, $\tilde{b} = \bigwedge_{i=1}^n (C_i = \{s_1, \eta_1\})$.

Theorem 4.3.1 (Properties of data structure D_t). (a) The number of transformations in L_t that are consistent with a given example may be exponential in the number of reachable entries and number of columns in a candidate key.
 (b) However, the data structure D_t can represent these potentially exponential number of transformations in polynomial size in number of reachable entries, number of candidate keys and number of columns in a candidate key.

Proof of (a) follows from the two examples discussed above, while proof of (b) follows from Theorem 4.3.2(a).

4.3.2 Synthesis Algorithm for L_t

```

GenerateStrt(σ: Input state, s: Output string)
1   $\tilde{\eta} := \emptyset$ ;  $\tilde{\eta}_{old} := \emptyset$ ; steps := 0;
2   $val := \emptyset$  //  $val : \eta \rightarrow T[C, r]$ 
3  foreach input variable  $v_i$ :
4    if  $((\eta := val^{-1}(\sigma(v_i))) = \perp)$ 
5      then {  $\eta := \text{NewNode}()$ ;  $\tilde{\eta} := \tilde{\eta} \cup \{\eta\}$ ;
6              $val(\eta) := \sigma(v_i)$ ;  $\text{Progs}[\eta] := \emptyset$ ; }
7     $\text{Progs}[\eta] := \text{Progs}[\eta] \cup \{v_i\}$ ;
8  while (steps++ ≤ k ∧  $\tilde{\eta}_{old} \neq \tilde{\eta}$ )
9     $\tilde{\eta}_{diff} := \tilde{\eta} - \tilde{\eta}_{old}$ ;  $\tilde{\eta}_{old} := \tilde{\eta}$ ;
10   foreach table T, col C, row r s.t.
11      $T[C, r] = val(\eta)$  for some  $\eta \in \tilde{\eta}_{diff}$ 
12      $B := \{ \bigwedge_{C' \in cKey} (C' = \{T[C', r], val^{-1}(T[C', r])\}) \mid$ 
13        $cKey \in \text{CandidateKeys}(T)\}$ ;
14     foreach column  $C'$  of table T s.t.  $C' \neq C$ :
15       if  $((\eta := val^{-1}(T[C', r])) = \perp)$ 
16       then {  $\eta := \text{NewNode}()$ ;  $\tilde{\eta} := \tilde{\eta} \cup \{\eta\}$ ;
17              $val(\eta) := T[C', r]$ ;  $\text{Progs}[\eta] := \emptyset$ ; }
18        $\text{Progs}[\eta] := \text{Progs}[\eta] \cup \{\text{Select}(C', T, B)\}$ ;
19   return  $(\tilde{\eta}, val^{-1}(s), \text{Progs})$ ;

```

Figure 45: The GenerateStr_t procedure for generating the set of all expressions (of depth at most k) in language L_t that are consistent with a given input-output example.

The inductive synthesis algorithm Synthesize for an expression language L learns the set of expressions in L (represented using data structure D) that are consistent with a given set of input-output ex-

$$\begin{aligned}
\text{Intersect}_t((\tilde{\eta}_1, \eta_1^t, \text{Progs}_1), (\tilde{\eta}_2, \eta_2^t, \text{Progs}_2)) &= (\tilde{\eta}_1 \times \tilde{\eta}_2, (\eta_1^t, \eta_2^t), \text{Progs}_{12}) \\
&\text{where } \text{Progs}_{12}[(\eta_1, \eta_2)] = \text{Intersect}_t(\text{Progs}_1[\eta_1], \text{Progs}_2[\eta_2]) \\
\text{Intersect}_t(v_i, v_i) &= v_i \\
\text{Intersect}_t(\text{Select}(C, T, B), \text{Select}(C, T, B')) &= \text{Select}(C, T, B'') \\
&\text{where } B'' = \text{Intersect}_t(B, B') \\
\text{Intersect}_t(\{\tilde{b}_i\}_i, \{\tilde{b}'_i\}_i) &= \{\text{Intersect}_t(\tilde{b}_i, \tilde{b}'_i)\}_i \\
\text{Intersect}_t(\{\tilde{p}_i\}_i, \{\tilde{p}'_i\}_i) &= \{\text{Intersect}_t(\tilde{p}_i, \tilde{p}'_i)\}_i \\
\text{Intersect}_t(C = s, C = s) &= C = s \\
\text{Intersect}_t(C = \eta_1, C = \eta_2) &= C = (\eta_1, \eta_2) \\
\text{Intersect}_t(C = \{s, \eta_1\}, C = \{s, \eta_2\}) &= C = \{s, (\eta_1, \eta_2)\} \\
\text{Intersect}_t(C = \{s_1, \eta_1\}, C = \{s_2, \eta_2\}) &= C = \{(\eta_1, \eta_2)\}, \text{ if } s_1 \neq s_2
\end{aligned}$$

Figure 46: The Intersect_t procedure for intersecting sets of expressions from language L_t . The Intersect_t procedure returns \emptyset in all other cases.

amples. Our synthesis algorithm consists of the following two procedures:

- The GenerateStr procedure for computing the set of all expressions (represented using data structure D) that are consistent with a given input-output example.
- The Intersect procedure for intersecting two sets of expressions (represented using data structure D). We describe Intersect procedure also using a set of rules.

Definition 4.3.1. (Soundness/ k -completeness of GenerateStr) Let $\tilde{e} = \text{GenerateStr}(\sigma, s)$. We say that GenerateStr procedure is *sound* if all expressions in \tilde{e} are consistent with the input-output example (σ, s) . We say that GenerateStr procedure is *k-complete* if \tilde{e} includes all expressions with at most k recursive sub-expressions that are consistent with the input-output example (σ, s) .

Definition 4.3.2. (Soundness/Completeness of Intersect) Let $\tilde{e}'' = \text{Intersect}(\tilde{e}, \tilde{e}')$. We say that Intersect is *sound* and *complete* iff \tilde{e}'' includes all expressions that belong to both \tilde{e} and \tilde{e}' .

The synthesis algorithm Synthesize involves invoking the GenerateStr procedure on each input-output example, and intersecting the results using the Intersect procedure as shown in Figure 47.

Procedure GenerateStr_t

The GenerateStr_t procedure, shown in Figure 45, operates by iteratively computing a set of nodes $\tilde{\eta}$ and updating two maps Progs and

```

Synthesize(( $\sigma_1, s_1$ ), ..., ( $\sigma_n, s_n$ ))
1   $P := \text{GenerateStr}(\sigma_1, s_1);$ 
2  for  $i = 2$  to  $n$ :
3     $P' := \text{GenerateStr}(\sigma_i, s_i); P := \text{Intersect}(P, P');$ 
4  return  $P$ ;

```

Figure 47: The general synthesis algorithm for version-space algebra based synthesis algorithm.

val in the loop at Line 8. Each node $\eta \in \tilde{\eta}$ represents a string $\text{val}(\eta)$ that is present in some table entry. The inverse map $\text{val}^{-1}(a)$ returns the node corresponding to string a or \emptyset if no such node exist. The map Progs associates every node η to a set of expressions (of depth at most steps), each of which evaluates to string $\text{val}(\eta)$ on the input state σ . The key idea of the loop at Line 8 is to perform 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 previous step, with the base case being the values of the input string variables.

Each iteration of the loop at Line 8 results in consideration of expressions whose depth is one larger than the set of expressions considered in the previous step. The depth of the expressions in language L_t can be as much as the total number of entries in all of the relational tables combined. Since we have not observed any intended transformation that requires self-joins, we limit the depth consideration to a parameter k whose value we set to be equal to the number of relational tables present in the spreadsheet. One might be tempted to use the predicate $(s \in \tilde{\eta} \vee \tilde{\eta}_{\text{old}} = \tilde{\eta})$ as a termination condition for the loop. However, this has two issues. The first issue is that it may happen co-incidentally that the output string s is computable by a transformation of depth smaller than the depth of the intended transformation on a given example, and in that case we would fail to discover the correct transformation. The other major issue is that it might also happen that the intended transformation does not belong to the language L_t , in which case the search would fail, but only after consideration of all expressions whose depth is as large as the total number of entries in all relational tables combined together.

The generalized boolean condition B is computed to be the set of all boolean conditions that uniquely identify row r in table T (Line 11). It considers the set of candidate keys of table T and for each column C' in a candidate key it learns the generalized predicate: $C' = \{T[C', r], \text{val}^{-1}[T[C', r]]\}$.

During the reachability computation, a node η can be reached through multiple paths and therefore the set of expressions associated with the node ($\text{Progs}[\eta]$) needs to be updated accordingly. When a node is revisited, the algorithm computes the `Select` expression with updated set of Boolean conditions B and adds it to the set (Line 16).

We now briefly describe how the GenerateStr_t procedure computes the set of expressions for each node in Example 4.2.3. It first creates a node η_1 such that $\text{Progs}[\eta_1] = \{v_1\}$, $\text{val}(\eta_1) = s_1$ and the frontier of reachable nodes is set as $\tilde{\eta}_{\text{diff}} = \{\eta_1\}$. We use node η_i to denote the node corresponding to string s_i such that $\text{val}(\eta_i) = s_i$. The algorithm then finds that the table entry $T_1[C_1, 1]$ is reachable from node η_1 with the generalized Boolean condition $B = \{C_1 = \{s_1, v_1\}\}$. The algorithm then makes the other column entries in the row, namely $T_1[C_2, 1]$ and $T_1[C_3, 1]$ also reachable and creates nodes η_2 and η_3 corresponding to them, and then sets $\text{Progs}[\eta_2] = \{\text{Select}(C_2, T_1, B)\}$ and $\text{Progs}[\eta_3] = \{\text{Select}(C_3, T_1, B)\}$. In the next iteration of loop, the frontier of reachable set is updated to $\tilde{\eta}_{\text{diff}} = \{\eta_2, \eta_3\}$ and the nodes that are reachable from this set are next computed. The algorithm finds that table entry $T_2[C_1, 1]$ is reachable from node η_2 and thereby makes nodes η_3 and η_4 reachable as well with corresponding Select expressions. Similarly, the nodes η_4 and η_5 become reachable from η_3 . In this manner, the algorithm keeps computing the set of reachable table entries iteratively until k iterations, where k is set to the number of relational tables n .

Procedure Intersect_t

The Intersect_t procedure takes two sets of expressions in L_t as input and computes the set of expressions that are common to both the sets. (Both input and output sets are represented using the data structure D_t .) Figure 46 describes the Intersect_t procedure for intersecting the sets of L_t expressions using a set of rules that are pattern matched for execution. For intersecting two expressions $(\tilde{\eta}_1, \eta_1^t, \text{Progs}_1)$ and $(\tilde{\eta}_2, \eta_2^t, \text{Progs}_2)$, we take the cross product of the set of nodes to get the new set of nodes $\tilde{\eta}_{12} = (\tilde{\eta}_1 \times \tilde{\eta}_2)$ with the target node (η_1^t, η_2^t) , and compute the new Progs_{12} map for each node $(\eta_1, \eta_2) \in \tilde{\eta}_{12}$ by intersecting their corresponding maps $\text{Progs}_1[\eta_1]$ and $\text{Progs}_2[\eta_2]$ respectively. The intersect rule for two select expressions requires the column name and table id to be the same and intersects the conditionals recursively. The candidate keys \tilde{b}_i as well as each column conditional \tilde{p} in a candidate key are intersected individually maintaining their corresponding orderings.

Theorem 4.3.2 (Synthesis Algorithm Properties). (a) The procedure GenerateStr_t is sound and k -complete. The computational complexity of GenerateStr_t procedure (and hence the size of the data structure constructed by it) is $O(t^2 p m)$ where t is the number of reachable strings in k iterations, p is the maximum number of candidate keys in any table and m is the maximum number of columns in any candidate key.

(b) The procedure Intersect_t is sound and complete. The computational complexity of Intersect_t procedure (and hence the size of

the data structure returned by it) is $O(d^2)$, where d is the size of the input data structures.

Proof. Proof of (a) follows from the following two stronger inductive invariants associated with the loop at line 8. (i) The set $\{\text{val}(\eta) \mid \eta \in \tilde{\eta}\}$ denotes the set of all string values that can be computed starting from the values of input variables v_i and using some expression of depth at most steps. (ii) $\text{Progs}[\eta]$ is the set of all expressions of depth at most steps that evaluate to $\text{val}(\eta)$ from σ . Let us suppose we have t number of reachable table entries computed by `GenerateStr` procedure. For each reachable table entry, we need to compute its generalized conditional B for which we go through each of its primary key and each column in that primary key. This gives us a complexity of $O(t \cdot p \cdot m)$. Now we can visit each reachable string from any of the other strings in worst case t times, which gives us the complexity of $O(t^2 \cdot p \cdot m)$. We can perform the check on line 10 in $O(1)$ time hashing all table entries. Proof of (b) follows from the semantics of \tilde{e}_t as defined in Figure 43. The complexity of Intersect_t follows from cross-product intersection based algorithm in Figure 46. Note that except intersecting generalized conditionals takes linear time as we maintain the ordering of primary keys and their constituent columns. \square

4.3.3 Ranking

We define a partial order between expressions in L_t that we use for ranking of these expressions. We prefer expressions of smaller depth (fewer nested chains of `Select` expressions). We prefer lookup expressions that use distinct tables for join queries (the most common scenario for end-users) as opposed to expressions involving self-joins. We prefer conditionals that consist of fewer predicates and prefer predicates that involve comparing columns with other table entries or input variables (as opposed to comparing columns with constant strings).

4.4 USER INTERACTION MODEL

The user expresses the intended task using few input-output examples. The synthesizer based on the above formalism then generates a ranked set of transformations that are consistent with those examples. We describe below some interaction techniques for automating the desired task or for generating a reusable transformation.

The user can run the top-ranked synthesized transformation on other inputs in the spreadsheet and check the generated outputs. If any output is incorrect, the user can fix it by providing the correct output and the synthesizer can repeat the learning process with the additional example that the user provided as a fix. Requiring the user to check the results of the synthesizer, especially on a large spread-

sheet, can be cumbersome. To enable easier interaction, the synthesizer can run *all* transformations on each new input to generate a set of corresponding outputs for that input. The synthesizer can then highlight those inputs (for user inspection) whose corresponding output set contains at least two outputs. The user can then focus their inspection on the highlighted inputs. Our prototype, implemented as an Excel add-in, supports this interaction model (which is also used in [49]).

On the other hand, if the user wishes to learn a reusable script, then the synthesizer may present the set of synthesized transformations to the user. Either the top-k transformations can be shown, or the synthesizer can walk the user through the data structure that succinctly represents all consistent transformations and let the user select the desired one. The transformations can be shown using the surface syntax, or can be paraphrased in a natural language. The differences between different transformations can also be explained by synthesizing a *distinguishing input* on which the transformations behave differently [65]. After receiving the correct output from the user on the distinguishing input, the synthesizer can repeat the learning process with this additional example.

4.5 SEMANTIC TRANSFORMATIONS

We now present an extension of the lookup transformation language L_t (described in Section. 4.2) with a syntactic transformation language L_s (from [49]). This extended language L_u , also referred to as semantic transformation language, adds two key capabilities to L_t : (i) It allows for lookup transformations that involve performing syntactic manipulations (such as substring, concatenation, etc.) on strings before using them to perform lookups, and (ii) It allows for performing syntactic manipulations on lookup outputs (which can then be used for performing further lookups or for generating the output string). This extension, as we show in Section. 4.6, also enables us to model transformations on strings representing standard data types such as date, time, etc. We first describe a syntactic transformation language.

SYNTACTIC TRANSFORMATION LANGUAGE L_s Gulwani [49] introduced an expression language for performing syntactic string transformations. We reproduce here a small subset of (the rules of) that language and call it L_s (with e_s being the top-level symbol).

$$\begin{aligned}
 e_s &:= \text{Concatenate}(f_1, \dots, f_n) \mid f \\
 \text{Atomic expr } f &:= \text{ConstStr}(s) \mid v_i \mid \text{SubStr}(v_i, p_1, p_2) \\
 \text{Position } p &:= k \mid \text{pos}(r_1, r_2, c) \\
 \text{Integer expr } c &:= k \mid k_1 w + k_2 \\
 \text{Regular expr } r &:= \epsilon \mid f_i \mid \text{TokenSeq}(f_1, \dots, f_n)
 \end{aligned}$$

The formal semantics of L_s can be found in [49]. For completeness, we briefly describe some key aspects of this language. The top-level expression e_s is either an atomic expression f or is obtained by concatenating atomic expressions f_1, \dots, f_n using the Concatenate constructor. Each atomic expression f can either be a constant string $\text{ConstStr}(s)$, an input string variable v_i , or a substring of some input string v_i . The substring expression $\text{SubStr}(v_i, p_1, p_2)$ 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.) $\text{SubStr}(v_i, p_1, p_2)$ is the substring of string v_i in between positions p_1 and p_2 . A position expression represented by a non-negative constant k denotes the k^{th} position in the string. For a negative constant k , it denotes the $(\ell + 1 + k)^{\text{th}}$ position in the string, where $\ell = \text{Length}(s)$. A position expression $\text{pos}(r_1, r_2, c)$ evaluates to a position t in the subject string s such that regular expression r_1 matches some suffix of $s[0 : t]$, and r_2 matches some prefix of $s[t : \ell]$, where $\ell = \text{Length}(s)$. Furthermore, if c is positive (negative), then t is the $|c|^{\text{th}}$ such match starting from the left side (right side). We use the expression $s[t_1 : t_2]$ to denote the substring of s between positions t_1 and t_2 . We use notation $\text{SubStr2}(v_i, r, c)$ as an abbreviation to denote the c^{th} occurrence of r in v_i , i.e., $\text{SubStr}(v_i, \text{pos}(\epsilon, r, c), \text{pos}(r, \epsilon, c))$.

A regular expression r is either a token fi , ϵ (which matches the empty string), or a token sequence $\text{TokenSeq}(fi_1, \dots, fi_n)$. The tokens fi range over some finite (but easily extensible) set and typically correspond to character classes and special characters. For example, tokens `UpperTok`, `NumTok`, and `AlphTok` 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. `SlashTok` matches the slash character. Special tokens `StartTok` and `EndTok` match the beginning and end of a string respectively.

Example 4.5.1. An Excel user wanted to transform names to a format where the last name is followed by the initial letter of the first name, e.g., “Alan Turing” \rightarrow “Turing A”. An expression in L_s that can perform this transformation is: $\text{Concatenate}(f_1, \text{ConstStr}(" "), f_2)$ where $f_1 \equiv \text{SubStr2}(v_1, \text{AlphTok}, 2)$ and $f_2 \equiv \text{SubStr2}(v_1, \text{UpperTok}, 1)$. This expression constructs the output string by concatenating the 2nd word of input string, the constant string whitespace, and the 1st capital letter in input string.

We now present the extended language L_u .

4.5.1 Semantic Transformation Language L_u

Let R_t and R_s denote the set of grammar rules of languages L_t and L_s respectively. We subscript each non-terminal in the two languages with t and s for disambiguating the names of non-terminals in the extended language. For example, f_s denotes the atomic expression f of the syntactic transformation language L_s . The expression grammar of the extended language L_u consists of rules $R_t \cup R_s$ in which the following rules are modified (with modifications shown in **bold**), and with e_s as the top-level symbol.

$$\begin{aligned} \text{Atomic expr } f_s &:= \text{ConstStr}(s) \mid \mathbf{e_t} \mid \text{SubStr}(\mathbf{e_t}, p_{s_1}, p_{s_2}) \\ \text{Predicate } p_t &:= C = s \mid C = \mathbf{e_s} \\ e_s &:= \text{Concatenate}(f_{s_1}, \dots, f_{s_n}) \mid f_s \end{aligned}$$

The top-level expression e_s of the extended language is either an atomic expression f_s or a Concatenate operation on a sequence of atomic expressions f_{s_i} , as before. However, the atomic expression f_s is updated to consist of a lookup expression e_t or its substring $\text{SubStr}(e_t, p_{s_1}, p_{s_2})$ (as opposed to only an input variable v_i or its substring). This lets the language model transformations that perform syntactic manipulations over table lookup outputs. The other modification is in the predicate expression p_t of language L_t , where we modify the conditional expression $C = e_t$ to $C = e_s$. This enables the language to model lookup transformations that perform syntactic manipulations on strings before performing the lookup. The updated rules have expected semantics and can be defined in a similar fashion as the semantics of rules in L_t and L_s . We now illustrate the expressiveness of the extended language using few examples.

The transformation in Example 4.1.1 can be represented in L_u as:
 $\text{Concatenate}(f_1, \text{ConstStr}("+0."), f_2, \text{ConstStr}("*"), f_3),$
 $f_1 \equiv \text{Select}(\text{Price}, \text{CostRec}, \text{Id} = f_4 \wedge \text{Date} = f_5),$
 $f_4 \equiv \text{Select}(\text{Id}, \text{MarkupRec}, \text{Name} = v_1),$
 $f_5 \equiv \text{SubStr}(v_2, \text{pos}(\text{SlashTok}, \epsilon, 1), \text{pos}(\text{EndTok}, \epsilon, 1)),$
 $f_2 \equiv \text{SubStr2}(f_6, \text{NumTok}, 1), \quad f_3 \equiv \text{SubStr2}(f_1, \text{DecNumTok}, 1),$
 $f_6 \equiv \text{Select}(\text{Markup}, \text{MarkupRec}, \text{Name} = v_1).$

The expression f_4 looks up the Id of the item in v_1 from the MarkupRec table and expression f_5 generates a substring of the date in v_2 , which are then used to look up 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 \$

sign. Finally, the top-level expression concatenates f_1 , f_2 , and f_3 with constant strings “+0.” and “*”.

Example 4.5.2. Indexing with concatenated strings:

A bike merchant maintained an inventory of BikePrices table, and wanted to compute the price quote table by performing lookup of bike Price after concatenating the bike name (v_1) and the engine cc value (v_2) as shown in Figure 48.

Input v_1	Input v_2	Output
Honda	125	11,500
Ducati	100	10,000
Honda	250	19,000
Ducati	250	18,000

BikePrices	
Bike	Price
Ducati100	10,000
Ducati125	12,500
Ducati250	18,000
Honda125	11,500
Honda250	19,000
...	...

Figure 48: A lookup transformation that requires concatenating input strings before performing selection from a table.

The desired transformation can be expressed in L_u as:
 $\text{Select}(\text{Price}, \text{BikePrices}, \text{Bike} = e_s)$ where $e_s = \text{Concatenate}(v_1, v_2)$.
 The expression e_s concatenates the two input string variables v_1 and v_2 , which is then used to perform the lookup in the BikePrices table.

Example 4.5.3. Concatenating table outputs: A user had a series of three company codes in a column and wanted to expand them into the corresponding series of company names using a table Comp that mapped company codes to the company names as shown in Figure 49.

Input v_1	Output
c4 c3 c1	Facebook Apple Microsoft
c2 c5 c6	Google IBM Xerox
c1 c5 c4	Microsoft IBM Facebook
c2 c3 c4	Google Apple Facebook

Comp	
Id	Name
c1	Microsoft
c2	Google
c3	Apple
c4	Facebook
c5	IBM
c6	Xerox
...	...

Figure 49: A nested syntactic and lookup transformation. It requires concatenating results of multiple lookup transformations, each of which involves a selection operation that indexes on some substring of the input.

This transformation is expressed in L_u as:
 $\text{Concatenate}(f_1, \text{ConstStr}(" "), f_2, \text{ConstStr}(" "), f_3)$, where
 $f_1 \equiv \text{Select}(\text{Name}, \text{Comp}, \text{Id} = \text{SubStr2}(v_1, \text{AlphTok}, 1))$,
 $f_2 \equiv \text{Select}(\text{Name}, \text{Comp}, \text{Id} = \text{SubStr2}(v_1, \text{AlphTok}, 2))$, and
 $f_3 \equiv \text{Select}(\text{Name}, \text{Comp}, \text{Id} = \text{SubStr2}(v_1, \text{AlphTok}, 3))$. The expressions f_1 , f_2 , and f_3 extract the first, second, and third words from the input string respectively, which are then used for performing the table lookups and the results are concatenated with whitespaces to obtain the output string.

4.5.2 Synthesis Algorithm

Data Structure for Set of Expressions in L_u

Let \tilde{R}_t and \tilde{R}_s denote the set of grammar rules for the data structures that represent set of expressions in languages L_t and L_s respectively (See [49] for description of \tilde{R}_s). We construct the grammar rules for the data structure that represents set of expressions in the extended language L_u by taking the union of the two rules $\tilde{R}_t \cup \tilde{R}_s$ and modifying some rules as follows:

$$\begin{aligned}\tilde{f}_s &:= \text{ConstStr}(s) \mid \tilde{e}_t \mid \text{SubStr}(\tilde{e}_t, \tilde{p}_{s_1}, \tilde{p}_{s_2}) \\ \tilde{p}_t &:= C = s \mid C = \tilde{e}_s \\ \tilde{e}_s &:= \text{Dag}(\tilde{\alpha}, \alpha^s, \alpha^t, \tilde{\xi}, W) \mid \tilde{f}_s, \text{ where } W : \tilde{\xi} \rightarrow 2^{f_s}\end{aligned}$$

The most interesting aspect of this data structure is the $\text{Dag}(\tilde{\alpha}, \alpha^s, \alpha^t, \tilde{\xi}, W)$ construct, which succinctly represents a set of Concatenate expressions in L_u using a dag (directed acyclic graph), where $\tilde{\alpha}$ is a set of nodes containing two distinct source and target nodes α^s and α^t , $\tilde{\xi}$ is a set of edges over nodes in $\tilde{\alpha}$ that induces a dag, and W maps each edge in $\tilde{\xi}$ to a set of atomic expressions. The semantics $\llbracket \cdot \rrbracket$ of the Dag constructor is:

$$\begin{aligned}\llbracket \text{Dag}(\tilde{\alpha}, \alpha^s, \alpha^t, \tilde{\xi}, W) \rrbracket &= \{ \text{Concatenate}(f_{s_1}, \dots, f_{s_n}) \mid \\ f_{s_i} &\in \llbracket W(\tilde{\xi}_i) \rrbracket, \xi_1, \dots, \xi_n \in \tilde{\xi} \text{ form a path between } \alpha^s \text{ and } \alpha^t \}\end{aligned}$$

The set of all Concatenate expressions represented by the Dag constructor includes exactly those whose ordered arguments belong to the corresponding edges on any path from α^s to α^t . This dag representation is similar to the representation of string expressions in [49]. However, in our case, the edges of the dag consist of more sophisticated (substrings of) lookup expressions, whose predicates can in turn be represented using nested-dags.

Consider Example 4.5.3, where the input string is “c4 c3 c1” and the output string is “Facebook Apple Microsoft” (of length 24). The dag G that represents the set of all transformations consistent with this input-output pair is shown in Figure 50. For better readability,

we only show some of the relevant nodes and edges of the dag G . The edge from node 0 to node 8 corresponds to all expressions \tilde{e}_1 that generate the string Facebook. One of the lookup transformations, $\text{Select}(\text{Name}, \text{Comp}, \text{Id} = \tilde{f}_1)$, in \tilde{e}_1 requires syntactic transformations \tilde{f}_1 to extract substring $c1$ from the input string, where \tilde{f}_1 is itself represented as a nested-dag as shown in the figure. The edges for expressions \tilde{e}_3 and \tilde{e}_5 also consist of similar nested-dags.

Theorem 4.5.1 (Properties of data structure D_u). (a) The number of transformations in L_u that are consistent with a given input-output example may be exponential in the number of reachable entries, the number of columns in a primary key, and the length of the largest reachable string.

(b) However, the data structure D_u can represent these potentially exponential number of transformations in size polynomial in the number of reachable entries, the number of primary keys, the number of columns in a primary key, and the length of the largest reachable string.

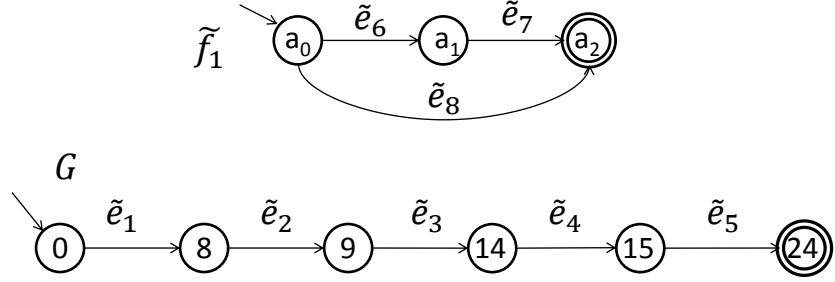
Proof. (a) Proof of number of transformations being exponential in the number of reachable entries and the number of columns in a primary key follows from Theorem 4.3.1(a). The number of transformations can also be exponential in the length of the largest reachable string as we are using GenerateStr_s procedure for checking reachability which has this worst case complexity. (b) We show that the size of the data structure generated is $O(t^2 p m \ell^2)$ in Theorem 4.5.2(a). \square

4.5.3 Synthesis Algorithm for L_u

Procedure GenerateStr_u

Recall that the GenerateStr_t procedure for language L_t (Section. 4.3.2) performs reachability on table entries based on exact string matches ($T[C, r] = \eta$). The key idea in case of the language L_u is to perform a more relaxed reachability on table entries taking into account the possibility of performing syntactic manipulations on previously reachable strings.

We first define a $\text{GenerateStr}'_t$ procedure by making two modifications to the GenerateStr_t procedure. First, we replace the condition “ $T[C, r] = \text{val}(\eta)$ ” in GenerateStr_t (Line 10 in Figure 45) by the condition “ $\text{GenerateStr}_s(\sigma \cup \tilde{\eta}, T[C, r])$ contains any expression that uses a variable from $\sigma \cup \tilde{\eta}$ ”. We use the notation $\sigma \cup \tilde{\eta}$ to denote a state that extends σ and maps η to $\text{val}(\eta)$ for all $\eta \in \tilde{\eta}$. The $\text{GenerateStr}'_t$ procedure marks a table entry as reachable if it can be computed using the GenerateStr_s procedure (from [49]) on previously reachable strings. The GenerateStr_s procedure can perform concatenation of constant strings and substrings of previously reachable strings ($\tilde{\eta}$). We add an



$$\begin{aligned} \tilde{f}_1 &\equiv \text{Dag}(\{a_0, a_1, a_2\}, a_0, a_2, \{\langle a_0, a_1 \rangle, \langle a_1, a_2 \rangle, \langle a_0, a_2 \rangle\}, W_2) \\ \text{where } W_2(\langle a_0, a_1 \rangle) &= \tilde{e}_6, W_2(\langle a_1, a_2 \rangle) = \tilde{e}_7, W_2(\langle a_0, a_2 \rangle) = \tilde{e}_8 \end{aligned}$$

$$\begin{aligned} G &\equiv \text{Dag}(\{0, \dots, 24\}, 0, 24, \{\langle i, j \rangle \mid 0 \leq i < j \leq 24\}, W_1) \\ \text{where } W_1(\langle 0, 8 \rangle) &= \tilde{e}_1, W_2(\langle 8, 9 \rangle) = \tilde{e}_2, W_2(\langle 9, 14 \rangle) = \tilde{e}_3, \dots \end{aligned}$$

$$\begin{aligned} \tilde{e}_1 &\equiv \{\text{Select}(\text{Name}, \text{Comp}, \text{Id} = \tilde{f}_1), \text{ConstStr}(\text{"Facebook"}), \dots\} \\ \tilde{e}_2 &\equiv \{\text{ConstStr}(\text{" "}), \dots\} \\ \tilde{e}_3 &\equiv \{\text{Select}(\text{Name}, \text{Comp}, \text{Id} = \tilde{f}_2), \text{ConstStr}(\text{"Apple"}), \dots\} \\ \tilde{e}_4 &\equiv \{\text{ConstStr}(\text{" "}), \dots\} \\ \tilde{e}_5 &\equiv \{\text{Select}(\text{Name}, \text{Comp}, \text{Id} = \tilde{f}_3), \text{ConstStr}(\text{"Microsoft"}), \dots\} \\ \tilde{e}_6 &\equiv \{\text{ConstStr}(\text{"c"}), \dots\} \\ \tilde{e}_7 &\equiv \{\text{ConstStr}(\text{"l"}), \dots\} \\ \tilde{e}_8 &\equiv \{\text{SubStr2}(v_1, \text{AlphTok}, 1), \dots\} \end{aligned}$$

Figure 50: A partial Dag representation of the set of expressions in Example 4.5.3.

additional check that the expressions synthesized by GenerateStr_s contains a variable from $\sigma \cup \tilde{\eta}$ to avoid expressions containing only constant string expressions. For our experiments, we enforce an even stronger restriction that there exists a string $\eta \in (\sigma \cup \tilde{\eta})$ such that either $T[C, r]$ is substring of η or η is a substring of $T[C, r]$. This provides efficiency without any practical loss of precision. The second modification is in Line 11 in Figure 45, where we replace the generalized predicate with $C' = \{\text{GenerateStr}_s(\sigma \cup \tilde{\eta}, T[C', r])\}$.

The GenerateStr_u procedure for the extended language L_u can now be defined as:

```

GenerateStru( $\sigma$ : Input state,  $s$ : Output string)
1  ( $\tilde{\eta}, \eta^t, \text{Progs}$ ) = GenerateStr't( $\sigma, s$ );
2  return GenerateStrs( $\sigma \cup \tilde{\eta}, s$ );

```

The GenerateStr_u procedure first constructs the set of all reachable table entries ($\tilde{\eta}$) from the set of input strings in σ using the $\text{GenerateStr}'_t$ procedure. It then uses the GenerateStr_s procedure to construct the Dag for generating the output string s from the set of strings that includes values of input variables (in state σ) as well as the reachable table entries (represented by $\tilde{\eta}$).

Consider the first input-output pair (“c4 c3 c1”, “Facebook Apple Microsoft”) in Example 4.5.3. The GenerateStr_u algorithm first uses the $\text{GenerateStr}'_t$ procedure to compute the set of reachable table entries. The $\text{GenerateStr}'_t$ procedure finds that the table entry $T[\text{Id}, 4] = \text{c4}$ is reachable from the input string “c4 c3 c1” as there exists an expression $\text{SubStr2}(v_1, \text{AlphTok}, 1)$ that can generate the string c4. It then adds the string Facebook from row 4 to the reachable set $\tilde{\eta}$. Similarly, table entries c3, Apple, c1, and Microsoft are also added to $\tilde{\eta}$. It then uses the GenerateStr_s procedure to construct a dag for generating the output string “Facebook Apple Microsoft” from the set {“c4 c3 c1”, c1, c2, c3, Facebook, Apple, Microsoft} ($\sigma \cup \tilde{\eta}$). It first creates a Dag of 25 nodes ($\tilde{\alpha} = \{0, \dots, 24\}$) with 0 as the source node and 24 as the target node. The algorithm then assigns a set of expressions to each edge $\langle i, j \rangle$ that can generate the substring $s[i, j]$. For example, the algorithm adds the expression that selects the string Facebook from $\sigma \cup \tilde{\eta}$ to the edge $\langle 0, 8 \rangle$; the expression in turn corresponds to a lookup transformation with nested sub-dags as shown in Figure 50. Similarly, it adds the expression \tilde{e}_2 that generates a whitespace to the edge $\langle 8, 9 \rangle$, and so on.

Procedure Intersect_u

The Intersect_u procedure for the extended language L_u consists of the union of rules of Intersect_t and Intersect_s procedures along with the following four additional rules.

$$\begin{aligned} \text{Intersect}_u(\tilde{e}_t, \tilde{e}'_t) &= \text{Intersect}_t(\tilde{e}_t, \tilde{e}'_t) \\ \text{Intersect}_u(C = \tilde{e}_s, C = \tilde{e}'_s) &= (C = \text{Intersect}_s(\tilde{e}_s, \tilde{e}'_s)) \\ \text{Intersect}_u(\text{SubStr}(\tilde{e}_t, \tilde{p}_{s_1}, \tilde{p}_{s_2}), \text{SubStr}(\tilde{e}'_t, \tilde{p}'_{s_1}, \tilde{p}'_{s_2})) &= \\ \text{SubStr}(\text{Intersect}_t(\tilde{e}_t, \tilde{e}'_t), \text{IntersectPos}(\tilde{p}_{s_1}, \tilde{p}'_{s_1}), & \\ \text{IntersectPos}(\tilde{p}_{s_2}, \tilde{p}'_{s_2})) & \end{aligned}$$

$$\begin{aligned} \text{Intersect}_u(\text{Dag}(\tilde{\alpha}_1, \alpha_1^s, \alpha_1^t, \tilde{\xi}_1, W_1), \text{Dag}(\tilde{\alpha}_2, \alpha_2^s, \alpha_2^t, \tilde{\xi}_2, W_2)) \\ = \text{Dag}(\tilde{\alpha}_1 \times \tilde{\alpha}_2, (\alpha_1^s, \alpha_2^s), (\alpha_1^t, \alpha_2^t), \tilde{\xi}_{12}, W_{12}), \text{ where} \\ \tilde{\xi}_{12} = \{ \langle (\alpha_1, \alpha_2), (\alpha'_1, \alpha'_2) \rangle \mid \langle \alpha_1, \alpha'_1 \rangle \in \tilde{\xi}_1, \langle \alpha_2, \alpha'_2 \rangle \in \tilde{\xi}_2 \} \\ \text{and } W_{12}(\langle (\alpha_1, \alpha_2), (\alpha'_1, \alpha'_2) \rangle) = \{ \text{Intersect}_s(f, f') \mid \\ f \in W_1(\langle \alpha_1, \alpha'_1 \rangle), f' \in W_2(\langle \alpha_2, \alpha'_2 \rangle) \} \end{aligned}$$

The intersect rule for Dag intersects the two dags in a manner similar to the intersection of two finite state automaton. The new mapping W_{12} is computed by performing intersection of the expressions on the two corresponding edges of the dags. (Rules for Intersect_s are defined in [49].)

Theorem 4.5.2 (Synthesis Algorithm Properties). (a) The GenerateStr_u procedure is sound and k -complete. The computational complexity of GenerateStr_u procedure is $O(t^2 p m \ell^4)$ (assuming $O(l^2)$ complexity for the new check on Line 10), and the size of the data structure constructed by it is $O(t^2 p m \ell^2)$, where t is the number of reachable strings in k iterations, p is the maximum number of candidate keys of any table, m is the maximum number of columns in any candidate key, and ℓ is the length of the longest reachable string. (b) The Intersect_u procedure is sound and complete. The computational complexity of Intersect_u (and hence the size of the data structure returned by it) is $O(d^2)$, where d is the size of the input data structures.

Proof. The soundness of GenerateStr_u procedure follows immediately from the soundness of the GenerateStr_t and GenerateStr_s procedures. To prove completeness, we have an invariant that $\text{GenerateStr}'_t$ generates all reachable strings of depth k (which can be proven similarly to Theorem 4.3.2(a)). The completeness of GenerateStr_u now follows from the completeness of GenerateStr_s . There are two modifications in GenerateStr_t to obtain $\text{GenerateStr}'_t$. We use GenerateStr_s for checking reachability in Line 10 which in worst case can take $O(l^2)$ time where l is the length of the output string. We consider the complexity of the check in Line 10 to be $O(l^2)$ and independent of

N (the number of table entries) as we can perform a substring check on table entries (whether $T[C, r]$ is substring of $\text{val}(\eta)$ or $\text{val}(\eta)$ is substring of $T[C, r]$) by creating a hash table of all different substrings of table entries. We also make a modification in computing the conditionals in Line 11 which introduces an additional complexity of $O(l^2)$ per reachable table entry to the complexity of GenerateStr_t from Theorem 4.3.2(a). This results in complexity of $\text{GenerateStr}'_t$ to be $O(t^2 \cdot p \cdot m \cdot \ell^2)$. The complexity of GenerateStr_s in the second step $O(t \cdot l \cdot |s|^2)$, which is dominated by the complexity of $\text{GenerateStr}'_t$ and we obtain the complexity of algorithm as $O(t^2 \cdot p \cdot m \cdot \ell^2)$. Proof of (b) follows from the soundness and completeness properties of the individual intersect procedures, namely Intersect_t and Intersect_a and the semantics of the additional rules defined above. The complexity of Intersect_u follows from the complexity of Intersect_t and Intersect_s . \square

The worst-case quadratic blowup in the size of the output returned by the Intersect procedure does not happen in practice (as we report in Section. 4.7) making the synthesis algorithm very efficient.

4.5.4 Ranking

The partial orders of ranking schemes of L_t and L_s are also used to rank expressions in L_u . In addition, we define some additional partial orders for expressions in L_u . We prefer lookup expressions that match longer strings in table entries for indexing than the ones that match shorter strings. We prefer lookup expressions with fewer constant string expressions and ones that generate longer output strings.

4.6 STANDARD DATA TYPES

The language L_u can also model a rich class of transformations on strings that represent standard data types such as date, time, phone numbers, currency, or addresses. Manipulation of these data types typically requires some background knowledge associated with these data types. For example, for dates we have the knowledge that month 2 corresponds to the string February, or for phone numbers we have the knowledge that 90 is the ISD code for Turkey. This background knowledge can be encoded as a set of relational tables in our framework (once and for all). This allows the synthesis algorithm for L_u to learn transformations over strings representing these data types. We now present some examples from Excel help-forums where users were struggling with performing manipulations over such strings.

Example 4.6.1 (Time Manipulation). An Excel user needed to have *spot times* (shown in column 1 in Figure 51(a)) converted to the hh:mm AM/PM format (as shown in column 2 in Fig-

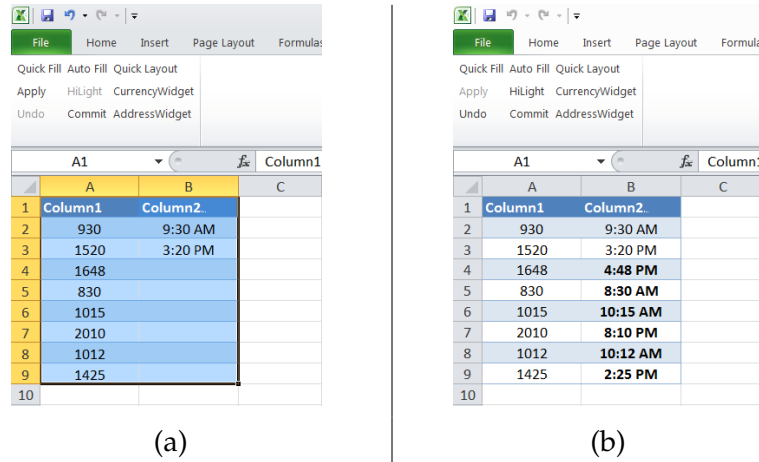


Figure 51: User interface of our programming-by-example Excel add-in. (a) and (b) are the screenshots before and after clicking the Apply button.

ure 51(a)). The user posted this problem on a help-forum to which an expert responded by providing the following macro: `TEXT(C1,"00 00")+0`. When we showed this example to a team of *Excel experts* in a live presentation, it drew a response: “There are 40 different ways of doing this!”. When we asked them to describe any one of those ways, we got the response “I do not exactly remember how to do it. I will have to investigate”.

We encode the background knowledge concerning time in a table *Time* with three columns *24Hour*, *12Hour* and *AMPM*, where the first column constitutes a primary key, and so does the combination of the second and third columns. The table is populated with 24 entries: $(0, 0, \text{AM}), \dots, (11, 11, \text{AM}), (12, 12, \text{PM}), (13, 1, \text{PM}), \dots, (23, 11, \text{PM})$. The desired transformation can be represented in our language as: `Concatenate(Select(12Hour, Time, b_1), ConstStr(" : "), SubStr(v_1 , -3, -1), ConstStr(" "), Select(AMPM, Time, b_1))` where $b_1 \equiv (24\text{Hour} = \text{SubStr}(v_1, \text{pos}(\text{StartTok}, \epsilon, 1), -3))$.

The `SubStr` expression in b_1 computes the substring of the input between the start and 3^{rd} character from end, to compute the hour part of the time in column v_1 . This hour string is then used to perform lookup in table *Time* to compute its corresponding *12Hour* format and *AMPM* value. These lookup strings are then concatenated with the minute part of the input string and constant strings `:` and .

Example 4.6.2 (Date Manipulation). An Excel user wanted to convert dates from one format to another as shown in Figure 52, 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.

We encode the background knowledge concerning months in a table *Month* with two columns *MN* and *MW*, where each of the columns

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

Figure 52: Formatting dates using examples.

constitutes a candidate key by itself. The table is populated with 12 entries: (1, January), ..., (12, December). We also maintain a table DateOrd with two columns Num and Ord, where the first column constitutes a primary key. The table contains 31 entries (1, st), (2, nd), ..., (31, st). The desired transformation is represented in L_u as :

Concatenate(SubStr(Select(MW, Month, MN = e_1), pos(StartTok, e , 1), 3),
 ConstStr(" "), e_2 , Select(Ord, DateOrd, Num = e_2), ConstStr(", "), e_3)
 where e_1 = SubStr2(v_1 , NumTok, 1), e_2 = SubStr2(v_1 , NumTok, 2), and
 e_3 = SubStr2(v_1 , NumTok, 3).

The expression concatenates the following strings: the string obtained by lookup of first number token of v_1 in table Month, the constant string whitespace, the second number token of v_1 , the string obtained by lookup of second number token of v_1 in table DateOrd, the constant string ", ", and the string corresponding to third number token of v_1 . Unfortunately, it is not possible to encode semantics of data-types with infinite domains using relational tables. One such data-type is numbers, which often entail rounding and formatting transformations [107].

Example 4.6.3 (Currency Manipulation). An administrative assistant wanted to perform a set of currency conversion tasks for reimbursement purposes. As an example, she gave the first two rows below stating that she wanted to convert 10 USD into corresponding EUR value using the conversion rate on the date 24/05/2010. The rates are looked up from a large centralized currency conversion table CurrTab comprising the exchange rates for different currencies for last 10 years.

Input v_1	Input v_2	Input v_3	Output
10 USD	EUR	2010-05-24	0.8202 * 10
20 USD	EUR	2010-05-26	0.8124 * 20
21 CHF	EUR	2010-06-03	0.7124 * 21
52 EUR	USD	2010-06-17	1.2345 * 52
25 EUR	USD	2010-06-20	1.2372 * 25
80 EUR	INR	2010-08-20	59.5146 * 80

CurrTab			
Src	Date	Dst	ExRate
USD	2010-05-24	EUR	0.8202
USD	2010-05-24	INR	46.9846
USD	2010-05-26	EUR	0.8124
EUR	2010-06-20	USD	1.2372
...

The desired transformation can be represented in L_u as: $\text{Concatenate}(e_t, \text{ConstStr}("*"), \text{SubStr2}(v_1, \text{NumTok}, 1))$, where $e_t \equiv \text{Select}(\text{ExRate}, \text{CurrTab}, b \wedge \text{Dst} = v_2 \wedge \text{Date} = v_3)$ and $b \equiv (\text{Src} = \text{SubStr2}(v_1, \text{alphaTok}, 1))$. The expression computes the ExRate value by indexing the CurrTab table with alphabet substring of input string v_1 and input strings v_2 and v_3 . This lookup output is then concatenated with constant string $*$ and the number substring of input string v_1 .

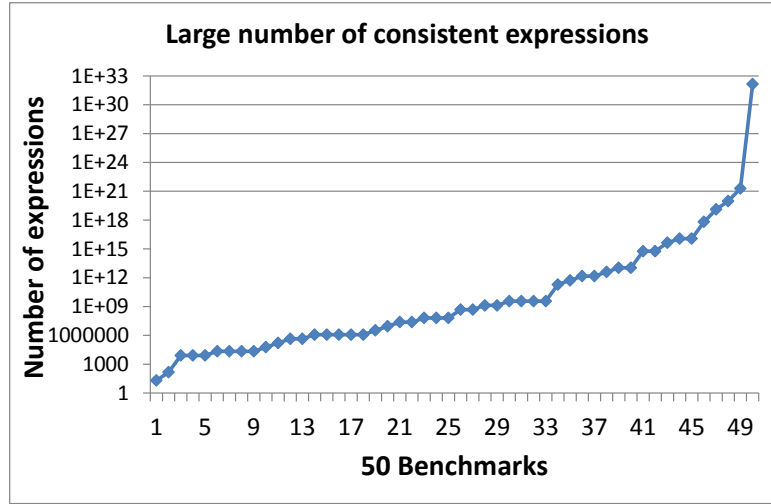
4.7 EXPERIMENTAL EVALUATION

We have implemented the inductive synthesis algorithm for the transformation language L_u in C# as an add-in for Microsoft Excel Spreadsheet system as shown in Figure 51. We hard-code a few useful relational tables of our own (such as the one that maps month numbers to month names), while also allowing the user to point to existing Excel tables to be used for performing the transformation.

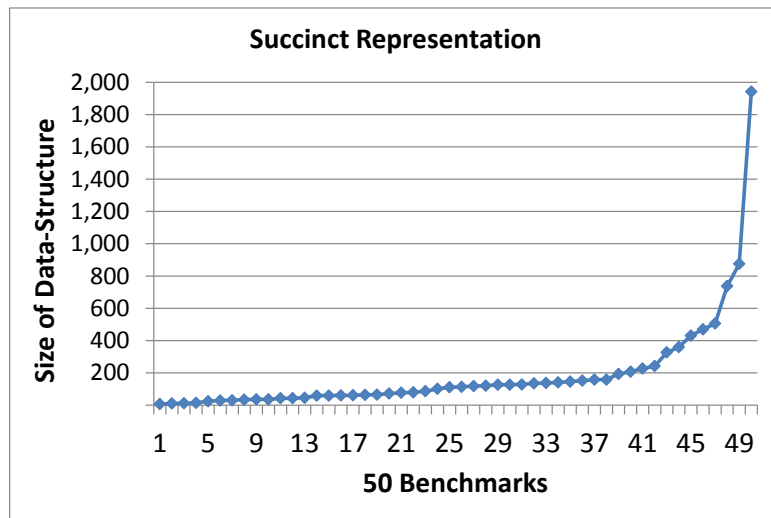
Benchmarks: We report experimental results on a set of 50 problems collected from several Excel help-forums and the Excel product team (including all problems described in this chapter). Out of these 50 problems, 12 problems can be modeled in the lookup language L_t whereas the remaining 38 of them require the extended language L_u .

Effectiveness of data structure D_u : We first present the statistics about the number of expressions in L_u that are consistent with the user-provided set of input-output examples for each benchmark problem in Figure 53(a). The figure shows that the number of such consistent expressions are very large and are typically in the range from 10^{10} to 10^{30} . Figure 53(b) shows that the size of our data structure D_u to represent this large number of expressions typically varies from 100 to 2000, where each terminal symbol in the grammar rules of the data structure contributes a unit size to the size of the data structure.

Effectiveness of ranking: Use of a ranking scheme enables users to provide fewer input-output examples to automate their repetitive task. Hence, the effectiveness of our ranking scheme can be measured by the number of examples required for the intended program to be ranked as the top-most program. In our evaluation, all benchmark problems required at most 3 examples to learn the transformation: 35

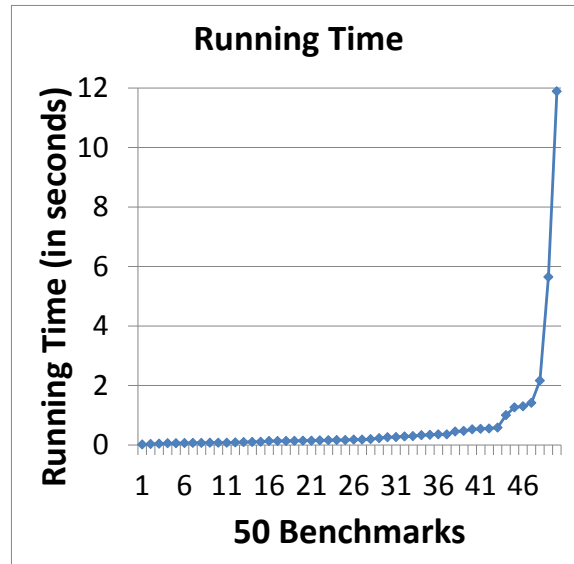


(a)

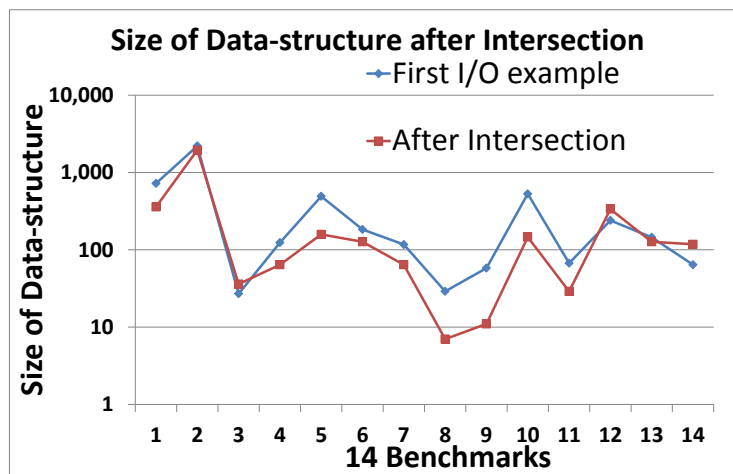


(b)

Figure 53: (a) Number of expressions consistent with given i-o examples and (b) Size of data structure (in terms of size of grammar derivation, where each terminal symbol contributes a unit size) to represent all consistent expressions.



(a)



(b)

Figure 54: (a) Running time (in seconds) to learn the program and (b) Size of the data structure D_u before and after performing Intersect_u .

benchmarks required 1 example, 13 benchmarks required 2 examples and 2 benchmarks required 3 examples.

Performance: We present the running time of our synthesis algorithm to learn the desired transformation for each benchmark problem in Figure 54(a) (sorted in increasing order). Note that 88% of benchmark problems finished in less than 1 second each and 96% of problems finished in less than 2 seconds each. The experiments were performed on a machine with Intel Core i7 1.87GHz CPU and 4GB RAM.

Size of data structure after Intersection: Finally, we show empirically that the Intersect_u procedure does not cause a quadratic blowup in the size of the data structure for any of our benchmark problems. We compare the sizes of the data structure corresponding to the first input-output example and the data structure obtained after performing the Intersect_u procedure in Figure 54(b). As we can see the size of the data structure mostly decreases after intersection and increases slightly in a few cases, but it is very far from a quadratic increase in its size.

4.8 RANKING IN FLASHFILL

We now present our novel learning to rank approach to learn a ranking function for efficiently predicting the correct (desired) expression in PBE systems from a large set of expressions induced from a few input-output examples. We show that our machine learning based approach LearnRank performs a lot better than a baseline approach of selecting the simplest program in FlashFill, and learns the desired program for 79% of benchmarks from only 1 input-output example.

4.8.1 Motivating Examples

We first present a few motivating examples from FlashFill that show three observations: (i) there are multiple correct programs in the set of programs induced from an input-output example, (ii) simple features such as size are not sufficient for preferring a correct program over incorrect programs, and (iii) there are huge number of programs induced from a given input-output example.

Example 4.8.1. An Excel user had a series of names in a column and wanted to add the title Mr. before each name. She gave the input-output example shown in Figure 55 to express her intent. The intended program concatenates the constant string "Mr." with the input string in column σ_1 .

The challenge for FlashFill to learn the desired transformation in this case is to decide which substrings in the output string "Mr. Roger" are constant strings and which are substrings of the input string

	Input σ_1	Output
1	Roger	Mr. Roger
2	Simon	
3	Benjamin	
4	John	

Figure 55: Adding Mr. title to the input names.

“Roger”. We use the notation $s[i..j]$ to refer to a substring of s of length $j - i + 1$ starting at index i and ending at index j . FlashFill infers that the substring $out_1[0..0] \equiv “M”$ has to be a constant string since “M” is not present in the input string. On the other hand, the substring $out_1[1..1] \equiv “r”$ can come from two different substrings in the input string ($in_1[0..0] \equiv “R”$ and $in_1[4..4] \equiv “r”$). FlashFill learns more than 10^3 regular expressions to compute the substring “r” in the output string from the input string, some of which include: 1st capital letter, 1st character, 5th character from end, 1st character followed by a lower case string etc. Similarly, FlashFill learns more than 10^4 expressions to extract the substring “Roger” from the input string, thereby learning more than 10^7 programs from just one input-output example. All programs in the set of learnt programs that include an expression for extracting “r” from the input string are incorrect, whereas programs that treat “r” as a constant string are correct. Some features that can help FlashFill to rank constant expressions for “r” higher than substring expressions are:

- Length of substring: Since the length of substring “r” is 1, it is less likely to be an input substring.
- Relative length of substring: The relative length of substring “r” as compared to the output string is small $\frac{1}{9}$.
- Constant neighbouring characters: The neighbouring characters “M” and “.” of “r” are both constant expressions.

Example 4.8.2. An Excel user had a list of names consisting of first and last names, and wanted to format the names such that the first name is abbreviated to its first initial and is followed by the last name as shown in Figure 56.

This example requires the output substring $out_1[0..0] \equiv “M”$ to come from the input string instead of it being the constant string “M”. The desired behavior in this example of preferring the substring “M” to be a non-constant string is in conflict with the desired behavior of preferring smaller substrings as constant strings in Example 4.8.1. Some features that can help FlashFill prefer the substring expression for “M” over the constant string expression are:

	Input σ_1	Output
1	Mark Sipser	M.Sipser
2	Louis Johnson	
3	Edward Davis	
4	Robert Mills	

Figure 56: Initials from input Firstnames.

- Output substring Token: The substring “M” of the output string is a token (Capital letter).
- String case change: The case of the substring does not change from the input substring.
- Regular expression Frequency: The regular expression to extract 1st capital letter occurs frequently in practice.

Example 4.8.3. An Excel user had a series of addresses in a column and wanted to extract the city names from them. The user gave the input-output example shown in Figure 57 to express this task.

	Input σ_1	Output
1	243 Flyer Dr,Cambridge, MA 02145	Cambridge
2	512 Wir Ave,Los Angeles, CA 78911	
3	64 128th St,Seattle, WA 98102	
4	560 Heal St,San Mateo, CA 94129	

Figure 57: Extracting city names from addresses.

FlashFill learns more than 10^6 different substring expressions to extract the substring “Cambridge” from the input string “243 Flyer Drive,Cambridge, MA 02145”, some of which are listed below.

- p_1 : Extract the 3rd alphabet token string.
- p_2 : Extract the 4th alphanumeric token string.
- p_3 : Extract substring between 1st and 2nd comma tokens.
- p_4 : Extract substring between 3rd capital token and the 1st comma token from left.
- p_5 : Extract substring between 1st and last comma tokens.

The problem with learning the substring expression p_1 is that on the input string “512 Wright Ave, Los Angeles, CA 78911”, it produces the output string “Los” that is not the desired output. On the other hand, the expression p_3 (or p_5) generates the desired output string “Los Angeles”. FlashFill needs to rank the expression p_3 (or p_5) higher than other expressions to generate the desired output

string from only one input-output example. Some of the features that can help ranking the expression p_3 higher are:

- Same left and right position logics: The regular expression tokens for left and right position logics for p_3 are similar (comma).
- Occurrence count of the match: The occurrence count of a substring between two comma tokens is 1 in comparison to the occurrence count of 3 for the `alphabet` token of p_1 .

4.8.2 Learning the Ranking Function

As we saw in the previous section, there are often a large number of (both correct and incorrect) programs induced from a few input-output examples in an expressive domain-specific language. This phenomenon is not just limited to FlashFill, but occurs regularly in most PBE systems. In this section, we formalize the problem of automatically learning a ranking function to predict a correct program from a set of correct and incorrect programs. Most previous approaches for *learning to rank* [26, 62, 43, 24] aim at ranking all relevant documents above all non-relevant documents or ranking the most relevant document as highest. However, in our case, we want to learn a ranking function that ranks any correct program higher than all incorrect programs. We use a supervised learning approach to learn such a function, but it requires us to solve two main challenges. First, we need some labeled training data for the supervised learning. We present a technique to automatically generate labeled training data from a set of input-output examples and the corresponding set of induced programs. Second, we need to learn a ranking function based on this training data. We use a gradient descent based method to optimize a novel loss function that aims to rank *any* correct program higher than *all* incorrect programs.

Preliminaries

The training phase consists of a set of tasks $T = \{t_1, \dots, t_n\}$. Each task t_i consists of a set of input-output examples $E^i = \{e_1^i, \dots, e_{n(t_i)}^i\}$, where example $e_j^i = (in_j^i, out_j^i)$ denotes a pair of input (in_i) and output (out_i). We assume that for each training task t_i , sufficiently large number of input-output examples E^i are provided such that only correct programs are consistent with the examples. Note that although preferable it is not a requirement that the examples be representative for the training task, since to evaluate the learnt ranking function we calculate how many examples are required to learn the transformation that is consistent with the provided (potentially incomplete) examples. Currently, we require PBE system designers to manually provide the examples for the training tasks, but this process can also

be partially automated using the idea of distinguishing inputs [65]. The task labels i on examples e_j^i are used only for assigning the training labels, and we will drop the labels to refer the examples simply as e_j for notational convenience. The complete set of input-output examples for all tasks is obtained by taking the union of the set of examples for each task $E = \{e_1, \dots, e_{n(e)}\} = \cup_t E^t$. Let p_i denote the set of synthesized programs that are consistent with the example e_i such that $p_i = \{p_i^1, \dots, p_i^{n(i)}\}$, where $n(i)$ denotes the number of programs in the set p_i . We define positive and negative programs induced from an input-output example as follows.

Definition 4.8.1 (Positive and Negative Program). A program $p \in p_j$ is said to be a positive (or correct) program if it belongs to the set intersection of the set of programs for all examples of task t_i , i.e. $p \in p_1 \wedge p_2 \wedge \dots \wedge p_{n(t_i)}$. Otherwise, the program $p \in p_j$ is said to be a negative (or incorrect) program i.e. $p \notin p_1 \wedge p_2 \wedge \dots \wedge p_{n(t_i)}$.

Automated Training Data Generation

We now present a technique to automatically generate labeled training data from the training tasks specified using input-output examples. Consider a training task t_i consisting of the input-output examples $E^i = \{(e_1, \dots, e_{n(t_i)})\}$ and let p_j be the set of programs synthesized by the synthesis algorithm that are consistent with the input-output example e_j . For a task t_i , we construct the set of all positive programs by computing the set $p_1 \wedge p_2 \wedge \dots \wedge p_{n(t_i)}$. We compute the set of all negative programs by computing the set $\{p_k \setminus (p_1 \wedge p_2 \wedge \dots \wedge p_{n(t_i)}) \mid 1 \leq k \leq n(t_i)\}$. The version-space algebra based representation allows us to construct these sets efficiently by performing intersection and difference operations over corresponding shared expressions.

We associate a set of programs $p_i = \{p_i^1, \dots, p_i^{n(i)}\}$ for an example e_i with a corresponding set of labels $y_i = \{y_i^1, \dots, y_i^{n(i)}\}$, where label y_i^j denotes the label for program p_i^j . The labels y_i^j take binary values such that the value $y_i^j = 1$ denotes that the program p_i^j is a positive program for the task, whereas the label value 0 denotes that program p_i^j is a negative program for the task.

Gradient Descent based Learning Algorithm

From the training data generation phase, we obtain a set of programs p_i associated with labels y_i for each input-output example e_i of a task. Our goal now is to learn a ranking function that can rank a positive program higher than all negative programs for each example of the task. We present a brief overview of our gradient descent based

method to learn the ranking function for predicting a correct program by optimizing a novel loss function.

We compute a feature vector $x_i^j = \phi(e_i, p_i^j)$ for each example-program pair (e_i, p_i^j) , $e_i \in E$, $p_i^j \in p_i$. For each example e_i , a training instance (x_i, y_i) is added to the training set, where $x_i = \{x_i^1, \dots, x_i^{n(i)}\}$ denotes the list of feature vectors and $y_i = \{y_i^1, \dots, y_i^{n(i)}\}$ denotes their corresponding labels. The goal now is to learn a ranking function f that computes the ranking score $z_i = (f(x_i^1), \dots, f(x_i^{n(i)}))$ for each example such that a positive program is ranked as highest.

This problem formulation is similar to the problem formulation of listwise approaches for learning-to-rank [24, 133]. The main difference comes from the fact that while previous listwise approaches aim to rank most documents in accordance with their training scores or rank the most relevant document as highest, our approach aims to rank any one positive program higher than all negative programs. Therefore, our loss function counts the number of examples where a negative program is ranked higher than all positive programs, as shown in Equation 1. For each example, the loss function compares the maximum rank of a negative program ($\text{Max}(\{f(x_i^j) \mid y_i^j = 0\})$) with the maximum rank of a positive program ($\text{Max}(\{f(x_i^k) \mid y_i^k = 1\})$), and adds 1 to the loss function if a negative program is ranked highest (and subtracts 1 if a positive program is ranked highest).

$$L(E) = \sum_{i=1}^{n(e)} L(y_i, z_i) = \sum_{i=1}^{n(e)} \text{sign}(\text{Max}(\{f(x_i^j) \mid y_i^j = 0\}) - \text{Max}(\{f(x_i^k) \mid y_i^k = 1\})) \quad (1)$$

The presence of sign and Max functions in the loss function in Equation 1 makes the function non-continuous. The non-continuity of the loss function makes it unsuitable for gradient descent based optimization as the gradient of the function can not be computed. We, therefore, perform smooth approximations of the sign and Max functions using the hyperbolic \tanh function and softmax function respectively (with scaling constants c_1 and c_2) to obtain a continuous and differentiable loss function in Equation 2.

$$L(y_i, z_i) = \tanh(c_1 \times (\frac{1}{c_2} \times \log(\sum_{y_i^j=0} e^{c_2 \times f(x_i^j)}) - \frac{1}{c_2} \times \log(\sum_{y_i^k=1} e^{c_2 \times f(x_i^k)}))) \quad (2)$$

We assume the desired ranking function $f(x_i^j) = \vec{w} \cdot x_i^j$ to be a linear function over the features. Let there be m features in the feature vector $x_i^j = \{g_1, \dots, g_m\}$ such that $f(x_i^j) = w_0 + w_1 g_1 + \dots + w_m g_m$. We use the gradient descent algorithm to learn the weights w_i .

of the ranking function that minimizes the loss function from Equation 2. Although our loss function is differentiable, it is not convex, and therefore the algorithm only achieves a local minima. We need to restart the gradient descent algorithm from multiple random initializations to avoid getting stuck in non-desirable local minimas.

4.8.3 Efficiently Ranking a Set of L_a Expressions

A key challenge in using any ranking methodology for expressions represented using version-space algebra based data structures is that of computational efficiency. The set of programs can not be enumerated explicitly for applying the learnt ranking function since that would break the expression sharing and make it computationally inhibitive. We need a mechanism to rank the expressions in a way that conforms to the sharing maintained by the data structure. In this section, we describe efficient features and algorithms for identifying the highest ranked expression amongst a huge set of expressions represented succinctly using Join Expressions and DAG Expressions. We first present certain properties of the features for different expression sharing that enable them to be computed efficiently over a large set of expressions. We then present corresponding algorithms that exploit these feature characteristics for efficiently computing the highest ranked expression.

Fixed Arity Expressions

Expressions with fixed arity allow *set-based sharing* among those expressions whose arguments evaluate to the same values. In order to efficiently rank these expressions, we introduce the notion of features that have abstraction functions with low abstract-dimension and project-dimension. We then present an algorithm that exploits such characteristics of features.

EFFICIENT FEATURES Our aim is to find an expression from the set of fixed arity expressions that has the highest rank $\sum_{j=1}^m w_j g_j$, where the expressions are represented succinctly using a Join expression $f(E_1, \dots, E_n)$. Since the ranking function is a linear weighted function of features, if all features depended on only one column (say E_i), we can easily enumerate the expressions individually for each column ($e \in E_i$) and compute the highest ranked expression $f(e_1, \dots, e_n)$ by selecting the highest ranked expression e_i for each individual column E_i . But often times features depend on more than one column, which leads to challenges in efficiently identifying the highest ranked expression. We use two key observations for computing with such features. The first key observation is that features that depend on

more than one column do not depend on all concrete values of other columns, but only on a few abstract values. We define a notion of *abstraction function* that characterizes the dependence of a feature on a set of abstract values for other columns. The second key observation is that the number of distinct values a feature can take for a given column are not that many in comparison to the number of expressions represented by the column. We introduce the notion of *project-dimension* for a feature to capture this set of distinct values. We then present an algorithm for selecting the highest-ranked expression whose complexity depends on product of abstract-dimensions of the abstraction functions and on the product of project-dimensions associated with the various features.

Definition 4.8.2 (Abstraction Function).

Let $f(E_1, \dots, E_n)$ denote a set of fixed arity expressions and A be a function over $E_1 \times \dots \times E_{k-1} \times E_{k+1} \times \dots \times E_n$ such that for all $e \in E_k$, for all $(e_1, \dots, e_{k-1}, e_{k+1}, \dots, e_n), (e'_1, \dots, e'_{k-1}, e'_{k+1}, \dots, e'_n) \in E_1 \times \dots \times E_{k-1} \times E_{k+1} \times \dots \times E_n$: $A(e_1, \dots, e_{k-1}, e_{k+1}, \dots, e_n) = A(e'_1, \dots, e'_{k-1}, e'_{k+1}, \dots, e'_n) \Rightarrow g(f(e_1, \dots, e_{k-1}, e, e_{k+1}, \dots, e_n)) = g(f(e'_1, \dots, e'_{k-1}, e, e'_{k+1}, \dots, e'_n))$. We refer to such a function as an *abstraction function* for feature g along column k , and refer to the cardinality of the range of function A as its abstract-dimension.

An abstraction function A of a feature g along column k captures the dependency of the feature on values of the other columns (other than column k). There may exist multiple abstraction functions for a given feature, but we select the best abstraction function (one with minimum abstract-dimension) for each feature. For example, an identity function is a valid abstraction function, but it is not a useful one since its abstract-dimension is very large (equal to the cross-product of the expression values from the remaining $n - 1$ columns). On the other extreme is an abstraction function that is a constant function (with abstract-dimension 1), which signifies that the feature only depends on a single column k . For features that depend on multiple columns, the corresponding best abstraction functions lie in between these two extremes.

Definition 4.8.3 (Project Dimension).

Let $f(E_1, \dots, E_n)$ denote a set of fixed arity expressions. The *project-dimension* r of a feature g along column k is defined as the maximum cardinality of the set of feature values $P(e) = \{g(f(e_1, \dots, e_{k-1}, e, e_{k+1}, \dots, e_n)) \mid e_i \in E_i\}$ for any $e \in E_k$, where the set $P(e)$ is referred to as the *projection set*.

The project dimension of a feature (along a column) captures the maximum number of distinct values a feature can take while keeping the column expression constant (for any expression value) and varying the expression values in other columns.

Example 4.8.4. Consider the binary position pair expression that takes as arguments two position logic expressions ($\text{pos}(r_1^l, r_2^l, c^l)$ and $\text{pos}(r_1^r, r_2^r, c^r)$). The list of features for ranking position pair expressions is shown in Figure 61 together with their abstract-dimensions, project-dimensions, and projection sets. For the feature $g_1 : v(r_1^l)$ that computes the frequency of left token sequence for left position logic, any constant function (say $c(0)$) is an abstraction function for g_1 along column 1 with abstract-dimension of 1, project-dimension of 1, and projection set $\{v(r_1^l)\}$. $c(0)$ denotes a constant function that always returns 0. For the feature $g_{13} : r_2^l = r_1^r$ that computes whether the left token sequence of right position logic is same as the right token sequence of left position logic, the function $\text{LeftTSOfRightPL}(r_1^r)$ is an abstraction function for g_3 along column 1 with an abstract-dimension of $|\tilde{p}_k|$, project-dimension of 2, and projection set $\{c(0), c(1)\}$.

EFFICIENT ALGORITHM We now describe an efficient algorithm to compute the highest ranked expression amongst the set of expressions represented succinctly using the Join expression. There are two main ideas of the algorithm. The first idea is that a scoring function such as $h = \sum_{j=1}^m w_j g_j$, which is a linear combination of features that depend on only one column is maximized by selecting the *best candidate* from each column, where the best candidate from each column is the one that maximizes the weighted combination of features that depend on that column. We use the observation of small projection set sizes to compute the best candidate expressions for each column by enumerating all possible expression values. The second key idea is that the set of all possible candidates can be partitioned into a set of classes $\text{Range}(A_1) \times \dots \times \text{Range}(A_n)$ obtained from the corresponding abstraction functions. For each such class, a feature is now dependent on only one column. Hence, the best candidate can be identified by selecting the best candidate from each class, and the best candidate for a class is obtained by identifying the best candidate from each column.

Let $f(E_1, \dots, E_n)$ be a set of fixed arity expressions and let g_1, \dots, g_m be a set of features for such expressions. Let A_j be an abstraction function for feature g_j along column k_j with abstract-dimension d_j . Let the projection set and project-dimension of feature g_j along column k_j be P_j and r_j respectively. Figure 58 describes the algorithm for finding an expression $f \in f(E_1, \dots, E_m)$ that maximizes a scoring function $h = \sum_{j=1}^m w_j g_j$ that is some positive linear combination of features g_j . The algorithm also takes as input, for each feature g_j , an abstraction function A_j along some column $k_j \in \{1, \dots, n\}$.

The algorithm first initializes the sets E'_i by enumerating all expressions $e \in E_i$ of column i and labels them with an empty set \emptyset . The

```

GetBestProg( $F(E_1, \dots, E_n), \{(w_j, g_j, A_j)\}_{j=1}^m$ )
1  Foreach  $1 \leq i \leq n$ :
2       $E'_i := \{(e, \emptyset) \mid e \in E_i\}$ ;
3  Foreach  $1 \leq j \leq m$ :
4       $E'_{k_j} := \{(e, \{(j, v_l)\} \cup T) \mid (e, T) \in E'_i, v_l \in P_j(e)\}$ ;
5  Foreach  $1 \leq i \leq n$ :
6      Sort  $(e, T) \in E'_i$  according to  $\sum_{(j, v_l) \in T} w_j v_l$ ;
7       $Best_i := \text{new Dictionary}()$ ;
8      Foreach  $(e, T) \in E'_i$  in decreasing sort order:
9           $\forall t \in \text{Tags}(e, T) \cap \text{Keys}(Best_i) : Best_i[t] := e$ ;
10  $maxScore := \perp$ ;  $maxTag := \perp$ ;
11 Foreach tag  $t \in \text{Range}(A_1) \times \dots \times \text{Range}(A_m)$ :
12      $totalScore := \sum_{i=1}^n Best_i[t]$ ;
13     if ( $totalScore > maxScore$ )
14          $maxScore := totalScore$ ;  $maxTag := t$ ;
15 return ( $Best_1[t], \dots, Best_n[t]$ );

```

Figure 58: Algorithm for finding the highest ranked expression amongst a set of fixed arity expressions.

loop at Line 3 makes $\beta_i = (|E_i| \times \prod_{1 \leq j \leq m: k_j = i} r_j)$ copies for each E_i , where each element in the copy is uniquely labelled with a set of pairs (j, v_l) where $1 \leq j \leq m$, $k_j = i$, and $v_l \in P_j(e)$. These copies are stored in E'_i . Each labelled copy $(e, (j, v_j))$ corresponds to the case where feature g_j on expression e takes the value v_j . Line 6 sorts the elements in E'_i according to its local score that is computed as weighted combinations of those features g_j that depend on column i under abstraction function A_j . In computing this local score, the appropriate weights w_j corresponding to the values v_l are used.

We define $\text{Tags}(e, T)$ to be the set of all combinations of abstract values under which the projection of feature g_j is v_l for any $(j, v_l) \in T$. The main intuition behind defining this set of abstract values is that under this partition of values, each feature g_j is only dependent on a single column k_j . Formally, $\text{Tags}(e, T) = \{(t_1, \dots, t_m) \mid \forall 1 \leq j \leq m : (j, v_l) \in T \Rightarrow (\forall (e_1, \dots, e_{k_j-1}, e_{k_j+1}, \dots, e_n) \in E_1 \times \dots \times E_{k_j-1} \times E_{k_j+1} \times \dots \times E_n : A_j(e_1, \dots, e_{k_j-1}, e, e_{k_j+1}, \dots, e_n) = t_j \Rightarrow g_j(e_1, \dots, e_{k_j-1}, e, e_{k_j+1}, \dots, e_n) = v_l)\}$. The size of the set $\text{Tags}(e, T)$ is $\alpha = \prod_{1 \leq j \leq m} d_j$.

Finally, the algorithm identifies the best candidate by selecting the best candidate from each class (Loop at Line 11), and computes the best candidate for a class t by identifying the best candidate from each column i (stored in $Best_i[t]$ at Line 9). The following theorem holds.

Theorem 4.8.1. Let $F(E_1, \dots, E_n)$ be a set of fixed arity expressions. Let g_1, \dots, g_m be any m features for such expressions. Let A_j be any abstraction function for feature g_j with abstract-dimension d_j and project-dimension r_j . Let w_1, \dots, w_m be m non-negative weights.

1. $\text{GetBestProg}(F(E_1, \dots, E_n), \{(w_j, g_j, A_j)\}_{j=1}^m)$ returns $f \in F(E_1, \dots, E_n)$ that maximizes $\sum_{j=1}^m w_j g_j$.
2. $\text{GetBestProg}(F(E_1, \dots, E_n), \{(w_j, g_j, A_j)\}_{j=1}^m)$ runs in time proportional to $n \times \alpha + \sum_{i=1}^n \beta_i \log \beta_i$, where $\alpha = \prod_{1 \leq j \leq m} d_j$ and $\beta_i = |E_i| \times \prod_{1 \leq j \leq m: k_j=i} r_j$.

Proof. (1) The correctness of the algorithm follows from the following observations. (i) A scoring function that is a linear combination of features that depend on only one column is maximized by selecting the *best candidate* from each column, where the best candidate from each column is the one that maximizes its local score, i.e., the weighted combination of features that depend on that column. (ii) The set of all possible candidates can be partitioned into various classes, where each class corresponds to a tag from $\text{Range}(A_1) \times \dots \times \text{Range}(A_n)$. For each such class, each feature is dependent on only one column. (iii) Hence, the best candidate can be identified by selecting the best candidate from each class (Loop at Line 11), and the best candidate for a class t is obtained by identifying the best candidate from each column i (stored in $\text{Best}_i[t]$ at Line 9).

(2) Note that β_i denotes the size of E'_i after the loop at Line 3. We assume that the assignment at Line 9 can be performed in time proportional to $\text{Tags}(e, T) \cap \text{Keys}(\text{Best}_i)$. This ensures that the loop at Line 8 runs in time proportional to $\alpha + \beta_i$. Hence, the loop at Line 5, which dominates the cost of the entire procedure, runs in time proportional to $\sum_{i=1}^n (\alpha + \beta_i \log \beta_i)$. \square

Associative Expressions

Associative expressions involve applying an associative operator with same input and output type to an unbounded sequence of expressions. The associative expressions allow path-based sharing, which are represented succinctly using a DAG data structure. We first present the notion of associative features and then present an efficient algorithm to compute the highest ranked expression amongst a set of associative expressions succinctly represented using a DAG.

ASSOCIATIVE FEATURES

Definition 4.8.4 (Associative Feature). A feature g over associative expressions is said to be *associative* if there exists an associative monotonically increasing binary operator \circ and a numerical feature h over expressions e_i such that $g(F(e_1, \dots, e_n)) = g(F(e_1, \dots, e_{n-1})) \circ h(e_n)$.

Example 4.8.5. The associative features for the Concatenate expression together with their corresponding binary operators and numerical features are shown in Figure 63. For the feature $g_1 : \text{NumArgs}$ that counts the number of arguments to the concatenate expression, operator \circ is the addition operator and h is the constant function $c(1)$. For the feature $g_3 : \text{ProdWeights}$ that computes the product of weights of edges on a concatenate expression path, operator \circ is the multiplication operator and h is the weight function.

EFFICIENT ALGORITHM We now describe an efficient algorithm for finding the highest ranked expression from an exponential number of expressions represented succinctly using the path-based sharing. The main idea behind the algorithm is to use a dynamic programming algorithm similar to Dijkstra's shortest path algorithm, where each node maintains the highest ranked path from the start node to itself, together with the corresponding feature values. The algorithm for computing the highest ranked expression from a set of DAG expressions is shown in Figure 59. The algorithm takes as input a DAG $\text{Dag}(\tilde{\eta}, \eta^s, \eta^t, W)$ and m weights with m associative features $\{(w_j, g_j)\}_{j=1}^m$, and returns an associative expression with the maximum score $\sum_{j=1}^m w_j g_j$.

The algorithm first topologically sorts the nodes in the DAG (Line 1) to get a list of nodes Q , and initializes the score and parent (par) values of each node to be $-\infty$ and \perp respectively. The score value of a node η stores the maximum value of the ranking function of any path from the start node η^s to η , whereas the par field stores a pointer to the previous node for the path of maximum score. For notational convenience, we use a special value ζ to denote the identity element for associative features such that for all values α , we have $\zeta \circ \alpha = \alpha$. Each node η is associated with a set of values $\{g'_k\}_{k=1}^m$ that stores the value of the associative features $\{g_k\}_{k=1}^m$ of the maximum score path from η^s to η . For the start node, the values g'_k and score are initialized to be ζ and 0 respectively.

The algorithm iterates over the set of nodes $i \in Q$ in the topologically sorted order to compute the values g'_k and score of the neighboring nodes j ($e_{ij} \in \text{edges}(W)$) (Lines 7-13). It computes the value $\text{score}' = \sum_{k=1}^m w_k (g'_k[i] \circ_k h_k(e_{ij}))$ for each node j , and accordingly updates the g'_k and score values for the node if the score' value is higher than the current score value for the node. The algorithm fi-

```

GetBestProg(Dag( $\tilde{\eta}, \eta^s, \eta^t, W$ ),  $\{(w_j, g_j)\}_{j=1}^m$ )
1  Q := TopologicalSort( $\tilde{\eta}, \eta^s, \eta^t, W$ )
2  Foreach  $\eta \in \tilde{\eta}$ :
3      score[ $\eta$ ] =  $-\infty$ ; par[ $\eta$ ] :=  $\perp$ ;
4  Foreach  $1 \leq k \leq m$ :
5       $g'_k[\eta^s] := \zeta$  // identity element
6  score[ $\eta^s$ ] = 0;
7  Foreach  $i \in Q$ : // in topologically sorted order
8      Foreach  $e_{ij} \in \text{edges}(W)$ :
9           $\text{score}' = \sum_{k=1}^m w_k(g'_k[i] \circ_k h_k(e_{ij}))$ ;
10         if (score[j] < score')
11             score[j] = score'; par[j] = i;
12         Foreach  $1 \leq k \leq m$ :
13              $g'_k[j] := g'_k[i] \circ_k h_k(e_{ij})$ ;
14  N := (); j :=  $\eta^t$ ;
15  while (par[j]  $\neq \perp$ ):
16      N := ( $e_{\text{par}[j], j}$ ) + N; j := par[par[j]];
17  return N;

```

Figure 59: Algorithm for finding the highest ranked expression amongst a set of associative expressions represented as a DAG.

nally returns the highest ranked path N constructed using the par values.

The following theorem holds.

Theorem 4.8.2. Let D be a DAG representing a succinct collection of associative expressions. Let g_1, \dots, g_m be m different associative features over associative expressions and let $\{w_1, \dots, w_m\}$ be some real-values non-negative weights.

1. $\text{GetBestProg}(D, \{(w_j, g_j)\}_{j=1}^m)$ returns $f \in D$ that maximizes $\sum_{j=1}^m w_j g_j$.
2. $\text{GetBestProg}(D, \{(w_j, g_j)\}_{j=1}^m)$ runs in time proportional to $O(n^2)$, where n is the number of nodes in D.

Proof. (1) The correctness follows from the following observations. (i) The rank of a path $\langle \eta^s, \eta_1, \dots, \eta_{p-1}, \eta_p \rangle$ can be computed from features of the path $\langle \eta^s, \eta_1, \dots, \eta_{p-1}, \eta_p \rangle$ using the property of the associative features $\sum_{j=1}^m w_j g_j(e_1, \dots, e_p)$

$= \sum_{j=1}^m w_j (g_j(e_1, \dots, e_{p-1}) \circ_k h_k(e_p))$. (ii) The maximum value of the ranking function for a path $(e_1, \dots, e_{p-1}, e_p)$ corresponds to the features for which the rank of path (e_1, \dots, e_{p-1}) is maximized, i.e.

$$\arg \max_{e_1, \dots, e_p} \sum_{j=1}^m w_j g_j(e_1, \dots, e_p) =$$

$\sum_{j=1}^m w_j (\arg \max_{e_1, \dots, e_{p-1}} (g_j(e_1, \dots, e_{p-1})) \circ_k h_k(e_p))$, as \circ_k is a monotonically increasing operator and $h_k(e_p)$ is a non-varying value for the edge e_p .

(2) Let the number of nodes in the DAG be n . The number of edges in the graph are $O(n^2)$. The topological sort on the DAG takes $O(n^2)$ time. The amortized complexity analysis of the loop (Lines 7-13) results in $O(n^2)$ complexity as each edge of the DAG is visited once for computing the score values. Therefore, the complexity of the algorithm is $O(n^2)$. \square

4.8.4 Case Study: FlashFill

We now present an instantiation of our ranking scheme for the FlashFill synthesis algorithm [49]. We use FlashFill as a case study because of the access to several FlashFill benchmarks from online forums, tutorials, and blogs. FlashFill uses a version-space algebra based data-structure to succinctly represent a huge set of programs. The expressions in FlashFill are shared at three different levels: (i) set-based sharing of position pair expressions at the lowest level, (ii) union expressions for atomic expressions on the DAG edges, and (iii) path-based sharing of concatenate expressions at the top level. We first describe efficient features for expressions at each one of these levels that conform to their respective sharing. We then learn a separate ranking function for each one of these levels using the gradient descent algorithm. We rank FlashFill expressions in a hierarchical manner by first computing highest ranked position pair expressions, then highest ranked atomic expressions, and then highest ranked concatenate expressions to compute the highest ranked top level program. We also present the evaluation of effectiveness and efficiency of the ranking scheme on several benchmark tasks.

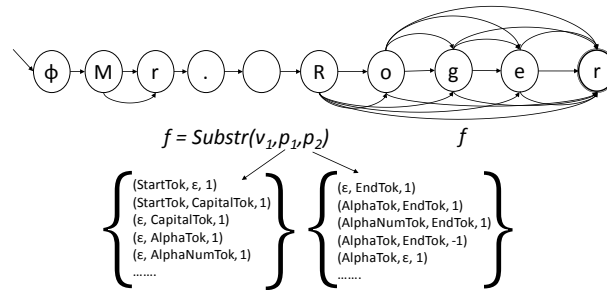


Figure 60: The DAG data structure for representing all programs induced by programs in Example 4.8.1.

The top-level DAG data structure `Dag` represents an exponential number of programs in polynomial space by sharing expressions for substrings of the output string. The DAG data structure represent-

ing all programs consistent with the input-output example in Example 4.8.1 is shown in Figure 60. The horizontal edges denote constant string expressions whereas the curved edges denote position pair expressions. Each position pair expression is further represented as two independent sets of left and right position logic expressions as shown for the substring expression a in the figure. The position expression $(\text{StartTok}, \epsilon, 1)$ denotes the 1st occurrence of StartTok token (that represents the beginning of the string). In this way, the DAG encodes a huge number of possible programs in the language that are consistent with the examples.

Automated Training Data Generation

We now show how to compute the set of positive and negative programs for a task automatically from a given set of input-output examples in FlashFill. Let Dag_k denote the DAG learnt by FlashFill for the input-output example e_k and Dag_\wedge be the DAG obtained after intersecting the DAGs for all examples, i.e. $\text{Dag}_\wedge \equiv (\text{Dag}_1 \wedge \text{Dag}_2 \cdots \wedge \text{Dag}_n)$. To compute positive and negative programs from the set p_k , the first challenge is to align the edges in the two DAGs. After aligning the edges, the common edges between the two dags constitute the positive expressions whereas the edges that are present in Dag_k but not in Dag_\wedge constitute the negative expressions. We run the DAG programs Dag_k and Dag_\wedge on the input string in_k and annotate the dag nodes with the indices of the output string out_k using the label function $\mathcal{L} : \eta \rightarrow \text{int}$. The start node η^s of a dag is annotated with index 0, such that $\mathcal{L}(\eta^s) = 0$. A node η_2 in a dag is annotated with index m (i.e. $\mathcal{L}(\eta_2) = m$) if we have $\mathcal{L}(\eta_1) = l$ and the expressions on the dag edge (η_1, η_2) generates the string $\text{out}_k[l..m]$ when run on the input string in_k . Once we have annotated the nodes of both dags, we collect the expressions on edges between nodes that have the same labels to compare. We denote the set of expressions that generates the string $\text{out}_k[l..m]$ in dag Dag_k as $s_{l,m,k}$, where $s_{l,m,k} \equiv \bigcup_{\eta_1, \eta_2 \in \text{Dag}_k} s(\eta_1, \eta_2), \mathcal{L}(\eta_1) = l, \mathcal{L}(\eta_2) = m$. We denote expressions that appear in $s_{(l,m,\wedge)}$ as positive expressions and expressions that appear in the set $s_{(l,m,k)} \setminus s_{(l,m,\wedge)}$ as negative expressions. In this manner, we compute the set of positive and negative expression values for each input-output example pair of a training benchmark.

Efficient Expression Features

The set of efficient expression features used for ranking expressions at each level of DAG sharing are as follows.

Feature	A, Col.	Abs. Dim.	Proj. Dim.	Proj. Set
$g_1 : v(r_1^l), g_2 : v(r_2^l)$	$c(0), 1$	1	1	$\{v(e)\}$
$g_3 : v(c^l), g_4 : v((r_1^l, r_2^l))$	$c(0), 1$	1	1	$\{v(e)\}$
$g_5 : v(\text{Length}r_1^l), g_6 : v(\text{Length}r_2^l)$	$c(0), 1$	1	1	$\{v(e)\}$
$g_7 : v(r_1^r), g_8 : v(r_2^r)$	$c(0), 2$	1	1	$\{v(e)\}$
$g_9 : v(c^r), g_{10} : v((r_1^r, r_2^r))$	$c(0), 2$	1	1	$\{v(e)\}$
$g_{11} : v(\text{Length}r_1^r), g_{12} : v(\text{Length}r_2^r)$	$c(0), 2$	1	1	$\{v(e)\}$
$g_{13} : r_2^l = r_1^r$	$r_1^r, 1$	$ \tilde{p}_k $	2	$\{c(0), c(1)\}$
$g_{14} : r_2^l = \epsilon \wedge r_1^r = \epsilon$	$g_{14}, 1$	2	2	$\{c(0), c(1)\}$
$g_{15} : r_1^l = \epsilon \wedge r_2^r = \epsilon$	$g_{15}, 1$	2	2	$\{c(0), c(1)\}$

Figure 61: The set of features for ranking position pair expression $\text{SubStr}(\sigma_i, \{\tilde{p}_j\}_j, \{\tilde{p}_k\}_k)$, where $\tilde{p}_j = \text{pos}(r_1^l, r_2^l, c^l)$, $\tilde{p}_k = \text{pos}(r_1^r, r_2^r, c^r)$. The table also shows the abstraction function, abstract dimension, project dimension, and projection set for features.

POSITION PAIR EXPRESSION FEATURES The binary position pair expressions take two position logic expressions as arguments. The features used for ranking the position pair expressions are shown in Figure 61 together with their abstraction functions, abstract-dimensions, project-dimensions, and projection sets. Note that all the listed features are efficient as they have low abstract and project dimensions. These features include frequency-based features denoting frequencies of: left and right token sequences of left position logic expression (g_1, g_2), occurrence Id of left position logic (g_3), left position logic (g_4), length of left and right token sequences of left position logic (g_5, g_6), left and right token sequences of right position logic expression (g_7, g_8), occurrence Id of right position logic (g_9), right position logic (g_{10}), length of left and right token sequences of right position logic (g_{11}, g_{12}). In addition to frequency-based features, there are also Boolean features that include whether the right token sequence of left position logic is equal to the left token sequence of the right position logic (g_{13}), the right token sequence of left position logic and left token sequence of right position logic are empty (g_{14}), and the left token sequence of left position logic and right token sequence of right position logic are empty (g_{15}).

The frequencies of position logic expressions are obtained from the occurrence of position logic expressions in the training data and are computed as follows. Let $n_+(r)$ and $n_-(r)$ denote the number of times a sub-expression r occurs as a correct and incorrect position logic expression respectively in the training set, and let $n(r) = n_+(r) + n_-(r)$. We compute the frequency of an expression $v(r_1)$ as $\frac{n_+(r_1)}{\sum_r n(r)}$.

Feature	Description
$g_1 : \text{isOutLTok}$	left position of s matches a token
$g_2 : \text{isOutRTok}$	right position of s matches a token
$g_3 : \text{isInLTok}$	left position of i matches a token
$g_4 : \text{isInRTok}$	right position of i matches a token
$g_5 : \text{isConstExpr}$	is a print constant expression
$g_6 : \text{Casing}$	casing transformation to obtain s
$g_7 : \text{RelLenInSubstr}$	$\text{lenSubstr} / \text{lenInpStr}$
$g_8 : \text{IsOutLConst}$	left expression of s is constant
$g_9 : \text{IsOutRConst}$	right expression of s is constant
$g_{10} : \text{IsPPEExpr}$	is a position pair expression
$g_{11} : \text{LenSubstr}$	length of s
$g_{12} : \text{RelLenOutSubstr}$	$\text{lenSubstr} / \text{lenOutStr}$
$g_{13} : \text{RankPPEExpr}$	rank of position pair expression

Figure 62: The features for ranking expressions on DAG edges, where s denotes the string obtained from executing the DAG edge expression and i denotes the input substring that matches with s .

ATOMIC EXPRESSION FEATURES An atomic expression corresponds to a substring of the output string, which can come from several positions in the input string in addition to being a constant string. This leads to multiple atomic expression edges between any two nodes of the DAG, which are represented explicitly using a Union expression and therefore we can use any set of features for ranking these expressions. The features for ranking expressions at this level are shown in Figure 62, some of which include whether the output substring and the corresponding input substring are at token boundaries or not, the expression is a constant string or a substring expression, absolute and relative lengths of the substring as compared to input and output strings, the left and right expressions of the output substring are constant expressions or not, and the rank of position pair expression obtained from the previous level.

CONCATENATE EXPRESSION FEATURES At the top-level of DAG, we use associative features to compute the ranking of paths. The set of associative features together with their corresponding binary operator and numerical feature are shown in Figure 63. These features include number of arguments in the Concatenate expression (g_1), the sum of weights of edges on the path (g_2), the product of weights of edges on the path (g_3), and the maximum (g_4) and minimum (g_5) weights of an edge on the path.

Feature	Binary Operator \circ	Numerical Feature h
$g_1 : \text{NumArgs}$	+	$c(1)$
$g_2 : \text{SumWeights}$	+	weight
$g_3 : \text{ProdWeights}$	\times	weight
$g_4 : \text{MaxWeight}$	Max	weight
$g_5 : \text{MinWeight}$	Min	weight

Figure 63: The set of associative features for ranking a set of $\text{Concatenate}(a_1, \dots, a_n)$ expressions together with their binary operator and numerical feature.

Experimental Evaluation

We now present the evaluation of our ranking scheme for FlashFill on a set of 175 benchmark tasks obtained from various online Excel help forums. We evaluate our algorithm on three different train-test partition strategies, namely 20-80, 30-70 and 40-60. For each partition strategy, we randomly assign the corresponding number of benchmarks to the training and test set. The experiments were performed on an Intel Core i7 3.20 GHz CPU with 32 GB RAM.

Training phase: We run the gradient descent algorithm 1000 times with different random values for initialization of weights, while also varying the value of the learning rate α from 10^{-5} to 10^5 (in increments of multiples of 10). We learn the weights for the ranking functions for the initialization and α values for which best performance is achieved on the training set.

Test phase: We compare the performance of the following two ranking schemes on the basis of number of input-output examples required to learn the desired task.

- **Baseline (No Ranking):** No ranking of expressions. We keep adding input-output examples to the task until there are only positive expressions remaining in the DAG.
- **LearnRank:** Our ranking scheme that uses the gradient descent algorithm to learn the ranking functions for position pair, atomic, and concatenate expressions in DAG.

One thing to note is that the algorithms for finding highest ranked position-pair and concatenate expressions require weights of the ranking function to be non-negative, but the gradient descent algorithm in the training phase can learn some negative weight values as well. In such cases of negative weight values, we treat the weight values as positive and make the corresponding feature values negative.

COMPARISON WITH BASELINE (NO RANKING) The average number of input-output examples required to learn a test task for 10 runs

Train-Test Partition	Average Examples	
	Baseline	LearnRank
20-80	4.19	1.52 \pm 0.07
30-70	4.17	1.49 \pm 0.06
40-60	4.18	1.44 \pm 0.07

Table 2: The average number of examples to learn test tasks for 10 runs of different training partitions.

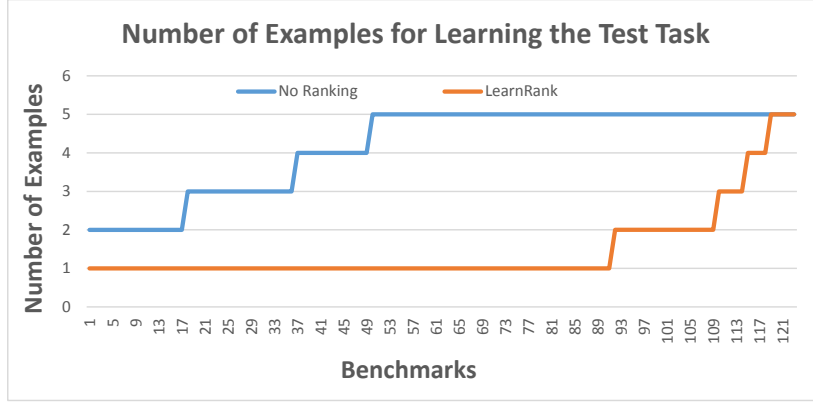


Figure 64: Comparison of LearnRank with the Baseline scheme for a random 30-70 partition.

of different train-test partitions is shown in Table 2. We can observe that with the introduction of ranking (LearnRank), the number of input-output examples required to learn the test tasks are drastically reduced. The LearnRank scheme performs much better than Baseline in terms of average number of input-output examples required to learn the desired task (1.49 vs 4.17). For a random 30-70 partition run, the number of input-output examples required to learn the 123 test benchmark tasks under the two ranking schemes is shown in Figure 64. The LearnRank scheme learns the desired task from just 1 example for 91 tasks (74%) as compared to 0 for Baseline. The LearnRank scheme learns the desired tasks from at most 2 examples for 110 tasks (89%), whereas Baseline learns only 18 tasks (14%) from 2 examples.

EFFICIENCY OF LEARNRANK We now present the efficiency of LearnRank scheme that uses efficient features and algorithms for maintaining the corresponding sharing between expressions while computing the highest ranked expressions. For evaluating the overhead of LearnRank scheme, we compare the running times of FlashFill without any ranking (NoRanking) and FlashFill augmented with LearnRank scheme over the same number of input-output examples for each test task. Note that even though the LearnRank scheme can

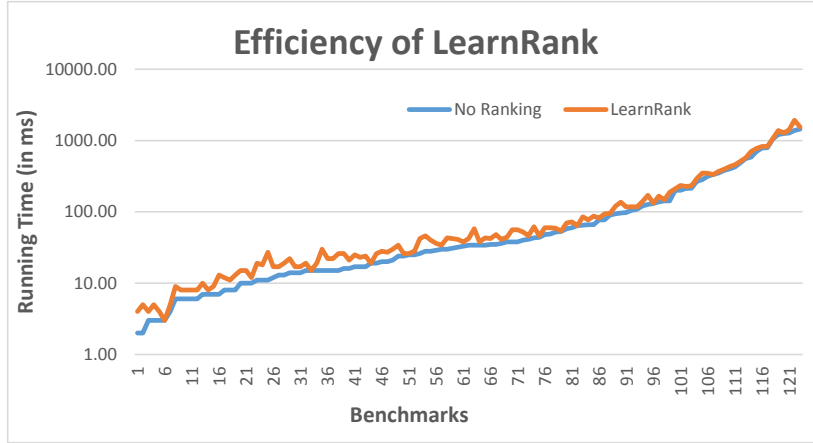


Figure 65: The running times for FlashFill without ranking (NoRanking) and with LearnRank.

learn the desired task from much fewer input-output examples, we still use all the examples for learning for a fair comparison. The running times of the two FlashFill versions is shown in Figure 65. We can observe that the overhead of LearnRank is quite small and the two running times are comparable. The average overhead of LearnRank over NoRanking is about 20 milliseconds (ms) per benchmark task whereas the median overhead is about 8 ms. This translates to an average overhead of about 29% and a median overhead of 25% in running times as compared to NoRanking.

STORYBOARD PROGRAMMING TOOL

Students learning to program find it challenging when there is a large gap between the abstractions at which algorithms are taught and explained in classrooms and the abstractions at which they are required to be programmed. One domain where this gap is rather large is data-structure manipulations, which are typically described using high-level “boxes-and-arrows” diagrams. Their translation to low-level pointer manipulating code, however, is non-intuitive, tedious, and error-prone.

To illustrate this gap, consider the manipulation shown in Figure 66. Part (a) of the figure shows a graphical description of the removal of a node from a doubly linked-list. The diagram—we call it a *storyboard*—communicates very clearly the effect of the manipulation. By contrast, the imperative code in Figure 66(b) is short but not self-explanatory; understanding this code essentially requires one to mentally recreate the image from the storyboard in part (a). We present the Storyboard Programming Tool (SPT) that aims to bridge this gap by using constraint-based synthesis technology to automatically implement data-structure manipulations that are provably correct with respect to high-level descriptions like the one illustrated by Figure 66.

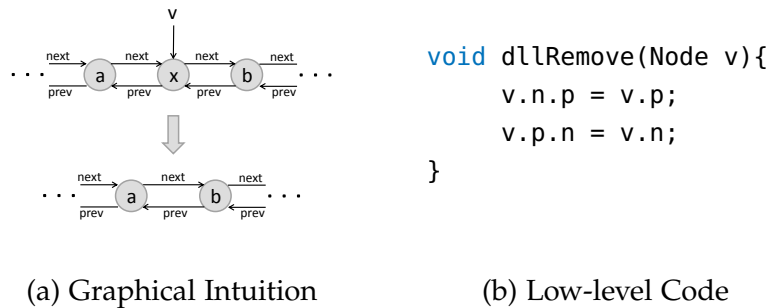


Figure 66: Doubly linked list deletion example in SPT.

The SPT system presented in this chapter assumes a textual input describing the storyboard, which is a set of descriptions of the state of a data-structure before and after manipulation. However, SPT has been extended with a graphical interface where users can draw storyboards using digital ink as well as provide voice-based inputs [100]. SPT supports a form of *Programming by Example* (PBE) [29] where the manipulations are synthesized from partial descriptions of their effects. Two important features, however, distinguish our system from traditional PBE systems. The first is the ability to abstract away those parts of the data-structure that are not relevant to the manipulation, as is done in Figure 7 through the use of ellipsis. This form of abstrac-

tion gives our system a lot of expressive power, because it allows a single figure to succinctly describe the behavior of the algorithm on an infinite number of concrete inputs, turning a simple input-output pair into a partial specification. The second difference with PBE is that our system asks the user to provide information about the loop skeleton of the solution; this information reduces the space of possible implementations that the system needs to consider and makes it less likely that the system will produce a solution that only works for the given examples. These two features allow us to synthesize complex data-structure manipulations from relatively simple storyboards.

The instructors today have two choices for testing whether students understand a data-structure manipulation: (i) ask the student to program it, or (ii) ask them to visually describe the manipulation. The problem with the first choice is that instructors are not just testing the understanding of the manipulation but also the programming skills. This problem is ameliorated with the second choice but now instructors need to grade these diagrams by hand. SPT helps instructors get best of both worlds because with SPT students do not need to program and the visual descriptions can be checked mechanically by verifying the synthesized low-level code.

The system is made possible by a new synthesis algorithm that combines previous work on constraint-based synthesis [117, 122] with abstract interpretation. Our algorithm is not the first to do this [122], but it is the first to scale to the large and complex abstract domains required to reason about data-structure manipulations. The key idea behind the algorithm is to use quantification to eliminate operations that require complex set-based reasoning, and to use the SKETCH solver [117] to solve these constraints. The new synthesis algorithm allows us to combine constraint-based synthesis with a form of shape-analysis loosely based on TVLA [77]. The shape analysis algorithm used by our system is not as powerful as many of those found in the literature [102], but it is powerful enough to reason about most operations involving trees and lists. The strength of our particular form of shape analysis, however, lies in the ease with which we can take the abstractions expressed as part of the storyboard and use them as the basis for an abstract domain that is then used to verify each candidate implementation.

We have used SPT to successfully synthesize several data structure manipulations such as insertion, deletion, search, reversal, and rotation operations over singly linked list, doubly linked list and binary search tree data structures. We have also used our framework to synthesize small puzzle problems as well as some manipulations involving a tricky real-world And Inverter Graph [86] data-structure used in the ABC solver [19].

5.1 EXAMPLE MANIPULATIONS WITH SPT

We now present an overview of the specification mechanism in SPT through two textbook data-structure manipulation examples: in-place linked list reversal and linked list deletion. Reversing a list with a loop and using only a constant amount of additional memory is non-trivial; in fact, the algorithm for this manipulation is a common question in technical interviews. The linked list deletion manipulation requires the implementation to iterate over the linked list until reaching the desired node to be deleted, and then requires a precise sequence of pointer updates to delete the node. We describe how a user describes the intended data-structure manipulations to SPT.

5.1.1 In-place Linked List Reversal

The specification mechanism of SPT is a *storyboard* that is composed of three elements: a set of *scenarios*, each of which corresponds to an abstract input-output pair; a set of fold and unfold definitions, and a skeleton of the looping structure of the desired algorithm.

A scenario in SPT is an input-output pair describing the effect of the manipulation on a potentially abstract data-structure, where abstraction is used to elide details of the data-structure that are not considered relevant. For example, Figure 67 below shows the main scenario describing the effect of reversing a linked list. Like the more

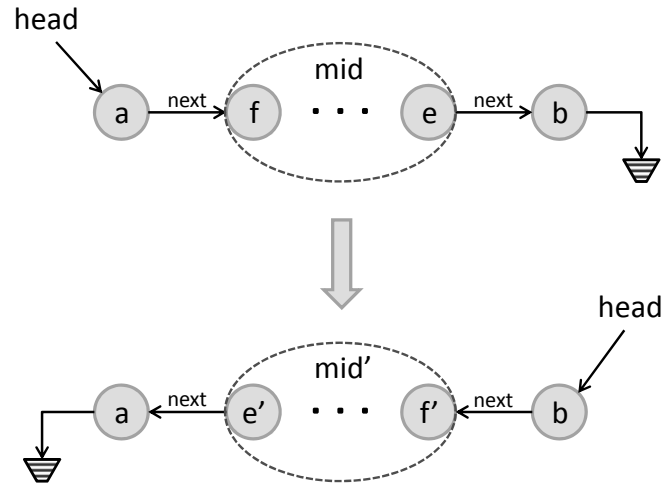


Figure 67: Graphical description of linked-list reverse

informal example in Figure 7, the scenario uses ellipses to abstract away part of the list. In our notation, however, the ellipses are formalized by using the concept of *summary nodes*. For this example, the scenario uses a summary node *mid* to represent the middle part of the list, which may vary in size for different runs of the algorithm. Out of

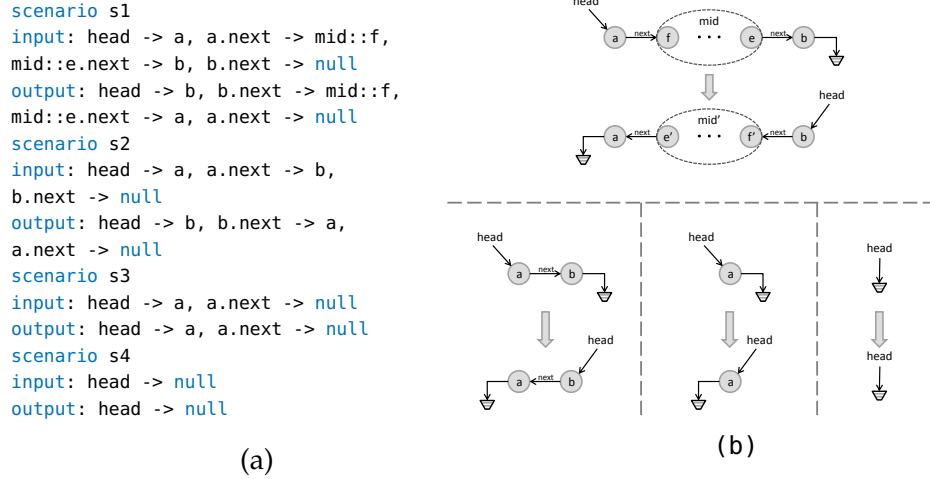


Figure 68: The storyboard scenarios for in-place linked list reversal.

all the nodes in the sub-list represented by *mid*, the first and last node deserve special attention because other nodes outside the sub-list *mid* may point to them. We call these special nodes *attachment points* of the summary node, and as we shall see, they play an important role in reasoning about scenarios. Figure 68(b) shows the complete set of scenarios needed for this example, including scenarios to describe the behavior of the algorithm on lists of length zero, one, and two. Figure 68(a) presents the text notation used by our system to describe the scenarios on the right.

In order to make scenarios precise, it is often necessary to provide additional information about the structure of summary nodes. For example, in Figure 67, the summary node *mid* represents a set of nodes with a very particular structure; specifically, the scenario only makes sense under the assumption that node *e* is reachable from node *f*. Our system allows the user to provide this structural information through fold and unfold predicates. For example, Figure 69 shows the fold and unfold predicates used to describe the recursive structure of the *mid* summary node. The predicates describe the structure of summary nodes in terms of their attachment points, in a similar way as Fradet *et al.* [42] used context-free graph grammars to describe shape types.

The exact syntax and semantics of the predicates will be discussed in detail in Section 5.3.2; for now, it is enough to understand that the predicates in Figure 69(a) are precise text representations of the recursive definitions illustrated in Figure 69(b). For example, the unfold rule shows two alternatives for the summary node *mid*: either the attachment points *f* and *e* are actually the same node *x'*, and this is the only node in *mid*, or *f* is a node *x'* whose next pointer points to the attachment point *f'* of another similar summary node *mid'*. For this example, fold is an inverse of unfold and could be derived automati-

cally, but Section 5.3.2 will show other examples where it is useful to define fold and unfold independently as a way to guide the solver to a specific solution.

The scenarios and the fold and unfold definitions together describe the effects of the desired manipulation, but recall that our specification included some non-functional requirements, such as the requirement that the implementation uses a single loop. This requirement is expressed by providing a skeleton of the looping structure of the desired algorithm. Figure 70 shows the skeleton for list reverse, which states that the implementation should contain exactly one while loop with some blocks of code before and after it. It consists of a while loop with a set of unknown statements before the loop, in the loop body and after the loop. The operator $**$ denotes a comparison expression, operator $??(n)$ denotes a block consisting of n unknown pointer assignment statements, and operator $?=(n)$ denotes a block consisting of n conditional assignment statements of the form $\text{if}(**) ??(1)$.

```

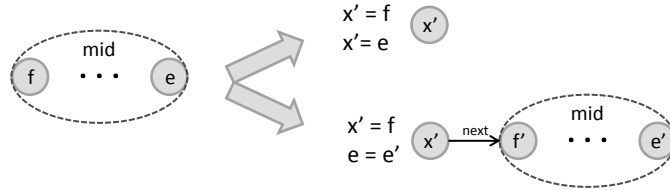
unfold mid::f x [in (mid::f, x)] [out (mid::e, x)] ()
unfold mid::f x [in (mid::f, x)] [out (mid::e, mid::e)] (x.next -> mid::f)

fold x mid::f [in (x, mid::f)] [out (x, mid::e)] ()
fold x mid::f [in (x, mid::f)] [out (mid::e, mid::e)] (x.next -> mid::f)

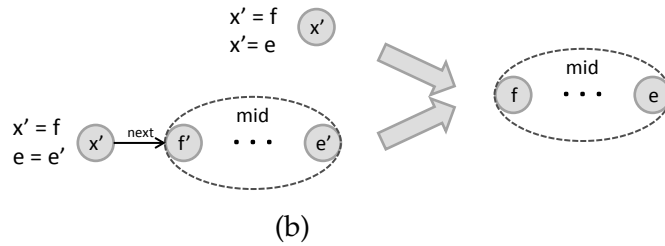
```

(a)

Unfold:



Fold:



(b)

Figure 69: Unfold and fold predicate definitions for mid summary node.

From a satisfying assignment to the constraints, the framework derives the imperative implementation shown in Figure 71(a) for the linked list reverse manipulation. The conditionals (true and false) from the conditional assignment statements are removed for better readability of the code in Figure 71(b). It can be noted that the imple-

```

Node llReverse(Node head){
  Node temp1, temp2, temp3;
  ?=(2) /* F1 */
  while(**){ /* F2 */
    ??(4) /* F3 */
  }
  ?=(2) /* F4 */
  return head;
}

```

Figure 70: Control flow sketch for list reversal. Each number corresponds to an unknown block of code.

<pre> Node llReverse(Node head){ Node temp1 = null, temp2 = null; Node temp3 = null; if(false) head = head; if(true) temp1 = head; while(temp1 != null){ // unfold temp1; head = temp1; temp1 = temp1.next; head.next = head; head.next = temp3; temp3 = head; // fold head; } if(false) head.next = head.next; if(false) head.next = head.next; } </pre>	<pre> Node llReverse(Node head){ Node temp1 = null, temp2 = null; Node temp3 = null; temp1 = head; while(temp1 != null){ // unfold temp1; head = temp1; temp1 = temp1.next; head.next = head; head.next = temp3; temp3 = head; // fold head; } } </pre>
(a)	(b)

Figure 71: (a) The synthesized implementation of list reverse, and (b) cleaned up version of (a) with false conditionals removed.

mentation did not use the program variable `temp2` and the loop body includes an extra *dead store* assignment statement (`head.next = head`). Aside from the additional assignment, the code is an efficient implementation of the desired algorithm, and the entire synthesis process takes only a couple of minutes.

5.1.2 Linked List Deletion

```

scenario s1
input: head -> front::f, y -> ly, front::l.next -> ly, back::l -> null, ly.next ->
      back::f
output: head -> front::f, front::l.next -> back::f, back::l.next -> null
scenario s2
input: head -> front::f, y -> ly, ly.next -> null, front::l.next -> ly
output: head -> front::f, front::l.next -> null
scenario s3
input: head -> ly, ly.next -> back::f, y -> ly, back::l.next -> null
output: head -> back::f, back::l.next -> null
scenario s4
input: head -> ly, ly.next -> null, y -> ly
output: head -> null

```

Figure 72: Textual description of linked list deletion storyboard shown in Figure 73.

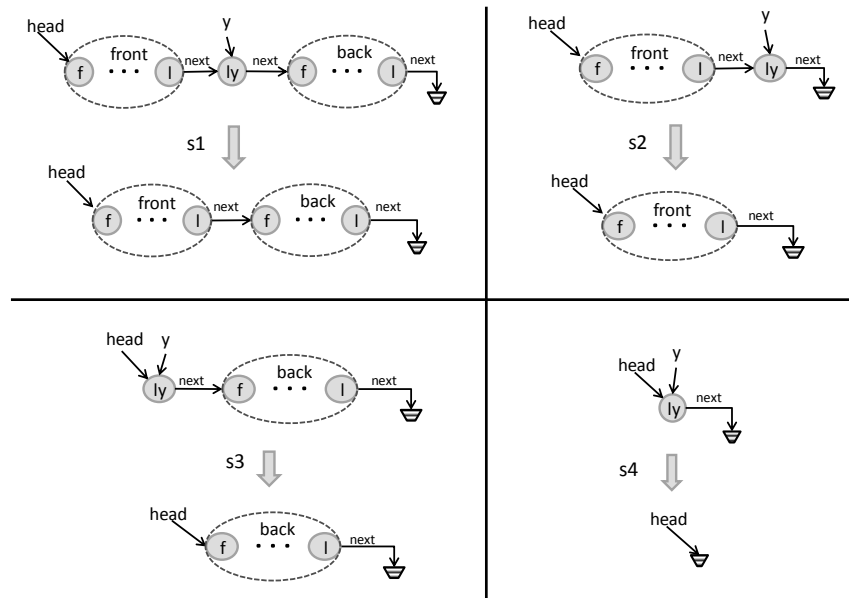


Figure 73: The storyboard consisting of four visual scenarios describing the input and output state descriptions for linked list deletion.

The scenarios for linked list deletion are shown in Figure 72, and the corresponding visual description of the scenarios is shown in Figure 73. The first scenario `s1` describes an abstract input-output example, where the input list consists of two *summary nodes* `front` and `back` and a concrete node `ly` that is to be deleted. The summary node

front contains two attachment points $\text{front}::f$ and $\text{front}::l$ denoting the first and last elements of the front list respectively. The other scenarios s_2 , s_3 and s_4 correspond to the cases of deleting the last node, the first node and the only node of the list respectively. Notice that there is no scenario corresponding to the case where the node to be deleted is not in the list. That means that the behavior of the synthesized code will be unspecified in such a case.

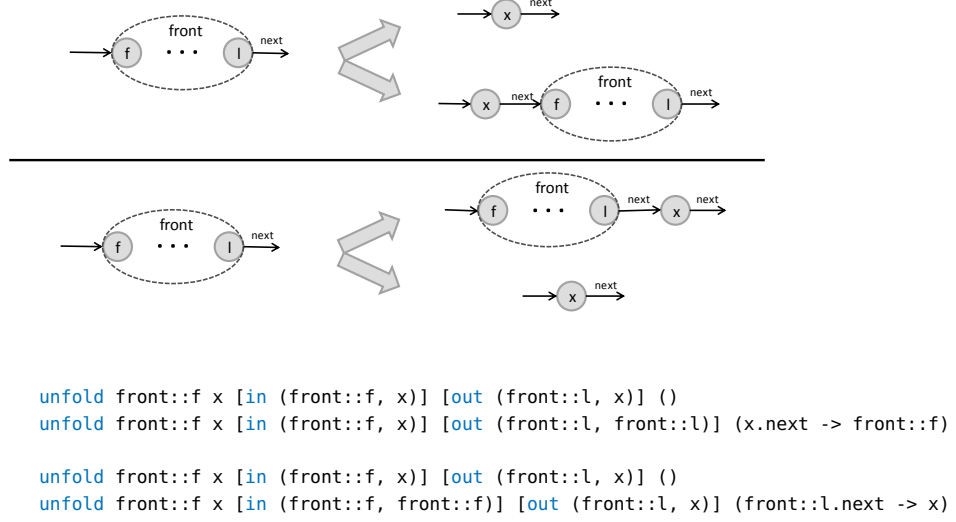


Figure 74: Two possible unfold definitions for summary node front.

The two possible unfold rules for the summary node front and their corresponding visual description are shown in Figure 74. The rule states that the summary node front either represents a single node x or a node x followed by another similar summary node front.

The loop skeleton for the linked list deletion manipulation is shown in Figure 75(a). It consists of a while loop with a set of unknown statements before the loop, in the loop body and after the loop. Given the scenarios, recursive definitions and the loop skeleton, SPT synthesizes the imperative implementation shown in Figure 75(b)

5.2 SPECIFICATION MECHANISM: STORYBOARD

In this section we present the specification mechanism used by SPT to specify, encode, and reason about the storyboard description. A storyboard comprises of a set of scenarios, a set of unfold-fold definitions, and a loop skeleton. The formalism is based on abstract interpretation, and is similar to that of TVLA; the primary difference is our treatment of summary nodes with attachment points. But before we describe the abstract interpretation, we need to define the concrete domain over which programs operate. In this concrete domain, the state of the program is defined by a fixed set of local variables and a set of memory locations (also called nodes), where each location

```

llDelete(Node head, Node y){
  Node temp1, temp2;
  ?=(2) /* h1 */
  while(**){ /* h2 */
    ??(4) /* h3 */
  }
  ?=(2) /* h4 */
}

```

(a)

```

llDelete(Node head, Node y){
  Node temp1, temp2;
  temp1 = head;
  while(temp1 != y){
    // unfold temp1;
    temp2 = temp1;
    temp1 = temp1.next;
    // fold prev;
  }
  if(temp2 == null)
    head = temp1.next;
  if(temp2 != null)
    temp2.next = temp1.next;
}

```

(b)

Figure 75: (a) The loop skeleton and (b) the synthesized implementation for linked list deletion.

can have a number of fields pointing to other memory locations. SPT assumes all nodes are of the same type. SPT also does not support the allocation of memory by a synthesized routine, so the set of nodes that the program has to reason about does not grow as the routine executes.

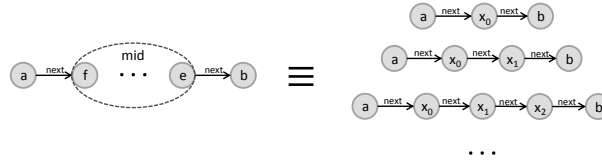


Figure 76: An abstract list representing infinite concrete lists

SCENARIOS A scenario in a storyboard describes the behavior of the manipulation using (potentially abstract) input-output data structure configurations. The storyboard in Figure 68 consists of four scenarios named s1, s2, s3, and s4. Each scenario consists of an input and output state, and can also contain optional intermediate states. Let $\mathcal{L}^\#$ represent the set of memory locations the synthesized program will operate on. Then, the state of the program is captured by two sets of predicates. First, for every variable v and location $l \in \mathcal{L}^\#$ there is a predicate $v(l)$ that indicates whether v points to location l . Then, for every field sel , there is a predicate $sel(l_1, l_2)$ that indicates whether a location l_1 has a field sel that points to location l_2 . These two sets of predicates encode a *concrete shape* which defines the instantaneous configuration of the heap at any point in the execution.

The abstract domain consists of sets of *abstract shapes*, where each abstract shape itself represents a set of concrete shapes. The abstract shapes are defined in terms of a set of locations \mathcal{L} . Each location $\text{loc} \in \mathcal{L}$ can be either a summary location or a concrete location; we use the predicate $\text{sm}(\text{loc})$ to indicate that loc is a summary location, so $\neg \text{sm}(\text{loc})$ indicates that the location is concrete. As we have stated before, a concrete location loc may serve as an *attachment point* for a summary location u , which we express with the notation $\text{loc} \in \mathcal{A}(u)$; we use the predicate $\text{apt}(\text{loc})$ to indicate the role of loc as an attachment point.

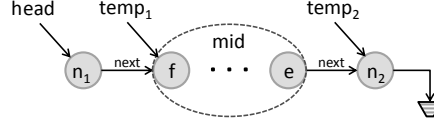


Figure 77: State configuration for a singly linked list

Example 5.2.1. The state configuration in Figure 77 is encoded as follows. The set of locations is given by $\mathcal{L} = \{n_1, \text{mid}, f, e, n_2\}$, with a summary node mid and attachment points $\mathcal{A}(\text{mid}) = \{f, e\}$. The set of program variables are $\mathcal{V} = \{\text{head}, \text{temp}_1, \text{temp}_2\}$. The variable predicates $\text{head}(n_1)$, $\text{temp}_1(f)$ and $\text{temp}_2(n_2)$ are true. The next selector predicates $\text{next}(n_1, f)$, $\text{next}(e, n_2)$ and $\text{next}(n_2, \text{null})$ are true, and $\text{next}(f, \text{mid}) = 1/2$ (similar to the 3-valued logic used in TVLA).

To make the definition of the abstract domain more formal, consider a concrete shape $S^\#$ with a set of nodes $\mathcal{L}^\#$, a selector predicate $\text{sel}^\#$ and a variable predicate $\text{var}^\#$, together with an abstract shape S with a set of nodes \mathcal{L} , selector sel and variable predicate var . We say that shape $S^\#$ is in the concretization of S ($S^\# \in \gamma(S)$) when there exists a relation $\mathcal{M} : \mathcal{L}^\# \times \mathcal{L}$ that satisfies the following conditions.

- Every node in $S^\#$ maps to some node in S and vice versa:
i.e. $\forall l_1 \in \mathcal{L}^\# \exists n_1 \in \mathcal{L} \text{ s.t. } \mathcal{M}(l_1, n_1) \text{ and } \forall n_1 \in \mathcal{L} \exists l_1 \in \mathcal{L}^\# \text{ s.t. } \mathcal{M}(l_1, n_1)$
- Nodes that do not map to summary nodes map to a single concrete node: i.e. for any $l_1 \in \mathcal{L}^\#$, if $\neg \exists n_2 \text{ s.t. } \text{sm}(n_2) \wedge \mathcal{M}(l_1, n_2)$ then $\mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_1, n_b) \Rightarrow n_a = n_b$.
- Summary nodes do not overlap: i.e.
 $\text{sm}(n_a) \wedge \mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_1, n_b) \Rightarrow (n_a = n_b \vee n_b \in \mathcal{A}(n_a))$.
- Edges between concrete nodes are preserved: i.e. given $l_1, l_2 \in \mathcal{L}^\#$ and $n_1, n_2 \in \mathcal{L}$ where $\neg \text{sm}(n_1)$ and $\neg \text{sm}(n_2)$, let $\mathcal{M}(l_1, n_1)$ and $\mathcal{M}(l_2, n_2)$; then,
 $\text{sel}^\#(l_1, l_2) \Leftrightarrow \text{sel}(n_1, n_2)$ and $\text{var}^\#(l_1) \Leftrightarrow \text{var}(n_1)$.
- Summary nodes own their associated attachment points: i.e. if $\mathcal{M}(l_1, n_2)$ and $n_2 \in \mathcal{A}(u)$ then $\mathcal{M}(l_1, u)$

- Any edge pointing to a summary node from the outside must point to one of its attachment points: i. e. let $\text{sel}^\#(l_1, l_2) \wedge \mathcal{M}(l_2, n_b) \wedge \text{sm}(n_b)$, then either $\mathcal{M}(l_1, n_b)$ or $\exists n_a \in \mathcal{A}(n_b) \text{ s.t. } \mathcal{M}(l_2, n_a)$, and for variables, $\text{var}^\#(l_2) \wedge \mathcal{M}(l_2, n_b) \wedge \text{sm}(n_b)$, then $\exists n_a \in \mathcal{A}(n_b) \text{ s.t. } \mathcal{M}(l_2, n_a)$
- Selector edges for summary nodes not originating in an attachment point are ignored: i. e. if $\text{sm}(n_a) \vee \text{sm}(n_b)$ then $\text{sel}(n_a, n_b) = 0 \vee n_a \in \mathcal{A}(n_b)$.
- Selector edges from an attachment point to its enclosing summary node will have value $1/2$: i. e. if $n_a \in \mathcal{A}(n_b)$ and $\exists l_1, l_2 \in \mathcal{L}^\# \text{ s.t. } \text{sel}^\#(l_1, l_2) \wedge \mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_2, n_b)$, and $\neg \exists n_c \text{ s.t. } \mathcal{M}(l_2, n_c) \wedge \neg \text{sm}(n_c)$, then $\text{sel}(n_a, n_b) = 1/2$.

In shape analysis it is common to use 3-valued logic to represent the values of selector and variable predicates in abstract shapes. However, notice that the rules above specifically require us to ignore most selector edges involving summary nodes. The restrictions imply that the only selector edges that will potentially have value equal to $1/2$ are edges from an attachment point to its corresponding summary node; that is why we have $\text{next}(f, \text{mid}) = 1/2$ in the earlier example. As we shall see in the next section, the transition rules in the abstract semantics are defined in such a way that if the algorithm under analysis ever tries to dereference a field corresponding to one of these half edges, it will transition into an error state. This makes the analysis simpler at the expense of added imprecision, but our analysis compensates for this imprecision by relying on the unfold predicates to *materialize*[102] summary nodes.

UNFOLD/FOLD DEFINITIONS In order for SPT to precisely reason about the abstract shapes, SPT needs some more information about the structure of the abstract nodes. A user provides this additional information using *unfold* and *fold* definitions. An *unfold* definition inductively defines the structure of concrete nodes that a corresponding abstract node represents. SPT uses the *unfold* definitions to concretize abstract nodes during its analysis. The *fold* predicate performs the analogous reverse operation, i.e. it abstracts the set of concrete nodes back to an abstract node. An *unfold* definition (resp. *fold*) is defined using a 5-tuple consisting of an abstract node, a concrete node, a set of in predicates, a set of out predicates, and a set of constraints that should hold after the *unfold* operation. The semantics of *unfold* and *fold* predicates is defined in Section 5.3.2.

5.3 HYPOTHESIS SPACE

5.3.1 A Simple Pointer Language L_p

The hypothesis space that defines the set of programs explored by SPT is defined using a combination of domain-specific language of pointers L_p , whose syntax is shown in Figure 78, and a user input of loop skeleton. The set of statements in L_p consists of pointer assignments, conditional pointer assignments, while loops, unfold, and fold statements. The concrete semantics of the language are as expected. There are three key restrictions in L_p : (i) only one dereferencing of pointers is allowed in pointer assignments, (ii) no memory allocation of new nodes is allowed, and (ii) only one level of Boolean operators is allowed in conditionals. These restrictions do constrain the family of programs that SPT can synthesize, but as we will see in Section 5.6, it is still rich enough to express a large family of manipulations over linked lists and binary search trees.

$$\begin{aligned}
 \text{Ptr Expr } e &:= v \mid v.\text{sel} \mid \text{null} \\
 \text{Bool Expr } b &:= \text{not } b \mid e_0 \text{ op}_c e_1 \mid b_0 \text{ op}_b b_1 \\
 \text{Comp Op op}_c &:= == \mid < \mid > \mid \leq \mid \geq \mid != \\
 \text{Bool Op op}_b &:= \text{and} \mid \text{or} \\
 \text{Stmt } s &:= e_0 = e_1 \mid s_0; s_1 \mid \text{while}(b) s \\
 &\quad \mid \text{if}(b) s \mid \text{unfold } v \mid \text{fold } v \\
 \text{Func Def. } p &:= f(e_1, \dots, e_n) \{ s \}
 \end{aligned}$$
Figure 78: The syntax for a simple pointer language L_p .

LOOP SKELETON The hypothesis space is parameterized by a loop skeleton provided as part of the storyboard. The loop skeleton helps SPT to constrain and guide the search space of possible implementations. The body of a loop skeleton is defined using the following simple grammar.

$$\text{Unknown Stmt Block } s := ? = (n) \mid ??(n) \mid \text{while}(**) s \mid s_0; s_1$$

The unknown statment block $? = (n)$ defines a block of size n of conditional statements, whereas $??(n)$ defines a block of size n of pointer assignment statements. The while statment $\text{while}(**) \{s\}$ defines a while loop with an unknown loop condition and loop body s .

5.3.2 Abstract Semantics of L_p

Having defined the pointer language L_p and the structure of the abstract domain, we now describe the abstract semantics of L_p , since SPT performs analysis over the abstract domain for synthesizing programs in L_p . Figures 79 and 80 show respectively the abstract semantics of L_p statements and conditionals. The figures also show the formal definitions of the transition rules associated with each construct. The transition rules relate the state before the transition—the pre-state represented with non-primed predicates—with the post-state represented with primed predicates. It is assumed that the values of all other predicates not mentioned in the transition rule remain unchanged. The abstract semantics for while statement is the standard fixpoint computation by performing iterative abstract execution of statements and conditionals of the loop body.

Statement	Abstract Semantics
$x = \text{null}$	$\forall l \in \mathcal{L} : x'(l) = 0$
$x = t$	$\forall l \in \mathcal{L} : x'(l) = t(l)$
$x = t.\text{sel}$	$\text{assert } \neg \exists l_1, l_2 \in \mathcal{L} : t(l_1) \wedge \text{sel}(l_1, l_2) \wedge \text{sm}(l_2)$ $\forall l \in \mathcal{L} : x'(l) = \exists l_1 t(l_1) \wedge \text{sel}(l_1, l)$
$x.\text{sel} = \text{null}$	$\text{assert } \neg \exists l_1, l_2 \in \mathcal{L} : x(l_1) \wedge \text{sel}(l_1, l_2) \wedge \text{sm}(l_2)$ $\forall l_1, l_2 \in \mathcal{L} : \text{sel}'(l_1, l_2) = \neg x(l_1) \wedge \text{sel}(l_1, l_2)$
$x.\text{sel} = t$	$\forall l_1, l_2 \in \mathcal{L} : \text{sel}'(l_1, l_2) = \text{sel}(l_1, l_2) \vee (x(l_1) \wedge t(l_2))$
unfold x	$\text{unfoldPred}(E, M, C) \wedge x(E) \implies$ $((\forall l_1 \xrightarrow{\text{in}} l_2 \in M : \text{fresh}(l_2) \wedge$ $\quad \forall v \in \mathcal{V} : v'(l_2) = v(l_1) \wedge v'(l_1) = 0 \wedge$ $\quad \forall l \in \mathcal{L} : \text{sel}'(l, l_2) = \text{sel}(l, l_1) \wedge \text{sel}'(l, l_1) = 0) \wedge$ $(\forall l_1 \xrightarrow{\text{out}} l_2 \in M : \text{fresh}(l_2) \wedge$ $\quad \forall l \in \mathcal{L} : \text{sel}'(l_2, l) = \text{sel}(l_1, l) \wedge \text{sel}'(l_1, l) = 0) \wedge C)$
fold x	$\text{foldPred}(E, M, C) \wedge (x(E) \wedge C) \implies$ $((\forall l_1 \xrightarrow{\text{in}} l_2 \in M : \text{fresh}(l_2) \wedge$ $\quad \forall v \in \mathcal{V} : v'(l_2) = v(l_1) \wedge v'(l_1) = 0 \wedge$ $\quad \forall l \in \mathcal{L} : \text{sel}'(l, l_2) = \text{sel}(l, l_1) \wedge \text{sel}'(l, l_1) = 0) \wedge$ $(\forall l_1 \xrightarrow{\text{out}} l_2 \in M : \text{fresh}(l_2) \wedge$ $\quad \forall l \in \mathcal{L} : \text{sel}'(l_2, l) = \text{sel}(l_1, l) \wedge \text{sel}'(l_1, l) = 0))$

Figure 79: Abstract semantics of L_p statements.

The rules follow a convention from shape analysis to assume that every statement of the form $\text{exp} = t$ is preceded by a statement of the form $\text{exp} = \text{null}$, where exp is either x or $x.\text{sel}$. This assumption simplifies the rules because the transition rule for $\text{exp} = t$ does not have to worry about destroying the value previously stored at exp [102]. The other important observation about the rules is the use of assertions for the two rules that do field dereferences. These assertions ensure that the system will transition into an error state when it tries to

Conditional	Abstract Semantics
$x == \text{null}$	$\forall l \in \mathcal{L} : \neg x(l)$
$x != \text{null}$	$\exists l \in \mathcal{L} : x(l)$
$x == t$	$\forall l \in \mathcal{L} : x(l) \iff t(l)$
$x.\text{data} > t.\text{data}$	$\forall l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \text{gt}(\text{data}, l_1, l_2)$
$x.\text{data} \geq t.\text{data}$	$\forall l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \text{gte}(\text{data}, l_1, l_2)$
$x.\text{data} == t.\text{data}$	$\forall l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \text{eq}(\text{data}, l_1, l_2)$
$x.\text{data} != t.\text{data}$	$\forall l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \neg \text{eq}(\text{data}, l_1, l_2)$

Figure 80: Abstract semantics of L_p conditionals.

dereference a selector that points to a summary node, which in turn guarantees that 1/2 values corresponding to these selector predicates will not propagate through the representation.

The abstract semantics for the class of conditionals is shown in Figure 80. It can be noted that although the assignment statements in our target language of programs ignore data fields of the data-structure (.data as opposed to .sel), the conditionals can reason about the data constraints. We store data predicates using gt, gte, eq etc., which encode data constraints over the data values of locations. We do not consider conditionals involving selector dereferencing of variables, e.g. of the form $x.\text{next} == \text{null}$, as they can be reduced into a conditional of the form $y == \text{null}$ where the variable y is first assigned by the statement $y = x.\text{next}$.

FOLD/UNFOLD SEMANTICS The unfold operation is described with a triple $\text{unfoldPred}(E, M, C)$. The first argument E is called the enabling node, and it represents the summary node that is being expanded. The transition rule for the unfold x statement performs the unfold operation only if the variable x points to the enabling node E . The second argument M is the location mapping $M : \text{loc} \rightarrow \text{loc}$, which describes how nodes before expansion relate to nodes after expansion. There are two kinds of location mappings in M : $\xrightarrow{\text{in}}$ mappings and $\xrightarrow{\text{out}}$ mappings. An $\xrightarrow{\text{in}}$ mapping maps a location loc_1 to loc_2 such that all variables and selector edges pointing to loc_1 in the pre-state should point to loc_2 in the post-state. An $\xrightarrow{\text{out}}$ mapping maps a location loc_1 to loc_2 such that all outgoing selector edges from loc_1 in the pre-state emanate from loc_2 in the post-state.

Finally, the description of unfold also includes a set of constraints C . These constraints describe how the new nodes will be connected together, and are asserted to hold in the post-state after unfolding. The transition rule for fold x statement works similarly to the unfold rule. The difference is that the fold operations are enabled only if the constraints C are also satisfied by the state configuration in addition to the requirement of x pointing to the enabling node E . The unfold

and fold predicate definitions on the summary node *mid* are shown in Figure 69.

Another set of fold-unfold examples for the binary search tree (bst) case studies is shown in Figure 81. The goal of bst search is to search for a value x in the tree where r represents its root. The bst search (contains) manipulation assumes that the value x always exists in the tree. The three cases of bst search (contains) unfold are: i) $x < y.val$, ii) $x = y.val$ and iii) $x > y.val$ as shown in Fig 81(a), where y denotes the root node of the subtree being unfolded. The unfold definition for the more general case of bst search is shown in Figure 81(b). The tree summary nodes labeled *stuff* are given without any unfold rules, which means they cannot be materialized, so the verifier will not be able to reason about any implementation that tries to visit them. In this way, the unfold rule is providing algorithmic insights, telling the synthesizer that a given region of the tree should not be visited or manipulated.

One important thing to note about unfold is that a given shape can be expanded in many different ways, as illustrated in Figure 69. This is expressed by having multiple *unfoldPred* triples with the same enabling node. As a consequence, every abstract shape in the pre-state of an unfold operation may be expanded into a *set* of abstract shapes. This expansion allows the analysis to maintain precision, but having to represent sets of abstract shapes in the abstract interpreter will pose an interesting challenge when we turn the problem into a constraint satisfaction problem.

Another very important aspect about unfold is that the presence of unfold changes the concretization relation γ between abstract and concrete shapes. In the absence of unfold, any arbitrary set of concrete nodes can be mapped to a summary node by the relation \mathcal{M} described in the previous section, but unfold has the effect of placing some structural constraints on the set of nodes that can be mapped to a summary node. This is a result of the requirement that unfold correspond to skip in the concrete domain; this means that if a given abstract shape S can be transformed by unfold into any shape in the set $\{S_i\}$, then the set of concrete shapes $\gamma(S)$ should equal $\bigcup_i \gamma(S_i)$. So we can refine the earlier definition of the concretization function to say that $S^\# \in \gamma(S)$ if it satisfies the requirements stated before *and* if $S^\# \in \bigcup_i \gamma(S_i)$, where S_i are all the shapes that can be derived from S through the application of unfold. The next section elaborates on how this definition relates to the instrumentation predicates used by TVLA to describe the structure of summary nodes.

RELATIONSHIP WITH TVLA In order to understand some of the more subtle aspects of our formalism, it is useful to understand how

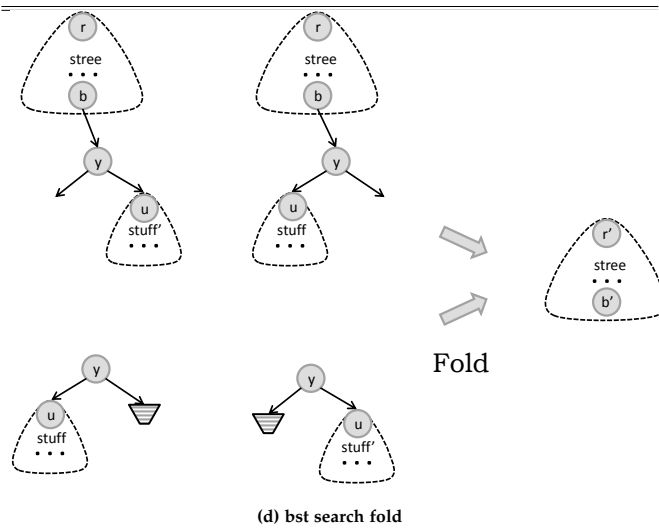
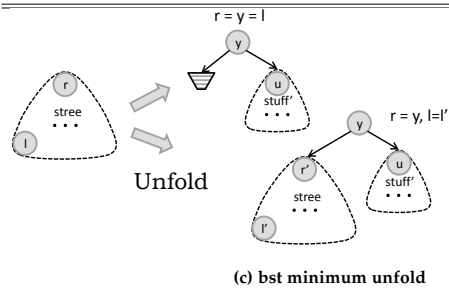
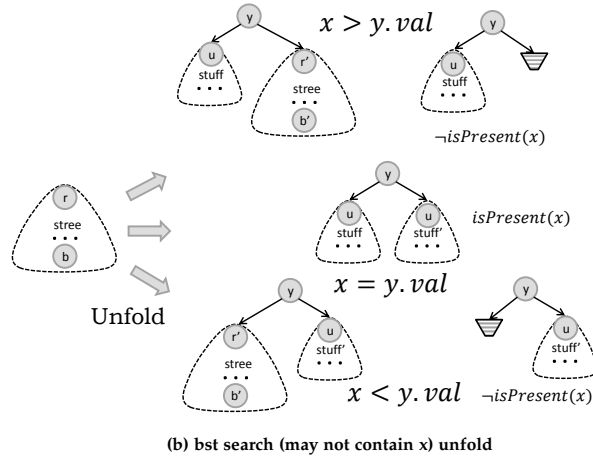
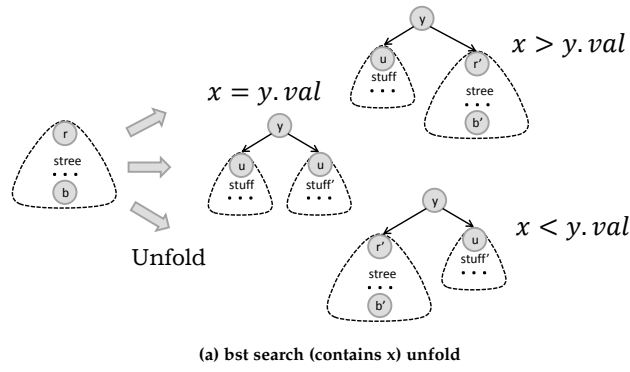


Figure 81: Unfold and fold operations for different data structure manipulations

it relates to the formalism in TVLA [77]. The two most apparent differences between the two formalisms are the use of attachment points as part of summary nodes and the use of fold and unfold.

In our system, the unfold definition serves two purposes: it provides a mechanism to convey structural properties of summary nodes, and it is also used to *materialize* summary nodes, i.e. to produce a set of more refined shapes that together represent the same set of concrete configurations as the original configuration. In TVLA, by contrast, structural properties are described through *instrumentation predicates*. These predicates are also used for materialization, but not directly; instead, a *focus* operation first expands a summary node into a set of possible shapes, and then a *coerce* operation uses the instrumentation predicates to refine the new shapes and to remove those that do not satisfy the required structural properties.

One can understand the unfold rules in our framework as a specialized way of describing instrumentation predicates. For example, from the unfold rule for mid we can derive the following instrumentation predicate:

$$\text{isMid}(f, e) = (f = e) \vee \exists f' (f.\text{next} = f' \wedge \text{isMid}(f', e)) \quad (3)$$

The predicate `isMid` encodes that every node in `mid` is reachable from the front attachment point, and therefore that the sub-list between `f` and `e` is acyclic. When we say that a summary node satisfies this predicate, it means that the summary node can only represent sets of nodes where we can find two nodes `f` and `e` that satisfy the predicate. The unfold operation induces this predicate because we want the effect of unfold in the abstract domain to be equivalent to the effect of skip in the concrete domain, and this will only be true if the summary nodes satisfy this predicate. It is interesting to see, however, that the structure of the predicate is very close to the structure of the unfold rules, with the attachment points serving as convenient parameters to the predicate.

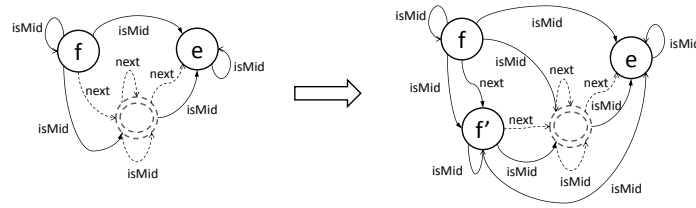


Figure 82: unfold in 3-valued shape analysis

Given such an instrumentation predicate, we can map our summary nodes with attachment points to a shape in TVLA; for example, Figure 82 shows how `mid` would look like as a shape in TVLA. Our unfold operation is equivalent to first applying materialization to partially concretize the summary node and then coerce to remove invalid shapes obtained after materialization.

The use of `unfold` in place of instrumentation predicates and the use of attachment points both have a number of advantages for the purpose of our framework. The first important advantage is that it simplifies the transition rules, because it eliminates the need to track instrumentation predicates. Another important benefit of using summary nodes with attachment points is that it simplifies the graphical representations, as can be readily appreciated by comparing Figure 82 with Figure 69. One clear difference between the two representations is that Figure 82 includes a number of selector edges with value $1/2$ which are not present in the diagrams in Figure 69. This is partly by convention, since we omit from our representation the selector edges within summary nodes, and partly because the assumptions in Section 5.2 ensure that references from concrete to summary nodes always point to their attachment points.

Compared to TVLA, our formalism allows for a simpler analysis and more concise graphical representations. The downside, of course, is reduced expressive power. If one were trying to verify arbitrary programs, the shortcuts taken by our system would make the analysis impractical—it would be too easy for the user to write a program that could not be verified because it violated one of our assumptions. On the other hand, as Vechev *et al.* have pointed out [130], the combination of synthesis with abstract interpretation means that the synthesizer can work around the limitations of the abstract interpreter by producing programs that are easy to verify. In our case, the way the synthesizer works around the limitations imposed by our assumption is by using `unfold` statements to materialize nodes at the right time and ensure that the assumption is never violated.

5.4 CONSTRAINT-BASED SYNTHESIS ALGORITHM

Having defined our basic abstract interpretation framework, we can now describe how we frame the synthesis problem as a set of constraints whose solution will describe the implementation we are looking for. The starting point for this process is the loop skeleton together with the scenarios.

The loop skeleton Sk constitutes the control flow graph of the implementation; unlike a CFG, however, the vertices are not just basic blocks because they can contain conditional assignments. More formally, we represent the control flow sketch Sk as a directed graph $Sk = G(V, E)$ where V represents a set of blocks of code which can either be a sequence of conditional assignment statements or a conditional (for loop exit conditions); E in turn represents potential transitions between these blocks of code. By convention, we say that vertex v_0 is the entry point of the graph node and v_N is the exit point; neither the entry nor the exit blocks contain any code. As for scenar-

ios, each of them is represented as a pair of input and output states $S_i = (\text{In}_i, \text{Out}_i)$.

We use standard techniques to encode each block of unknown statements as a parameterized function $F_i(\text{in}, c_i)$, where the parameter c_i selects which block of code out of the set of possible blocks of code F_i will represent. The inputs and outputs of F_i are elements of the abstract domain, which happen to be sets of shapes. Now, the set of possible sequences of conditional assignments is infinite, but bounding the maximum length of the statement sequence makes the set finite as there are a finite number of assignment statements and conditionals of the form shown in Figure 79 and Figure 80 respectively.

The goal of the synthesis process is to find values of c_i such that for each scenario S_k , the least fixed point solution to the following equation satisfies $t_N = \text{Out}_k$:

$$t_0 = \text{In}_k \wedge \forall v_i \in (V \setminus v_0) \quad t_i = F_i\left(\bigcup_{j \in \text{pred}(v_i)} t_j, c_i\right) \quad (4)$$

The function $\text{pred}(v_i)$ in the equation above indicates the set of predecessors of node v_i . The equation above is fairly simple, but two challenges prevent us from solving it directly with an SMT solver. First, the equation above requires us to find not just any solution, but the least fixed point solution. Additionally, the t_i in the equation above are elements in the abstract domain, which is composed of sets of shapes. Such sets can get quite big given the nature of our domain, so representing them naively in an SMT solver is infeasible. In the rest of this section, we describe how our framework addresses both of these problems.

5.4.1 Computing Least Fixed Points

In order to find a least fixed point solution to Equation 4, we start from the assumption that an iterative method can reach a least fixed point after visiting each vertex in S_k at most K times. Now, let P^K be the set of all paths in S_k that visit each vertex at most K times. For each path $p_i \in P^K$, we can define a path transformer $p_i(\text{in}, \vec{C})$ which is the composition of all the transfer functions $F_t(\text{in}, c_t)$ of all vertices v_t in the path, where $\vec{C} = [c_0, \dots, c_N]$. Then, the least fixed point solution to the value of t_N in Equation 4 will be given by

$$\bigcup_{p_i \in P^K} p_i(\text{In}_k, \vec{C})$$

The equivalence follows from the distributivity of F (i.e. the fact that $F(a \cup b) = F(a) \cup F(b)$). Given a solution to the equation above, it is easy to check the assumption of K convergence by simply checking the solution against Equation 4.

5.4.2 Dealing with sets of abstract shapes

As discussed before, in order to feed the constraints into a solver, we would like to avoid having to reason about sets of abstract shapes. Our strategy will be as follows. First, from a transfer function F , we can define a function F^j that returns a singleton set containing the j^{th} element of the set returned by F , or an empty set if there is no j^{th} element. Thus, $F(a) = \cup_j F^j(a)$, where each F^j produces either singleton or empty sets.

The strategy even works when composing functions thanks to the distributivity of F . Because of this property, if we have a function $F(a)$ and a function $T(a)$, then the composition $F(T(a))$ can be computed as $\cup_{i,j} F^j(T^i(a))$.

In the case of our transfer functions $F_i(\text{in}, c)$, it is relatively easy to derive the functions $F_i^j(\text{in}, c)$. For example, one of the statements that can produce multiple shapes from a single one is `unfold`, so if we want a function to return only the j^{th} shape produced by `unfold`, we only use the j^{th} `unfoldPred` instead of using all of them. Composing the transfer functions for each block into path expressions, we get a path expression $p_i^{\vec{j}}(\text{In}_k, \vec{C})$, where instead of composing functions $F_i(\text{in}, c_i)$ in the path, we compose functions $F_i^j(\text{in}, c_i)$. With this transformation, the constraint we need to solve becomes:

$$\exists \vec{C} \quad (\text{Out}_k = \bigcup_{\vec{j}} \bigcup_{p_i \in P} p_i^{\vec{j}}(\text{In}_k, \vec{C}))$$

The set unions in the equation above can be turned into universal quantifiers to produce the following equation:

$$\exists \vec{C} \quad (\forall_{\vec{j}} \forall_{p_i \in P} p_i^{\vec{j}}(\text{In}_k, \vec{C}) \in \text{Out}_k \quad \wedge \quad \exists_{\vec{j}} \exists_{p_i \in P} p_i^{\vec{j}}(\text{In}_k, \vec{C}) = \text{Out}_k)$$

The universally quantified part of the equation forces the union of the path transformers to be a subset of Out_k , while the existentially quantified part of the constraint ensures that the singleton Out_k is a subset of the union of the path transformers. The equation above no longer has to reason about sets with more than one element, but in exchange for that, it has to cope with $\exists \forall$ quantifier alternation. However, the `SKETCH` system is very effective in dealing with such doubly quantified formulas, so our system actually translates the above equation into a sketch and uses the `SKETCH` solver to find a solution to all the unknowns.

5.4.3 Termination

Equation 4 only ensures partial correctness, so there is no termination guarantee for the synthesized implementation. However, we have

found that adding a few additional constraints was enough to guarantee terminating solutions for all the examples we examined. The additional constraint was to require that for every state reachable inside any loop in the program it is possible to satisfy the loop exit condition in an additional K loop unrollings. If the unfold and fold predicates satisfy well-formedness constraints [93] and with the restriction of using only one unfold and fold operation per loop, the framework can guarantee termination of the synthesized implementation using a reasoning similar to [20]. This restriction works for data-structure manipulations that perform a single pass over the data-structure.

5.5 USER INTERACTION MODEL

A storyboard specification—the scenarios, the loop skeleton and the fold and unfold definitions—comprise a partial specification of the desired manipulation. For example, the list reverse storyboard is a partial specification because the abstraction does not define the relationship between `mid` in the input and `mid` in the output. In this case, asking the synthesizer to produce a solution with a small number of statements is sufficient to ensure the correct answer, but sometimes the user may have to provide the system with additional information. In keeping with the PBE model, this additional information usually takes the form of additional scenarios with concrete examples, but a user can also provide intermediate state configurations, add predicates in scenarios or provide a more detailed implementation sketch in place of the simple loop skeletons. The strength of our synthesis approach is that it can combine these different constraints into a concrete implementation. Moreover, as shown in Section 5.6, the constraints imposed by the storyboards are strong enough that in the few cases where the specification has to be strengthened, it only takes a few additional concrete scenarios or an intermediate state configuration to guide the framework to synthesize the correct implementation.

5.6 EXPERIMENTAL EVALUATION

In our experiments with the Storyboard framework, the key questions we explored were: i) how does it scale for synthesizing reasonably complex data-structure manipulations, ii) how much additional information is required to be provided in the storyboards other than just the input-output scenarios, iii) how much having abstraction in the scenarios help and iv) can we use it to synthesize user-defined data-structure manipulations.

Table 3 presents the experimental results of the case studies that we performed with the framework. The experiments were run on an Intel Core-i7 1.87GHz CPU with 4GB of RAM. The first column in the table

shows the name of the manipulation where ll refers to singly linked-list, dll refers to doubly linked-list and bst refers to binary search tree. The table presents the details about number of scenarios used in storyboard, the total time it took to synthesize the implementation, the number of clauses in the SAT translation of the constraints and the memory used. Even though this is not the most efficient encoding of the constraints, we were able to synthesize all the manipulations in less than 6 minutes each using less than 1 GB of memory.

For reducing the search space in our experiments, we had to restrict the usage of unfold and fold statements to at most once at the beginning and end locations inside the loop respectively; which works well for single-pass algorithms. The case studies with the *Interm* column marked yes required some additional intermediate state configuration, e.g. in the storyboard for linked list insertion, we also had to provide intermediate state configuration after the loop body in the skeleton for helping the synthesizer to converge faster. These intermediate configurations present a natural interface for providing hints about the manipulation. Some case studies also required composition of storyboards (marked with a *), e.g. the bst-deletion storyboard required composition of bst-search and bst-find-min storyboards.

In some cases like bst-deletion, we found that the abstract input-output specification was too weak and allowed many undesired solutions; but it was easily fixed by providing a couple of concrete input-output bst instances. We also performed an experiment where we only provided concrete examples for these manipulations, the synthesizer either generated an undesired solution or got timed out and never converged for most of these case studies. This experiment shows the ability of abstract input-output examples to prune a big search space of undesired programs.

We have used our framework to synthesize manipulations for a complicated real-world AIG data structure. AIG is a DAG that encodes the structural implementation of the logical functionality of a circuit [86] using two-input AND gates and inverters. Each internal node of AIG represents an *and* gate and has two parents corresponding to the two inputs of the gate. The child list information for each node is overlayed inside the node itself by keeping a pointer to the first child and pointers to the sibling nodes. Even though our framework currently can not synthesize arbitrary graph manipulations, we exploit the listness property of the child lists of AIG nodes for synthesizing its manipulations.

Even for complicated data-structures like red-black trees, where it is difficult to draw a simple storyboard expressing the complex invariants about the data-structure, we found the storyboard framework helpful for synthesizing fragments of low-level code of different cases individually and then manually composing the synthesized code to obtain a complete implementation. Figure 83 shows the story-

board for red-black tree `fixInvariant` method (part of the insertion procedure) that we obtained from an online lecture note [4]. We used the framework to synthesize low-level code for the four cases, which were then easily composed manually inside the complete algorithm. Our tool and more details about the case studies can be found at the storyboard website [5].

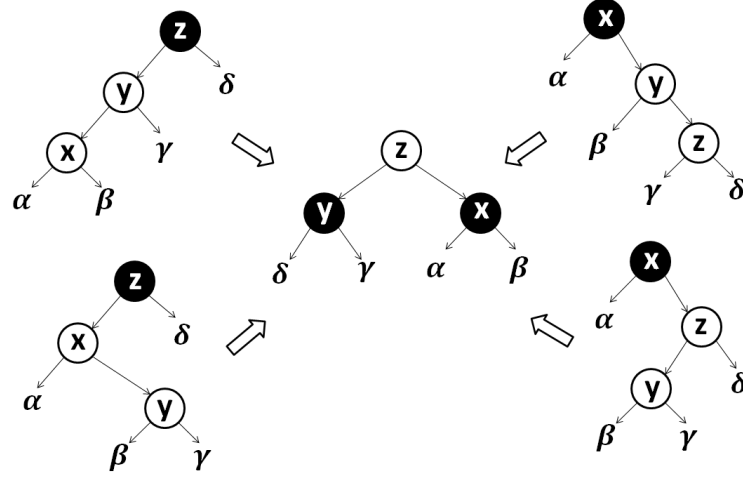


Figure 83: Red-black tree `fixInvariant` storyboard.

Manipulation	#Scens	Time	#Clauses	Memory	Loops	Interm
ll-insertion	4	2m9s	1.99M	0.75GB	1	Yes
ll-deletion	4	1m48s	1.88M	0.54GB	1	Yes
ll-reversal	4	1m49s	1.3M	0.35GB	1	No
ll-find-last	4	0m56s	1.02M	0.29GB	1	No
ll-swap-first-last	4	4m18s	1.08M	0.31GB	1	Yes
dll-traversal	4	1m58s	1.72M	0.88GB	1	No
dll-reversal	4	3m47s	2.04M	0.49GB	1	No
bst-search(contains)	1	1m02s	0.62M	0.37GB	1	No
bst-search	1	6m07s	0.77M	0.45GB	1	No
bst-find-min	1	0m58s	0.63M	0.18GB	1	No
bst-find-max	1	0m23s	0.57M	0.16GB	1	No
bst-left-rotate	3	3m18s	1.41M	0.50GB	0	No
bst-right-rotate	3	3m15s	1.47M	0.43GB	0	No
bst-insertion*	3	1m52s	1.04M	0.46GB	1	Yes
bst-deletion*	6	3m13s	0.63M	0.62GB	2	Yes
aig-insertion*	4	1m04s	0.17M	0.31GB	1	Yes

Table 3: Experimental results for case studies

RELATED WORK

This thesis presents new automated program synthesis techniques for problems from various domains ranging from Visual Programming (Storyboard Programming) to End-user Programming (FlashFill) and Computer-aided Education (AutoProf). In addition to related work on Program Synthesis, we also present related work from the fields of Programming by Examples and Demonstrations, Computer-aided Education and Automated Grading, Automated Program Repair, Automated Database Query Synthesis, and the user of Machine Learning for Program Synthesis.

6.1 PROGRAM SYNTHESIS

Software synthesis has been an active research area at least since the early 80s when Waldinger and Manna [82, 83] did seminal work on deductive synthesis. In this work, a program was extracted from the proof of correctness of the specification. A more algorithmic approach to synthesis was pioneered by Pnueli and Rosner in the context of finite state controllers [96]. More recently, Program synthesis has seen a renewed interest and has been used for many applications such as synthesis of efficient low-level code [118, 74], data structure manipulations [110], inference of efficient synchronization in concurrent programs [130], snippets of excel macros [56, 108], relational data representations [60, 61], protocol synthesis [128], and angelic programming [17]. A comprehensive survey on various program synthesis techniques and different intention mechanisms for specification can be found in [48].

One of the more recent successful synthesis system, SKETCH [118, 117], takes a partial program and a reference implementation as input and uses a constraint-based CEGIS algorithm to synthesize the completion of the partial program that is functionally equivalent to the reference implementation. We enhance the CEGIS algorithm with abstraction interpretation based semantics of shapes for the Storyboard Programming system, and with minimize constraints to obtain the CEGISMIN algorithm for the AutoProf system. In general, the template of the desired program as well as the reference specification is unknown and puts a considerable burden on programmers to provide them. For the Storyboard Programming system, we use abstract and concrete input-output examples as the specification for the data structure manipulation, and use a simplified loop skeleton grammar to generate templates. For the AutoProf system, we use the student

solution rewritten using the error model as the template program and teacher’s solution as the reference implementation.

ABSTRACT INTERPRETATION BASED PROGRAM SYNTHESIS The idea of using abstract interpretation for synthesis was recently introduced by Vechev, Yahav, and Yorsh [130], as a follow up to earlier work on synthesis of concurrent data-structures [129]. Their system is designed to synthesize efficient synchronization for concurrent programs, and is very different from SPT, both in its scope and in the algorithms it uses. Unlike their system, SPT is based on a more general constraint-based approach that allows us to handle extremely large search spaces with no apparent structure.

The idea of using a constraint-based approach for abstract interpretation was previously introduced by Gulwani *et al.* [52]. Recently, similar techniques have been extended to synthesize invariants [53] and even complete programs [122]. Some important distinctions between this work and SPT are the use of storyboards to capture insights, as well as our path-based representation of the constraints to support a very large and complex abstract domain. The idea of using a sketch to define the structure of the implementation was adapted from the original work on sketch based synthesis [119]. The idea was originally applied to the domain of bit-stream manipulations [118], such as ciphers and error correction codes, and has been applied more recently to scientific programs [119] and concurrent data-structures [120]. Although SKETCH can synthesize some of the data-structure manipulations, it requires the programmer to provide detailed sketches and only provides bounded guarantees for the synthesized implementation. Additionally, writing specifications for data-structure manipulations tend to be harder, because they have to be written as tricky test harnesses.

Recent work in data representation synthesis [59] automatically synthesizes efficient data-structure representations for a given set of data usage patterns. The representations are built from a library of data-structure building-blocks and support only a fixed set of common interface methods. SPT supports the implementation of more general data-structure manipulations, such as the list reverse example. The price of the generality is a more involved interaction model compared with the push-button interface provided by that system. PINS [123] introduced the idea of focusing on individual paths when generating constraints. Their approach does not build upon abstract interpretation as ours and SPT lets the synthesizer select interesting paths automatically using the CEGIS algorithm unlike a heuristic technique used by PINS. Gulwani *et al.* [54, 66] have proposed component-based synthesis techniques for synthesizing tricky (but loop-free) code snippets from a given multi-set of components. These

techniques are not applicable in our setting as we deal with loopy programs.

6.2 PROGRAMMING BY EXAMPLE AND DEMONSTRATIONS

Shaw [105] developed a framework for learning restricted Lisp programs from a single input/output. The framework is not for general programs and is also not guaranteed to learn the correct program. Pygmalion [115] was one of the first successful programming by demonstration systems. The programmer provided concrete execution of the program on a concrete example with the help of icons and the system inferred some recursive program from the example. Tinker [79], aimed at beginning programmers, lets one write Lisp programs by providing Lisp expressions or mouse inputs to handle the execution on concrete examples of input data. These concrete program executions are then generalized to symbolic executions and in the process ambiguities were resolved by asking the programmer for disambiguations. These systems alleviate somewhat the problem for programmer to worry about abstract inputs but they still require the programmer to know how the program is supposed to execute on concrete inputs.

The Storyboard Programming system is different from previous works in programming by example (or demonstration) as it requires no concrete program executions on the example inputs from the user and works with abstract examples (an infinite number of concrete) input-output examples. Moreover, SPT requires a loop skeleton of the program to be provided by the programmer to structure the program search space and not let the synthesizer synthesize arbitrary programs. This idea of providing programmer's insights in a multimodal form using concrete examples, abstract examples, and loop skeleton helps rule out a large subset of undesirable programs.

Our work in the FlashFill Programming by Example system learns semantic string transformations from input-output examples. The key idea that distinguishes our work from all previous work in PBE systems is the idea of defining the hypothesis space using a custom-designed expressive domain-specific language that can express most semantic transformations that end-users need to perform, but at the same time is efficiently learnable. This domain-specific language allows us to design a polynomial time synthesis algorithm for efficiently learning all expressions in the language that are consistent with the examples. Our synthesis algorithm is sound and complete and we use ranking techniques to learn the desired programs from very few examples.

VISUAL PROGRAMMING USING EXAMPLES The systems using graphics for aiding programming, debugging, and program understanding

have been an intriguing research area from a very long time. Myers [90] classifies these systems on the basis of three broad categories: Visual Programming (with Program Visualizations), Programming by Example, and interactive/batch systems. Visual Programming refers to systems that allow programmers to specify program computations graphically whereas Program visualization is used for graphically visualizing data structures at run-time for debugging purposes. Programming by Example approach uses a finite set of input-output pairs and tries to infer a program that conforms to those examples.

AMBIT/G [25] was one of the earliest efforts for representing data and programs as predefined pictures and using a pattern matching language to execute them. Grail [35] could compile flow chart programs directly to executable code. Some systems like Shaw's [105] were proposed that could learn a restricted set of programs from input/output pairs. Programming visualization systems [21] [89] were also developed to display runtime data structure information for debugging of the programs. Several other systems like Pygmalion [115] and Thinglab [18] were developed to help programmer define computations pictorially. Programming Languages like Visual Basic also allowed programmers to write GUI applications from general sub-components using drag-and-drop techniques.

Grail [35] was one of the earliest systems that compiled flowcharts to executable code. The AMBIT/G [25] language represented both programs and the data as graphs. Then the pictorial program was pattern matched for its execution. The framework was used to describe list-structure garbage collection program and reduction-analysis string parser. Even though these approaches alleviate the problem of writing code by letting the programmers use static predefined pictures but the burden of figuring out the exact sequence of operations still lies with the programmer. Also attempting to capture dynamic transformations through static diagrams makes the resulting programs much difficult.

Sketchpad [125] is a seminal work that lead to a whole new field of human-computer interaction, and is considered a great breakthrough for the computer graphics research. In this framework, a programmer used sketched geometrical object shapes like straight lines, arcs etc. using a *light pen*. The programmer could also express constraints on these shapes to get regular geometrical objects on the screen and used graphical buttons for providing options like copying etc. Even though this was a revolutionary work, this did not cater to the general purpose programming purposes.

Thinkpad [99] system combined some ideas from programming by example, constrained-based systems, and graphical programming frameworks. It used data abstracts to let user draw data structures graphically and use constraints to specify data structure invariants. The programmer could then manipulate these graphical abstractions

and perform an execution of the program on some example input. This system provided a platform for programmer to program pictorially but still the problem of reasoning about the precise execution of the program remained.

Most of these systems did not attempt to exploit the semantic information present in the pictures; rather they used them only syntactically as a means of communication between the programmer and the machine. SPT combines ideas from visual programming, programming by example, and software synthesis research. It lets programmer specify graphical specifications for input-output pairs (potentially infinite number of them). The Storyboard Programming system is different from the previous work in visual programming as it does not require programmers to provide a pictorial execution of the program. Only the input and output pictures are required which are much easier to reason about rather than different executions of the complete program.

TEXT-EDITING SYSTEMS USING DEMONSTRATIONS AND EXAMPLES:

Nix described a text-editing system that synthesizes *gap programs* based on examples [94]. FlashFill [49] is a programming by example system for automating syntactic string transformations in spreadsheets. It synthesizes programs with restricted form of regular expressions, conditionals, and loops for performing syntactic string transformations. Our work leverages FlashFill to perform semantic string transformations.

The Programming-by-demonstration (PBD) systems for text-editing like SMARTedit [76] or simultaneous editing [85] require the user to provide a complete demonstration or trace, where the demonstration consists of a sequence of the editor state after each primitive action, really spelling out how to do the transformation, but on a given example. This is considered to be one of the major obstacles in the adoption of programming by demonstration systems [75]. Our system is based on Programming by Example (as opposed to Programming by Demonstration) – it requires the user to provide only the final state (as opposed to also providing the intermediate states).

Topes [104, 103] provides end-users an abstraction of their data and helps them describe constraints on how to validate the data. From the given data, it infers some basic format such as numbers and words, and allows the users to modify the format by adding more constraints or specify additional formats. With these rules, Topes can validate user's data and provide error messages regarding why validation of certain strings failed. In addition, since it has knowledge of the formats, it also provides a finite set of formats as a recommendation in which a user might want to reformat the data. This can be viewed as an abstraction for providing Excel custom format strings, but is more powerful as additional constraints can be specified. Our tool is a com-

plete PBE system and allows users to format the data type strings in any arbitrary format without asking them to specify these format.

The work in [58] describes a programming by example technology for learning layout transformations on tables. In contrast, our work in FlashFill describes a learning algorithm for synthesizing string transformations based on table lookups.

Our work on learning semantic transformations in FlashFill is based on a novel learning algorithm. None of the examples we presented can be addressed by any of these systems because they do not implement any reasoning about semantic data types. The language of syntactic string transformations over which our system is built upon is also quite expressive, which enables our system to learn more complex and a much larger class of transformations than the ones that these other systems can learn.

Ad-hoc Data Manipulation for Programmers: The PADS project [40, 39] has enabled simplification of ad hoc data processing tasks for programmers by contributing along several dimensions: development of domain specific languages for describing text structure or data format, learning algorithms for automatically inferring such formats, and a markup language to allow users to add simple annotations to enable more effective learning of text structure. The learned format can then be used by programmers for documentation or implementation of custom data analysis tools. In contrast, we enable end-users (non-programmers) to perform small, often one-off, repetitive tasks on their spreadsheet data. Asking end-users to provide annotations for learning (relatively simple) text structure, and then asking them to develop custom tools to format/process the inferred structure is way above their expertise and usability bar. Hence, we automate end-to-end process, which includes not only learning the text structure from inputs, but also learning the desired transformation from outputs.

6.3 COMPUTER-AIDED EDUCATION AND GRADING

The technology developed in the programming languages and formal methods community can play a big role in enhancing Education. Recently, it has been applied to multiple aspects of Education including problem generation [114, 11, 9] and solution generation [55]. In our work on AutoProf, we push the frontier forward to cover another aspect namely automated grading. Recently [10] also applied automated grading to automata constructions and used syntactic edit distance like ours as one of the metrics. Our work differs from theirs in two regards: (a) our corrections for programs (which are much more sophisticated than automata) are teacher-defined, while [10] considers a small pre-defined set of corrections over graphs, and (b) we use the Sketch synthesizer to efficiently navigate the huge search space, while [10] uses brute-force search. A recent work by Gulwani

et al. [55] also uses program synthesis techniques for automatically synthesizing solutions to ruler/compass based geometry construction problems. Their focus is primarily on finding a solution to a given geometry problem whereas we aim to provide feedback on a given programming exercise solution.

AUTOMATED GRADING APPROACHES The survey by Douce et al. [34] presents a nice overview of the systems developed for automated grading of programming assignments over the last forty years. Based on the age of these systems, they classify them into three generations. The first generation systems [63] graded programs by comparing the stored data with the data obtained from program execution, and kept track of running times and grade books. The second generation systems [64] also checked for programming styles such as modularity, complexity, and efficiency in addition to checking for correctness. The third generation tools such as RoboProf [30] combine web technology with more sophisticated testing approaches. PEX4FUN [126] takes it a step further by adding social gaming component of *coding duels*. All of these approaches are a form of test-cases based grading approach and can produce feedback in terms of failing test inputs, whereas our technique generates feedback about the changes required in the student submission to make it correct.

AI BASED PROGRAMMING TUTORS There has been a lot of work done in the AI community for building automated tutors for helping novice programmers learn programming by providing feedback about semantic errors. These tutoring systems can be categorized into the following two major classes:

Code-based matching approaches: LAURA [8] converts teacher's and student's program into a graph based representation and compares them heuristically by applying program transformations while reporting mismatches as potential bugs. TALUS [88] matches a student's attempt with a collection of teacher's algorithms. It first tries to recognize the algorithm used and then tentatively replaces the top-level expressions in the student's attempt with the recognized algorithm for generating correction feedback. The problem with these approach is that the enumeration of all possible algorithms (with its variants) for covering all corrections is very large and tedious on part of the teacher.

Intention-based matching approaches: LISP tutor [38] creates a model of the student goals and updates it dynamically as the student makes edits. The drawback of this approach is that it forces students to write code in a certain pre-defined structure and limits their freedom. MENO-II [121] parses student programs into a deep syntax tree whose nodes are annotated with plan tags. This annotated tree is then matched with the plans obtained from teacher's solution.

PROUST [68], on the other hand, uses a knowledge base of goals and their corresponding plans for implementing them for each programming problem. It first tries to find correspondence of these plans in the student's code and then performs matching to find discrepancies. CHIRON [101] is its improved version in which the goals and plans in the knowledge base are organized in a hierarchical manner based on their generality and uses machine learning techniques for plan identification in the student code. These approaches require teacher to provide all possible plans a student can use to solve the goals of a given problem and do not perform well if the student's attempt uses a plan not present in the knowledge base.

Our approach performs semantic equivalence of student's attempt and teacher's solution based on exhaustive bounded symbolic verification techniques and makes no assumptions on the algorithms or plans that students can use for solving the problem. Moreover, our approach is modular with respect to error models; the local correction rules are provided in a declarative manner and their complex interactions are handled by the solver itself.

6.4 AUTOMATED PROGRAM REPAIR

The techniques developed for automated program repair is also related to our work in the AutoProf system. Könighofer et. al. [71] present an approach for automated error localization and correction of imperative programs. They use model-based diagnosis to localize components that need to be replaced and then use a template-based approach for providing corrections using SMT reasoning. Their fault model only considers the right hand side (RHS) of assignment statements as replaceable components. The approaches in [67, 124] frame the problem of program repair as a game between an environment that provides the inputs and a system that provides correct values for the buggy expressions such that the specification is satisfied. These approaches only support simple corrections (e.g. correcting RHS side of expressions) in the fault model as they aim to repair large programs with arbitrary errors. In AutoProf, we exploit the fact that we have access to the dataset of previous student mistakes that we can use to construct a *concise and precise* error model. This enables us to model more sophisticated transformations such as introducing new program statements, replacing LHS of assignments etc. in our error model. Our approach also supports minimal cost changes to student's programs where each error in the model is associated with a certain cost, unlike the earlier mentioned approaches.

Mutation-based program repair [32] performs mutations repeatedly to statements in a buggy program in order of their suspiciousness until the program becomes correct. The large state space of mutants (10^{12}) makes this approach infeasible. AutoProf uses a sym-

bolic search for exploring correct solutions over this large set. There are also genetic programming approaches that exploit redundancy present in other parts of the code for fixing faults [14, 41]. These techniques are not applicable in our case as such redundancy is not present in introductory programming problems.

Techniques like Delta Debugging [134] and QuickXplain [70] aim to simplify a failing test case to a minimal test case that still exhibits the same failure. AutoProf can be complemented with these techniques to restrict the application of rewrite rules to certain failing parts of the program only. There are many algorithms for fault localization [15, 45] that use the difference between faulty and successful executions of the system to identify potential faulty locations. Jose et. al. [69] recently suggested an approach that uses a MAX-SAT solver to satisfy maximum number of clauses in a formula obtained from a failing test case to compute potential error locations. These approaches, however, only localize faults for a single failing test case and the suggested error location might not be the desired error location, since we are looking for common error locations that cause failure of multiple test cases. Moreover, these techniques provide only a limited set of suggestions (if any) for repairing these faults.

6.5 QUERY SYNTHESIS IN DATABASES

Within the database literature, our work on semantic transformations is most closely related to the problems of *record matching*, *learning schema matches* and *query synthesis*. We have detailed some differences below, but the most significant difference is that we put these concepts together.

Record Matching: The task of syntactic manipulation performed before a lookup operation in our extended transformation language can be likened to the problem of *record matching*. Most of the prior work in this area [36, 72] has focused on designing appropriate similarity functions such as edit distance, jaccard similarity, cosine similarity, and HMM25. A basic limitation of most of them is that they have limited customizability. Arasu et.al. have proposed a customizable similarity measure that can either be user-programmed [12] or can be inferred from examples of matching textual records [13]. In both these cases, the underlying transformation rules only involve constant strings, e.g., $US \rightarrow \text{United States}$. Our record matching is also inferred from examples, but it involves generalized transformation rules consisting of syntactic operations such as regular expression matching, substring, and concatenate.

Learning Complex Schema Matches: The problem of synthesizing semantic string manipulations is also related to the problem of finding complex semantic matches between the data stored in disparate sources. The iMAP system [33] finds the schema matches that

involve concatenation of column strings across different tables using a domain-oriented approach. Another approach by Warren and Tompa [131] learns the relationships that involve concatenation of column substrings, but within a single table using a greedy approach. Our language-theoretic approach learns relationships that involve concatenation of column substrings across multiple database tables without using any domain knowledge about the column entries.

Query Synthesis by Example: The *view synthesis* [31, 127] problem aims to find the most succinct and accurate query for a given database view instance. The high-level goal of this work is similar to that of our inductive synthesis algorithm for the lookup transformation language L_t , but there are some key differences: (i) View synthesis techniques infer a relation from a large representative example view, while we infer a transformation from a set of few example rows (which is a critical usability aspect for end-users). (ii) View synthesis techniques infer the most likely relation, while our lookup synthesis algorithm infers a succinct representation of all possible hypotheses, which enables its extension to a synthesis algorithm for the language L_u . (iii) The technique in [31] does not consider *join* or *projection* operations.

6.6 RANKING IN PROGRAM SYNTHESIS

There have been several related work on using a manual ranking function for ranking of synthesized programs (or expressions). Gvero et. al. [57] use weights to rank the expressions for efficient synthesis of likely program expressions of a given type at a given program point. These weights depend on the lexical nesting structure of declarations and also on the statistical information about the usage of declarations in a code corpus. PROSPECTOR [81] synthesizes jungloid code fragments (chain of objects and method calls from type τ_{in} to type τ_{out}) by ranking jungloids using the primary criterion of length, and secondary criteria of number of crossed package boundaries and generality of output type. Perelman et. al. [95] synthesize hole values in partial expressions for code completion by ranking potential completed expressions based on features such as class hierarchy of method parameters, depth of sub-expressions, in-scope static methods, and similar names. PRIME [87] uses relaxed inclusion matching to search for API-usage from a large collection of code corpuses, and ranks the results using the frequency of similar snippets. The SEMFIX tool [92] uses a manual characterization of components in different complexity levels for synthesizing simpler expression repairs. Our ranking scheme also uses some of these features, but we learn the ranking function automatically using machine learning unlike these techniques which need manual definition and parameter tuning for the ranking function.

SLANG [98] uses the regularities found in sequences of method invocations from large code repositories to synthesize likely method invocation sequences for code completion. It uses alias and history analysis to extract precise sequences of method invocations during the training phase, and then trains a statistical language model on the extracted data. CodeHint [44] is an interactive and dynamic code synthesis system that also employs a probabilistic model learnt over ten million lines of code to guide and prune the search space. The motivation of our ranking technique is also similar as we want to learn models of programs from a database of training benchmarks. The main difference in our technique is that it is based on version-space algebra based representation, where we compute all possible conforming programs and then efficiently rank them in a hierarchical fashion based on different expression sharing using the corresponding efficient features and algorithms.

LEARNING TO RANK Learning to rank has been an active area of research in the machine learning community over the last decade. Most learning to rank approaches can be categorized into three broad categories: pointwise, pairwise, and listwise. The pointwise approaches transform the ranking problem to a classification [91] or ordinal regression problem [26]. This approach requires numerical rank values for each training instance. The pairwise approaches transform the ranking problem to a classification problem of preferring one object over another such that the number of misclassified pairs are minimized [62, 43, 22]. The downside of this approach is that it needs to construct a pair of positive and negative expression for each input-output example that leads to a quadratic increase in the number of training instances. The listwise approach uses list of objects as a training instance and directly optimizes the evaluation measures. ListNet [24] proposes probabilistic models of permutation and top-k probability for defining listwise loss functions for a neural network model and uses gradient descent algorithm to compute the ranking function. ListMLE [133] models the listwise ranking problem as that of minimizing the likelihood loss function. Our approach is also a listwise approach as it learns a ranking function over a list of program expressions. A major difference in our technique, however, is that it aims to rank *any* positive expression higher than *all* negative expressions unlike all previous techniques.

MACHINE LEARNING FOR PROGRAMMING BY EXAMPLE A recent work by Menon et al. [84] uses machine learning to bias the search for finding a composition of a given set of typed operators based on clues obtained from the examples. Raychev et. al. [97] use A* search based on a heuristic function of length of current refactoring sequence and estimated distance from target tree for efficient

learning of software refactorings from few user edits. On the other hand, we use machine learning to identify an intended program from a given set of programs that are consistent with a given set of examples. Our technique is applicable to domains where it is possible to compute the set of all programs that are consistent with a given set of examples [56, 50]. SMARTedit [76] is a PBD (Programming By Demonstration) text-editing system where a user presents demonstration(s) of the text-editing task and the system tries to generalize the demonstration(s) to a macro by extending the notion of version-spaces to model plausible macro hypotheses. The macro language of SMARTedit is not as expressive as FlashFill's, and furthermore the task demonstrations in SMARTedit reduce a lot of ambiguity in the hypothesis space. Liang et al. [78] introduce hierarchical Bayesian prior in a multi-task setting that allows sharing of statistical strength across tasks. This allows to provide an inductive bias to common sub-tasks across multiple tasks and helps in learning the desired user intention from few demonstrations. Our underlying language and representation of string manipulation programs is very different from the combinatory logic based program representation used by Liang et al., which requires us to use a different approach for learning the ranking function.

FUTURE WORK

In this chapter, I describe some of the work I am planning to work on in the short term to extend the capabilities of the three synthesis systems. I also present some ideas and directions for building upon these synthesis techniques for creating such systems for making programming accessible to a much larger class of end-users, students, and programmers.

Probabilistic Synthesis in FlashFill

While going through many FlashFill blogs and tutorial videos, we observed that often times users were trying to use FlashFill to transform strings that represented *unstructured* and *non-uniform* data types. The unstructured data types refers to a collection of data type strings whose constituent fields do not follow a logical pattern recognizable by syntactic regular expressions. The non-uniform data types refers to a collection of data type strings that are in multiple different formats. The example shown in Figure 84 shows a collection of ambiguous and non-uniform date strings. An Excel user wanted to format dates present in different formats into a uniform format as shown in the Figure. Note that the semantic transformation language can not express such transformations because: (i) there exists input strings such as 4/18/2010 for which FlashFill has not seen an input-output example yet, and (ii) the fields are not recognizable by syntactic regular expressions. Sometimes, the data is also ambiguous, e.g. the string 8-7-2010, which can be interpreted as both 7 August and 8 July.

	Input	Output
1	24.9	24 Sep 2010
2	6-21-2010	21 Jun 2010
3	29.1	29 Jan 2010
4	8-7-2010	7 Aug 2010
5	4/18/2010	18 Apr 2010
6	16.8	16 Aug 2010

Figure 84: Learning transformations on unstructured, non-uniform, and ambiguous data type strings.

We are developing a probabilistic domain-specific language for handling transformations on noisy, non-uniform, and ambiguous data.

This language is parameterized by semantic entity descriptions that enables it to efficiently support more sophisticated transformations (requiring semantic knowledge) over such data. The expressions learnt in this language are assigned likelihood probabilities based on the provided input-output examples as well as other input strings present in the spreadsheet. Because of the probabilistic semantics, it can also handle noisy input-output examples (inconsistent examples) such as spelling mistakes or small typos in the user-provided output strings.

Learning Probabilistic Error Models from edX data

The AutoProf system currently requires instructors to provide error models consisting of common mistakes that students are making on a given problem. This manual effort of creating error models is time-consuming and tedious for instructors, and sometimes even prohibitively expensive to find correction rules for mistakes that occur infrequently in practice. The syntax of error model language is also something teachers need to learn to be able to write error models for AutoProf.

We propose a technique to automatically learn error models from the logs of thousands of students attempts. The edX data logs consists of sequence of student attempts for a given problem, and many students correct their mistakes themselves after performing a sequence of corrections. For such student logs, we can perform a `diff` on their consecutive attempts to find what changed between the two attempts, which would in turn lead to a potential correction rule. We can then compute the frequency distribution of occurrence of each correction rule to assign a probability (cost) for the rule, and can then use probabilistic synthesis techniques (similar to FlashFill above) for finding most likely changes to a student program.

AutoProf as an Intelligent Tutoring System for Programming

The AutoProf feedback describing exactly what is wrong with a program and how to fix it is great for instructors and teaching assistants, who are trying to grade student programs and provide detailed feedback. But such detailed feedback is not great for students who are trying to learn programming as it might prevent them from learning the conceptual mistake and essential debugging skills.

We are currently working on using the information obtained from the AutoProf analysis to add another feedback level for tutoring. The main idea is to provide a more general feedback that helps students learn how to debug their programs. Instead of telling the location and fix of the problem, the feedback asks them to print certain variables and expressions in their program, which in turn allows students to understand their mistake and find fixes for them. We are working

with the MITx and edX team to deploy AutoProf with this feedback level in the 6.0001 class in the 2014 Fall semester, and are planning to perform some studies on the effectiveness of such feedback for helping students learn programming.

Table Interface for SPT

We are currently collaborating with researchers from the Multimodal Understanding Group at MIT to create a tablet interface for SPT, where a user can draw the examples directly on the tablet screen, and can interact with the system using even more modalities such as speech to provide additional specifications [100]. The specifications obtained from such an interface are often not as detailed as one expects in the textual specification of SPT, so the system now needs to perform a few inferences about the missing information. We are also planning to perform studies with students using the tablet interface to see how natural such an interface is for students to learn data structure manipulations.

Other Program Synthesis Applications for Accessible Programming

We now describe few other exciting applications of program synthesis technology for making programming accessible to an even larger community.

For End-users: Nowadays only a handful of people can create smartphone applications but the ability to easily program them can empower end-users to build custom apps for their routine tasks. There are already systems like AppInventor¹ and TouchDevelop² that make it easier to build such apps, but they still involve some form of programming. We would like to build a system where end-users can provide a few demonstrations of expected app behaviors using a vocabulary of high-level components, which can then be generalized by a synthesizer to create the custom app. Another domain that end-users struggle with is the domain of web programming, e.g. for writing scripts to extract data from a website or enter form values from a database. We want to combine ideas from FlashFill to create a system where users can provide demonstrations of the desired task, and the system then learns the intended program.

For Programmers: We would like to integrate example-based specification mechanisms into mainstream programming languages as even programmers can benefit greatly from them. For data processing and text manipulation tasks, programmers typically use languages like Awk and Perl, but every language supports a different regular expression syntax and even figuring out the desired regular expression

¹ <http://appinventor.mit.edu>

² <http://www.touchdevelop.com>

is quite tricky. We want to develop a language with a tighter integration of example-based specification mechanisms such that the compiler can synthesize the intended regular expressions. Another domain where such an integration would be useful is auto-completion and intellisense in IDEs. We would like to provide language constructs to programmers to express partial expressions, which the synthesizer can use to provide hints for possible well-typed completions.

For Students (Education Technology): We would like to collaborate with education researchers to measure the learning outcomes of the AutoProf tutoring system for programming. Interestingly, similar program synthesis techniques can also be used for building tutoring and feedback systems for other K-12 STEM subjects such as mathematics and physics [51]. For proof problems that are common in mathematics and logic courses, where students need to write step-by-step solutions and where there are multiple correct ways of solving, a technique based on AutoProf can be used to provide feedback and to teach different concepts. We would like to build such a system for them and this would also give us an opportunity to impart some computational concepts to K-12 students.

Program synthesis techniques also have a big role to play for the problem of content creation. For example, enabling students and teachers to easily enter their structured content (problems or solutions) naturally and easily as opposed to using existing techniques like Microsoft Word or Latex. The content creation problem can be formalized as a synthesis problem as follows: given a term t , a rule/axiom r , and a partial expression t'_p , complete the partial expression t'_p to t' such that $t \rightarrow_r t'$. In other words, our goal is to build a system that can auto-populate the term t' by learning the desired transformation on a term t . Our initial results for building systems for content creation in the domains of trigonometry and probability have been quite encouraging. We are also exploring other natural specification mechanisms for content creation such as voice and ink.

Another interesting side-effect of the MOOCs movement has been the capability to capture large amounts of data about student interactions. We would like to leverage this vast amount of data to learn common sources of student misunderstandings as well as developing customized learning experiences based on a student's performance.

CONCLUSIONS

In this thesis, I presented three systems `SPT`, `FlashFill` extensions, and `AutoProf` that work towards achieving the goal of making programming accessible to a large class of users namely end-users and students. These systems are based on new program synthesis approaches that allow for more natural and intuitive specification mechanisms. The hypothesis spaces in these approaches can be fixed or parametric (user-defined). The key idea for designing fixed hypothesis spaces is to define them using domain-specific languages that are expressive enough to express most tasks in the domain but at the same time concise enough for efficiently learning expressions in them. The user-defined hypothesis spaces are parameterized with intuitive user inputs to allow users to easily define and control the search space. We use constraint-based and version-space algebra based synthesis algorithms to efficiently learn programs in this large hypothesis space that conform to the provided specification. The user interaction model is also important for such systems for enabling users to refine their intent and to make them usable in practice. We believe this thesis present a first step towards using the power of automated program synthesis for democratizing programming, and there are many new exciting systems that can be built upon these foundations to make programming accessible to an even larger class of people.

BIBLIOGRAPHY

- [1] Excel 2013's coolest new feature that should have been available years ago, CNNMoney. <http://cnnmoneytech.tumblr.com/post/27346588168/excel-2013s-coolest-new-feature-that-should-have-been?iid=EL>.
- [2] FlashFill in Microsoft Excel 2013. <http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html>.
- [3] Excel Flash Fill is a Brilliant Time Saver. <http://www.lifehacker.com.au/2012/07/excel-flash-fill-is-a-brilliant-time-saver/>.
- [4] Insertion in Red-black tree. <http://cs.wellesley.edu/~cs231/fall01/red-black.pdf>.
- [5] The Storyboard Programming Tool (SPT). <http://people.csail.mit.edu/rishabh/storyboard/>.
- [6] 5 hot features of Microsoft Office 2013. <http://slideshow.techworld.com/3372966/5-hot-features-of-microsoft-office-2013/5/>.
- [7] The most notable feature in Excel is Flash Fill, Times of India. <http://timesofindia.indiatimes.com/tech/itslideshow/15015129.cms>.
- [8] Anne Adam and Jean-Pierre H. Laurent. LAURA, A System to Debug Student Programs. *Artif. Intell.*, 15(1-2):75–122, 1980.
- [9] Umair Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. In *IJCAI*, 2013.
- [10] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *IJCAI*, 2013.
- [11] Erik Andersen, Sumit Gulwani, and Zoran Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI*, 2013.
- [12] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

- [13] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.
- [14] Andrea Arcuri. On the automation of fixing software bugs. In *ICSE Companion*, 2008.
- [15] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.
- [16] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: a word processor with a crowd inside. In *UIST*, 2010.
- [17] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.
- [18] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.*, 3(4):353–387, 1981. ISSN 0164-0925.
- [19] Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, pages 24–40, 2010.
- [20] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, pages 101–112, 2008.
- [21] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.
- [22] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *ICML*, 2005.
- [23] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
- [24] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, 2007.
- [25] Carlos Christensen. On the implementation of ambit, a language for symbol manipulation. *Commun. ACM*, 9(8):570–573, 1966.

- [26] David Cossock and Tong Zhang. Subset ranking using regression. *Learning Theory*, 4005:605–619, 2006.
- [27] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [28] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
- [29] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-03213-9.
- [30] Charlie Daly. Roboprof and an introductory computer programming course. ITiCSE, 1999.
- [31] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.
- [32] V. Debroy and W.E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, 2010.
- [33] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Y. Halevy, and Pedro Domingos. iMAP: Discovering complex mappings between database schemas. In *SIGMOD*, pages 383–394, 2004.
- [34] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.
- [35] T. O. Ellis, J. F. Heafner, and W. F. Sibley. The grail project: An experiment in man-machine communications. Technical Report RM-5999-ARPA, RAND, 1969.
- [36] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
- [37] P.A. Ertmer, J.C. Richardson, B. Belland, D. Camin, P. Connolly, G. Coulthard, K. Lei, and C. Mong. Using peer feedback to enhance the quality of student online postings: An exploratory study. *Journal of Computer-Mediated Communication*, 12(2):412–433, 2007.
- [38] Robert G. Farrell, John R. Anderson, and Brian J. Reiser. An interactive computer-based tutor for lisp. In *AAAI*, 1984.

- [39] Kathleen Fisher and David Walker. The PADS project: an overview. In *ICDT*, pages 11–17, 2011.
- [40] Kathleen Fisher, David Walker, and Kenny Qili Zhu. Learn-PADS: automatic tool generation from ad hoc data. In *SIGMOD*, pages 1299–1302, 2008.
- [41] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *GECCO*, 2009.
- [42] Pascal Fradet and Daniel Le Metayer. Shape types. In *POPL*, pages 27–39. ACM Press, 1997.
- [43] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *The Journal of machine learning research*, 4:933–969, 2003.
- [44] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663, 2014.
- [45] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
- [46] Mike Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, April 2009.
- [47] Mike Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, April 2009.
- [48] Sumit Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
- [49] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.
- [50] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012.
- [51] Sumit Gulwani. Example-based learning in computer-aided stem education. *Commun. ACM*, 57(8):70–80, 2014.
- [52] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

- [53] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09: Proceedings of the 2009 Conference on Verification Model Checking and Abstract Interpretation*, pages 120–135, 2009.
- [54] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
- [55] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.
- [56] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet Data Manipulation Using Examples. *Communications of the ACM Research Highlight*, 55(8):97–105, August 2012.
- [57] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.
- [58] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.
- [59] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *PLDI*, pages 38–49. ACM, 2011.
- [60] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In *PLDI*, 2011.
- [61] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.
- [62] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. Large margin rank boundaries for ordinal regression. *Advances in Neural Information Processing Systems*, pages 115–132, 1999.
- [63] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5), May 1969.
- [64] David Jackson and Michelle Usher. Grading student programs using assyst. *SIGCSE*, 1997.
- [65] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

- [66] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.
- [67] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.
- [68] W. Lewis Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. *IEEE Trans. Software Eng.*, 11(3):267–275, 1985.
- [69] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.
- [70] Ulrich Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, 2004.
- [71] Robert Könighofer and Roderick Paul Bloem. Automated error localization and correction for imperative programs. In *FM-CAD*, 2011.
- [72] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803, 2006.
- [73] Chinmay Kulkarni and Scott R. Klemmer. Learning design wisdom by augmenting physical studio critique with online self-assessment. Technical report, Stanford University, 2012.
- [74] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. *PLDI*, 2010.
- [75] Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–67, 2009.
- [76] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.
- [77] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
- [78] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.
- [79] Henry Lieberman. Tinker: Example-based programming for artificial intelligence. In *IJCAI*, 1981.
- [80] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. TurkIt: human computation algorithms on mechanical turk. In *UIST*, 2010.

- [81] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.
- [82] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979. ISSN 0098-5589. doi: <http://doi.ieeecomputersociety.org/10.1109/TSE.1979.234198>.
- [83] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/357084.357090>.
- [84] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, 2013.
- [85] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.
- [86] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS, UC Berkeley, 2005.
- [87] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, pages 997–1016, 2012.
- [88] William R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3:1–16, 1987.
- [89] B. A. Myers. Displaying data structures for interactive debugging. Technical Report CSL-80-7, Xerox PARC, 1980.
- [90] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *SIGCHI Bull.*, 17(4):59–66, 1986. ISSN 0736-6906.
- [91] Ramesh Nallapati. Discriminative models for information retrieval. In *Proceedings of SIGIR*, pages 64–71, 2004.
- [92] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *ICSE*, 2013.
- [93] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Weingan Chin. Automated verification of shape and size properties via separation logic. In *In VMCAI*. Springer, 2007.
- [94] Robert P. Nix. Editing by example. *TOPLAS*, 7(4):600–621, 1985.

- [95] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286, 2012.
- [96] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 652–671, London, UK, 1989. Springer-Verlag. ISBN 3-540-51371-X.
- [97] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin T. Vechev. Refactoring with synthesis. In *OOPSLA*, pages 339–354, 2013.
- [98] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, 2014.
- [99] R.V. Rubin, E.J. Colin, and S.P. Reiss. Think pad: A graphical system for program-ming by demonstration. *IEEE Software*, 2: 73–79, 1985. ISSN 0740-7459.
- [100] Andrew Sabisch. *CoMo: A Whiteboard that converses about Code*. PhD thesis, MIT CSAIL, 2014.
- [101] W. Sack, E. Soloway, and P. Weingrad. From PROUST to CHIRON: Its design as iterative engineering: Intermediate results are important! In *In J.H. Larkin and R.W. Chabay (Eds.), Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches.*, pages 239–274, 1992.
- [102] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: <http://doi.acm.org/10.1145/292540.292552>.
- [103] Chris Scaffidi. Topes: Enabling end-user programmers to validate and reformat data, 2009.
- [104] Christopher Scaffidi, Brad A. Myers, and Mary Shaw. Intelligently creating and recommending reusable reformatting rules. In *IUI*, 2009.
- [105] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *IJCAI'75: Proceedings of the 4th international joint conference on Artificial intelligence*, pages 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

- [106] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. (In submission).
- [107] Rishabh Singh and Sumit Gulwani. Synthesizing Number Transformations from Input-Output Examples. In *CAV*, pages 634–651, 2012.
- [108] Rishabh Singh and Sumit Gulwani. Learning Semantic String Transformations from Examples. *PVLDB*, 5(8):740–751, 2012.
- [109] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011.
- [110] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011.
- [111] Rishabh Singh and Armando Solar-Lezama. SPT: Storyboard Programming Tool. In *CAV*, pages 738–743, 2012.
- [112] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.
- [113] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.
- [114] Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.
- [115] David Canfield Smith. *Pygmalion: a creative programming environment*. PhD thesis, Stanford, CA, USA, 1975.
- [116] Douglas R. Smith. Kids: A semi-automatic program development system. In *Client Resources on the Internet, IEEE Multimedia Systems*, pages 302–307, 1990.
- [117] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.
- [118] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
- [119] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, 2007.
- [120] Armando Solar-Lezama, Chris Jones, Gilad Arnold, and Rastislav Bodík. Sketching concurrent datastructures. In *PLDI 08*, 2008.

- [121] Elliot Soloway, Beverly Park Woolf, Eric Rubin, and Paul Barth. Meno-II: An Intelligent Tutoring System for Novice Programmers. In *IJCAI*, 1981.
- [122] Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. From program verification to program synthesis. *POPL*, 2010.
- [123] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, pages 492–503. ACM, 2011.
- [124] Stefan Simon Staber, Barbara Jobstmann, and Roderick Paul Bloem. Finding and fixing faults. In *Correct Hardware Design and Verification Methods*, Lecture notes in computer science, pages 35 – 49, 2005.
- [125] Ivan E. Sutherland. *Sketchpad, A Man-Machine Graphical Communication System*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1963. ISBN 0-8240-4411-8.
- [126] N. Tillmann, J. De Halleux, Tao Xie, and J. Bishop. Pex4Fun: Teaching and learning computer science via social gaming. In *CSE&T*, 2012.
- [127] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.
- [128] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.
- [129] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, 2008. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/1379022.1375598>.
- [130] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.
- [131] Robert H. Warren and Frank Wm. Tompa. Multi-column substring matching for database schema translation. In *VLDB*, pages 331–342, 2006.
- [132] Daniel S Weld, Eytan Adar, Lydia Chilton, Raphael Hoffmann, and Eric Horvitz. Personalized online education - a crowdsourcing challenge. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

- [133] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. Listwise approach to learning to rank: theory and algorithm. In *ICML*, 2008.
- [134] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28: 183–200, 2002.