Synthesis from Examples

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

ABSTRACT

Examples are often a natural way to specify computational structures such as programs, queries, and sequences. Synthesizing such structures from example based specification has applications in automating end-user programming and in building intelligent tutoring systems. Synthesis from examples involves addressing two key technical challenges: (i) design of an efficient search algorithm – these algorithms have been based on various paradigms including version-space algebras, SAT/SMT solvers, numerical methods, and even exhaustive search, (ii) design of a user interaction model to deal with the inherent ambiguity in the example based specification. This paper illustrates various algorithmic techniques and user interaction models by describing inductive synthesizers for varied applications including synthesis of tricky bitvector algorithms, spreadsheet macros for automating repetitive data manipulation tasks, ruler/compass based geometry constructions, new algebra problems, sequences for mathematical intellisense, and grading of programming problems.

1. INTRODUCTION

The IT revolution over the past few decades has resulted in widespread access to computational devices. Unfortunately, it has not been easy to interact with these devices because of the need to under the syntax and semantics of computational structures (such as programs or queries) required to interact with these devices. Program synthesis or automatic programming has the revolutionary potential to change this landscape.

Program synthesis is the task of automatically synthesizing a program in some underlying language from a given specification using some search technique [7]. The synthesis technology has the potential to impact various classes of users in the technology pyramid ranging from algorithm/system designers [35, 10, 14, 13, 36, 15] and software developers [34, 25] at the top of the pyramid to end-users [9, 8, 12, 28, 29] and students/teachers [11, 30, 26, 31] at the bottom of the pyramid. In this paper, we will present applications for various such classes of users.

A traditional view of program synthesis is that of synthesis from complete specifications. One approach is to give a specification as a formula in a suitable logic [24, 35, 10, 36, 15]. Another is to write the specification as a simpler, but possibly far less efficient program [33, 16, 34]. While these approaches have the advantage of completeness of specification, such specifications are often unavailable, difficult to write, or expensive to check against using automated verification techniques. In this paper, we focus on another style of specifications, namely specification by examples [22, 6].

It is natural to ask what prevents the synthesizer from synthesizing a trivial program that simply performs a table lookup as follows, when provided with the set $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of input-output pairs.

switch x
  case x_1: return y_1;
  case x_2: return y_2;
  ⋮
  case x_n: return y_n;

The restriction on the underlying search space is what often prevents such trivial solutions. In particular, the search space might permit only a bounded number of statements or conditionals. This paper describes various examples of interesting and useful search spaces that are often described as domain-specific languages.

There are two key challenges in designing inductive synthesizers that take examples as specifications. The first challenge is that of designing a good user interaction model that can deal with the inherent ambiguities in examples, which are often an under-specification of the user's intent. The inductive synthesizers presented in this paper use a variety of effective user interaction models.

The second challenge is that of designing an efficient algorithm that can search for structures (in the underlying search space) that are consistent with the user provided examples. The inductive synthesizers presented in this paper use techniques that have been developed in various communities including use of SAT/SMT solvers (formal methods community), version space algebras [21] (machine learning community), and A*-style goal-directed heuristics (AI community).

We next describe a variety of synthesis by example technologies for various classes of users. We start out by describing a technique for synthesizing bitvector algorithms, which targets algorithm designers (§2). Then, we describe inductive synthesizers for spreadsheet data manipulation, which targets end-users (§3). Finally, we present some surprising applications of this technology in the area of intelligent tutoring systems, which targets students and teachers (§4).

This paper is based on the keynote given at the 3rd Workshop on Advances in Model-Based Software Engineering (WAMBSE) held at IIT Kanpur (India) in February 2012.

2. BITVECTOR ALGORITHMS

Finding a new algorithmic solution for a given problem requires human ingenuity. Use of computational techniques to discover new algorithmic insights can be the ultimate application of program synthesis. One domain of algorithms that has been shown amenable to automated synthesis is the class of bitvector algorithms [18, 38], which are typically straight-line sequence of instructions that use both arithmetic and logical bitwise operators.[1] Such programs can be quite unintuitive and extremely difficult for average, or sometimes even expert, programmers to discover methodically.

Consider the task of designing a bitvector algorithm that masks off the right-most significant 1-bit in an input bitvector. More formally, the bitvector algorithm takes as input one bitvector x and outputs a bitvector s such that s is obtained from x by setting the right-most significant 1-bit in x to 0. For example, the bitvector algorithm should transform the bitvector 01100 into 01000. A simple method to accomplish this would be to iterate over the input bitvector starting from the rightmost end until a 1 bit is found and then set it to 0. However, this algorithm is worst-case linear in the number of bits in the input bitvector. Furthermore, it uses undesirable branching code inside a loop. There is a non-intuitive, but quite elegant, way to achieving the desired functionality in constant time by using a tricky composition of the standard subtraction operator and the bitwise logical & operator, which are supported by almost every architecture. In particular, the desired functionality can be achieved using the following composition: $x \,\&\, (x-1)$. The reason why we can do this seemingly worst-case linear task in unit time using the subtraction operator and the logical bitwise-and operator is because the hardware implementations of these operators manipulate the constituent bits of the bitvectors in parallel in constant time.

As another example, consider the task of computing (the floor of) the average of two 32-bit integers x and y. Note that computing average using the expression $(x+y)/2$ is inherently flawed and vulnerable since it can overflow. However, using some bitwise tricks, the average can be computed without overflowing; one such way to compute it is: $(x \,\&\, y) + ((x \oplus y) >> 1))$.

Such tasks can be described by writing a logical specification that relates the input and output bitvectors. We have presented a technique for synthesizing bitvector algorithms from logical specifications [10]. However, such logical specifications may often be tricky to write themselves. A simple alternative to writing such logical specifications is to provide input-output examples. We detail below an interesting interaction model that can be used to guide the user towards providing more descriptive examples [14].

User Interaction Model.

Given a set of input-output examples, the synthesizer searches for programs that map each input in the given set to the corresponding output. The number of such programs may usually be unbounded, if the search space consists of all possible programs. However, since the search space is usually restricted, the number of such programs may either be 0, 1, or more than 1. If the synthesizer is unable to find any such program over the search space, the synthesizer declares failure. If the synthesizer finds exactly 1 program, the synthesizer declares success and presents the program to the user. If the synthesizer finds at least two programs P_1 and P_2, both of which map each input in the given set to the corresponding output, the synthesizer declares the user specification to be partial. It then generates a *distinguishing input*, an input on which the two programs P_1 and P_2 yield different results, and asks the user to provide the output corresponding to the distinguishing input. The synthesis process is then repeated after adding this new input-output example to the previous set of input-output examples.

Search Algorithm.

Our synthesis algorithm [14] is based on a novel *constraint-based approach* that reduces the synthesis problem to that of solving two kinds of constraints: the *I/O-behavioral constraint* whose solution yields a candidate program consistent with the given set of input-output examples, and the *distinguishing constraint* whose solution provides the input that distinguishes between non-equivalent candidate programs. These constraints can be solved using off-the-shelf SMT (Satisfiability Modulo Theory) solvers. Instead of performing an expensive combinatorial search over the space of all possible programs, our technique leaves the inherent exponential nature of the problem to the underlying SMT solver—whose engineering advances over the years allow them to effectively deal with problem instances that arise in practice (which are usually not hard, and hence end up not requiring exponential reasoning).

EXAMPLE 1. *Consider the task of synthesizing the bitvector algorithm that masks off the rightmost contiguous sequence of 1s in the input bitvector. The synthesizer driven input-output interaction process is illustrated in Figure 1. The user may start out by providing one input-output example (*01011, 01000*) for the desired program. The synthesizer generates a candidate program* $(x+1) \,\&\, (x-1)$ *that is consistent with the input-output pair* $(01011, 01000)$*. Then, it checks whether a semantically different program also exists and comes up with an alternative program* $(x+1) \,\&\, x$ *and a distinguishing input* 00000 *that distinguishes the two programs, and asks the user for the output for the distinguishing input. The newly obtained input-output pair* $(00000, 00000)$ *rules out one of the candidate programs, namely,* $(x+1) \,\&\, (x-1)$*. In the next iteration, with the updated set of input-output pairs, the synthesizer finds two different programs* $-(\neg x) \,\&\, x$ *and* $(((x \& -x) \mid -(x-1)) \& x) \oplus x$ *and a distinguishing input* 00101*. It then asks the user for the output for* 00101*. The newly added pair* $(00101, 00100)$ *rules out* $(((x \& -x) \mid -(x-1)) \& x) \oplus x$*. Note that at this stage, the program* $(x+1) \& x$ *remains a candidate, since it was not ruled out in the earlier iterations. In next four iterations, the synthesizer driven interaction leads to four more input-output pairs:* $(01111, 00000)$*,* $(00110, 00000)$*,* $(01100, 00000)$ *and* $(01010, 01000)$*. The semantically unique program generated from the resulting set of input-output pairs is the desired program:* $(((x-1) \mid x) + 1) \,\&\, x$*.*

[1] These algorithms "typically describe some plausible yet unusual operation on integers or bit strings that could easily be programmed using either a longish fixed sequence of machine instructions or a loop, but the same thing can be done much more cleverly using just four or three or two carefully chosen instructions whose interactions are not at all obvious until explained or fathomed" [38].

User	Oracle		
Input → Output	**Program 1**	**Program 2**	**Distinguishing Input ?**
01011 → 01000	$(x+1)\,\&\,(x-1)$	$(x+1)\,\&\,x$	00000 ?
00000 → 00000	$-(\neg x)\,\&\,x$	$(((x\&-x)\mid-(x-1))\&x)\oplus x$	00101 ?
00101 → 00100	$(x+1)\&x$	$\cdots$	01111 ?
01111 → 00000	$\cdots$	$\cdots$	00110 ?
00110 → 00000	$\cdots$	$\cdots$	01100 ?
01100 → 00000	$\cdots$	$\cdots$	01010 ?
01010 → 01000	$(((x-1)\mid x)+1)\&x$	None	Program is $\boldsymbol{(((x-1)\mid x)+1)\&x}$

Figure 1: An illustration of the synthesizer driven interaction model for synthesis from input-output examples (for the task of turning off the rightmost contiguous sequence of 1 bits). Program 1 and Program 2 are two semantically different programs generated by the synthesizer that are consistent with the past set of input-output pairs provided by the user. The synthesizer also produces a distinguishing input on which the two programs yield different results, and asks the user for the output corresponding to the distinguishing input. The process is repeated until the synthesizer can find at most one program.

3. SPREADSHEET DATA MANIPULATION

More than 500 million people worldwide use spreadsheets for storing and manipulating data. Unfortunately, the state of the art of interfacing with spreadsheets is far from satisfactory. Spreadsheet systems, like Microsoft Excel, come with a maze of features, but end-users struggle to find the correct features to accomplish their tasks [20]. More significantly, programming is still required to perform tedious and repetitive tasks such as transforming names or phone-numbers or dates from one format to another, cleaning data, or extracting data from several text files or web pages into a single document. Excel allows users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts in Visual Basic or .NET programming languages. However, since end-users are not proficient in programming, they find it too difficult to write desired macros or scripts. Moreover, even skilled programmers might hesitate to write a script for a *one-off* repetitive task.

We performed an extensive case study of spreadsheet help forums and observed that string, number, and table processing constitute a very common class of programming problems that end-users struggle with. We have defined various domain specific languages L for manipulating strings (§3.1, §3.2), numbers (§3.3), and tables (§3.4). Each of these languages is expressive enough to capture several real-world tasks in the underlying domain, but also restricted enough to enable efficient learning from examples. For each of these languages, we have developed an inductive synthesizer that can generate a script from input-output examples. We describe below the high-level structure of the search algorithm and the user interaction model that is common to all these synthesizers [9]. More specific details of the synthesizers can be found in respective papers [8, 28, 29, 12].

Search Algorithm.

The number of programs in the underlying domain specific language L that are consistent with a given set of input-output examples can be huge. We first define a data structure D (based on a version-space algebra [21]) to succinctly represent a large set of such programs. Our synthesis algorithm for language L applies two key procedures: (i) `Generate` learns the set of all programs, represented using data structure D, that are consistent with a given single example. (ii) `Intersect` intersects these sets (each corresponding to a different example). We also develop a scheme that ranks programs, preferring programs that are more general. Each ranking scheme is inspired by Occam's razor, which states that a smaller and simpler explanation is usually the correct one. The ranking scheme is used to select the top-ranked programs from among the set of all programs that are consistent with the user provided examples. Note that the choice of data-structure D, the procedures `Generate` and `Intersect`, and the ranking scheme are all specific to the language L.

User Interaction Model.

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

We next describe some task domains to which we have applied this general methodology.

3.1 Syntactic String Transformations

Spreadsheet users often struggle with reformatting or cleaning data in spreadsheet columns [8]. For example, consider the following tasks.

EXAMPLE 2 (PHONE NUMBERS). *An Excel user wants to uniformly format the phone numbers in the input column, adding a default area code of "425" if the area code is missing.*

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

Our tool [8] synthesizes a script for performing the desired transformation from the first two example rows and uses it to produce the entries in the next three rows (shown here in boldface for emphasis) of the output column.

EXAMPLE 3 (GENERATE ABBREVIATION). *The following task was presented originally as an* Advanced Text Formula [37].

Input v_1	Output
Association of Computing Machinery	ACM
Principles Of Programming Languages	**POPL**
International Conference on Software Engineering	**ICSE**

The above-mentioned tasks require syntactic string transformations. We have developed an expressive domain-specific language of string-processing programs, that supports limited conditionals and loops, syntactic string operations such as substring and concatenate, and matching based on regular expressions [8].

3.2 Semantic String Transformations

Some string transformation tasks also involve 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 date, time, currency, or phone number [28]. For example, consider the following task from an Excel help forum.

EXAMPLE 4. *A shopkeeper wants to compute the* `selling price` *of an item (Output) from its* `name` *(Input v_1) and* `selling date` *(Input v_2). The inventory database of the shop consists of two tables: (i)* `MarkupRec` *table that stores* `id`, `name` *and* `markup percentage` *of items, and (ii)* `CostRec` *table that stores* `id`, `purchase date` *(in month/year format), and* `purchase price` *of items. The selling price of an item is computed by adding its purchase price (for the corresponding month) to its markup charges, which in turn is calculated by multiplying the markup percentage by the purchase price.*

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

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

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

To perform the above task, the user must perform a join of the two tables on the common item `Id` *column to lookup the item* `Price` *from its* `Name` *(v_1) and selling* `Date` *(substring of v_2), besides performing syntactic string manipulations. Our tool [28] synthesizes a script for performing the desired transformation from the first two example rows and uses it to produce the entries in the next three rows (shown here in boldface for emphasis).*

For automation of such tasks, we have extended the language mentioned in §3.1 with a relational algebra for selecting strings from relational tables [28]. This extended language also enables manipulation of strings that represent standard data types whose semantic meaning can be encoded as a database of relational tables. For example, consider the following date manipulation task.

EXAMPLE 5 (DATE MANIPULATION). *An Excel user wanted to convert dates from one format to another, and the fixed set of hard-coded date formats supported by Excel 2010 do not match the input and output formats. Thus, the user solicited help on a forum.*

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

We can encode the required background knowledge for the date data type in two tables, namely a `Month` *table with 12 entries:* $(1, \texttt{January}), \ldots, (12, \texttt{December})$ *and a* `DateOrd` *table with 31 entries* $(1, \texttt{st}), (2, \texttt{nd}), \ldots, (31, \texttt{st})$.

3.3 Number Transformations

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

Following is an example of a task that requires number transformations [29].

EXAMPLE 6 (DURATION MANIPULATION). *An Excel user needed to convert the "raw data" in the input column to the lower range of the corresponding "30-min interval" as shown in the output column.*

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

In this example, we first need to be able to extract the hour component of the duration in input column v_1 and perform a rounding operation on the minute part of the input to round it to the lower 30-min interval. Our tool [29] learns the desired transformation using only the first two input-output examples.

For automation of such tasks, we have extended the domain-specific language mentioned in §3.1 with a number transformation language that can describe formatting and rounding transformations [29].

3.4 Table Layout Transformations

End-users often transform a spreadsheet table not by changing the data stored in the cells of a table, but instead by changing how the cells are grouped or arranged. In other words, users often transform the *layout* of a table [12].

EXAMPLE 7. *The following example input table and subsequent example output table were provided by a novice on an Excel user help thread to specify a layout transformation:*

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

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

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

We have designed a domain-specific language that can express a rich set of practical transformations over tabular data (such as the one required to map the example input table to the example output table shown in Example 7). Our inductive synthesizer for table layout transformations [12] can synthesize scripts in this language from example input and output tables. The user can then apply the synthesized script to transform the layout of possibly a much larger input table.

4. INTELLIGENT TUTORING SYSTEMS

The need for use of technology in education cannot be under-stated. Classroom sizes are increasing and the cost of education is rising. Recent trends such as advent of devices with new form factors (e.g., tablets and smartphones that are speech/ink/touch enabled), and social networking have created a favorable opportunity in the marketplace. This has given rise to several online educational initiatives such as Khan Academy [1], Udacity [5], MITx [2], and Stanford online classes [4], which have the potential of providing quality education to a large number of students. However, the presence of a larger number of students enrolled in a course further increases the challenge of providing personalized feedback to students. This is where intelligent tutoring systems can play a pivotal role. They can help both students and teachers with various repetitive and structured tasks in education such as *feedback and grading*, *problem generation* (of a certain difficulty level and that exercise use of certain concepts), *solution generation* (to any given problem in the subject domain), and even *digital content creation* that provides a better experience than pen and paper. After all, teachers are teaching the same material over and over again and have to grade similar kinds of mistakes made by students year after year.

It turns out that the technology of synthesis from examples can help with each of the above-mentioned aspects of an intelligent tutoring system. We illustrate this by presenting below few applications from different subject domains.

4.1 Solution Generation (for Geometry)

Geometry is regarded to be one of the most difficult as well as important subjects in high-school curriculum. Geometry education is supposed to help exercise logical abilities of the left-brain, visualization abilities of the right-brain, and hence enables students to make the two connect and work together as one. In this section, we consider the problem of synthesizing high-school ruler/compass based geometry constructions [11].

Geometry constructions are essentially straight-line programs that manipulate geometry objects (points, lines, and circles) using ruler and compass operators. Hence, the problem of synthesizing geometry constructions is very similar to the problem of synthesizing bit-vector algorithms (§2), which are straight-line programs over bit-vector operators.

EXAMPLE 8. *Consider the problem of constructing a triangle, given its base L (with end-points p_1 and p_2), a base angle a, and sum of the other two sides r. We wish to synthesize a program S that performs the above construction (i.e., obtains the third vertex p) using ruler and compass instructions.*

The following example represents one such construction (where p_1, p_2, L, r, a constitute the input tuple, and p constitutes the output).

$$L = \texttt{Line}(p_1 = \langle 81.62, 99.62\rangle, p_2 = \langle 99.62, 83.62\rangle)$$
$$r = 88.07 \qquad a = 0.81 \text{ radians}$$
$$p = \langle 131.72, 103.59\rangle$$

From this example, our tool [11] synthesizes the following program over an extended library of ruler/compass operations.

```
ConstructTriangle(p1,p2,L,r,a):
  L1 := ConstructLineGivenAngleLinePoint(L,a,p1);
  C1 := ConstructCircleGivenPointLength(p1,r);
  (p3, p4) := LineCircleIntersection(L1,C1);
  L2 := PerpendicularBisector2Points(p2,p3);
  p5 := LineLineIntersection(L1,L2);
  return p5;
```

Search Algorithm.

Given an input-output example, the synthesizer performs a brute-force search by applying all possible ruler/compass operators on the concrete input state and derivative states in a recursive manner in an attempt to reach the concrete output state. We use two key ideas in order to make this search feasible in practice. First, we extend the basic instruction set of the geometry programming language with higher level primitives (such as the ones used in Example 8), which represent common constructions found in textbook chapters. This is inspired by how humans use their experience and knowledge gained from previous chapters to perform complicated constructions. This transforms the search space to one that has a smaller depth but larger width. Second, we prune the forward exhaustive search using a goal-directed heuristic. This is inspired by backward reasoning often performed by humans in doing such constructions. For example,

this heuristic might suggest that if a construction results in a line L_1 that passes through a given output point P, then it may be useful to apply that construction (since we might be able to construct another line L_2 in the future that also passes through P, and hence intersection of L_1 and L_2 can yield desired point P).

User Interaction Model.

A construction that works for a randomly chosen input-output model of the geometry construction problem, will also work for any arbitrary input with high probability over the choice of the random model. (This result follows from an extension of the classic polynomial identity testing theorem [27] to expressions with arbitrary algebraic operators [11]). Such a random model can be generated by using some numerical solver over the logical constraints that specify the construction problem (after randomly fixing values for the independent variables).

4.2 Problem Generation (for Algebra)

Generating fresh problems that involve using the same set of concepts and have the same difficulty level as the problems discussed in the class, is a tedious task for the teacher. Even motivated students want to have access to such fresh *similar* problems, when they fail to solve a given problem and had to look at the solution. We desire to automatically generate fresh problems that are similar to a given problem, where the user interactively works with the tool to fine-tune the notion of similarity.

We have developed a synthesis algorithm that can take a proof problem p in Algebra, and can synthesize new problems that are similar to p [30]. Our technique is fairly general and is applicable to several subfields of Algebra such as Multivariate Polynomials, Trigonometry, Summations over Series, applications of Binomial theorem, Calculus (Integration and Differentiation), Matrices and Determinants.

Example 9 (Limit/Series).

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{2i^2+i+1}{5^i} = \frac{5}{2}$$

Given the above ("example") problem, our tool [26] generated several similar problems, some of which are:

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{3i^2+2i+1}{7^i} = \frac{7}{3}$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{3i^2+3i+1}{4^i} = 4$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{i^2}{3^i} = \frac{3}{2}$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{5i^2+3i+3}{6^i} = 6$$

Example 10 (Trigonometry). *From [23].*

$$\frac{\sin A}{1+\cos A} + \frac{1+\cos A}{\sin A} = 2\csc A$$

Given the above problem, our tool generated 8 similar problems, of which 2 are present in the same textbook, and the remaining 6 are new problems with similar proof strategy. We list some of the new problems below:

$$\frac{\cos A}{1-\sin A} + \frac{1-\sin A}{\cos A} = 2\tan A$$

$$\frac{\cos A}{1+\sin A} + \frac{1+\sin A}{\cos A} = 2\sec A$$

$$\frac{\cot A}{1+\csc A} + \frac{1+\csc A}{\cot A} = 2\sec A$$

$$\frac{\tan A}{1+\sec A} + \frac{1+\sec A}{\tan A} = 2\csc A$$

$$\frac{\sin A}{1-\cos A} + \frac{1-\cos A}{\sin A} = 2\cot A$$

Example 11 (Integration). *From [39].*

$$\int (\csc x)\,(\csc x - \cot x)\,dx = \csc x - \cot x$$

where $x \in \left[\frac{\pi}{6}, \frac{\pi}{3}\right]$.

Given the above problem, our tool generated 8 similar problems, of which 4 are present in the tutorial and the remaining 4 are new problems. We list some of the problems below:

$$\int (\tan x)\,(\cos x + \sec x)\,dx = \sec x - \cos x$$

$$\int (\sec x)\,(\tan x + \sec x)\,dx = \sec x + \cos x$$

$$\int (\cot x)\,(\sin x + \csc x)\,dx = \sin x - \csc x$$

Example 12 (Determinants). *From [17].*

$$\begin{vmatrix} (x+y)^2 & zx & zy \\ zx & (y+z)^2 & xy \\ yz & xy & (z+x)^2 \end{vmatrix} = 2xyz(x+y+z)^3$$

where $x, y, z \in [0, 1]$.

Given the above problem, our tool generated 6 similar problems, of which 3 are already present in the same textbook and the remaining 3 are new. Some of the generated problems are given below (Here $s = (x+y+z)/2$*):*

$$\begin{vmatrix} x^2 & (s-x)^2 & (s-x)^2 \\ (s-y)^2 & y^2 & (s-y)^2 \\ (s-z)^2 & (s-z)^2 & z^2 \end{vmatrix} = 2s^3(s-x)(s-y)(s-z)$$

$$\begin{vmatrix} y^2 & x^2 & (y+x)^2 \\ (z+y)^2 & z^2 & y^2 \\ z^2 & (x+z)^2 & x^2 \end{vmatrix} = 2(xy+yz+zx)^3$$

$$\begin{vmatrix} -xy & yz+y^2 & yz+y^2 \\ zx+z^2 & -yz & zx+z^2 \\ xy+x^2 & xy+x^2 & -zx \end{vmatrix} = xyz(x+y+z)^3$$

$$\begin{vmatrix} yz+y^2 & xy & xy \\ yz & zx+z^2 & yz \\ zx & zx & xy+x^2 \end{vmatrix} = 4x^2y^2z^2$$

User Interaction Model.

The user provides an example problem p. The tool abstracts problem p to a query Q, which defines the search space of potential problems. The query execution engine generates a set of valid problems from this search space. If the user is not satisfied with some of the problems in the

generated set, the user can either modify the query manually, or indicate undesirable problems after which the tool can refine the query automatically. The process of query execution is then repeated on the new query.

Search Algorithm.

The query execution engine systematically enumerates the entire search space defined by Q (by instantiating all possible resolutions of choice nodes in Q). The size of this search space can be huge (exponential in the number of choice nodes in Q), and hence we need an efficient algorithm to quickly check whether or not a potential problem is a valid problem. For this, the tool performs randomized (approximate) testing (using finite-precision arithmetic), the probabilistic correctness of which follows from a generalized polynomial identity testing theorem [11].

4.3 Content Creation (for Mathematical Text)

Inputting mathematical text into a computer remains a painful task, despite several improvements to document editing systems over the years. Markup languages like LaTeX lead to unreadable text in encoded form, while WYSIWYG editors like Microsoft Word require users to change cursor position several times, and switch back and forth between keyboard and mouse input.

We believe that mathematical text, like several human created artifacts, is quite structured with low entropy, and hence it is amenable to both encryption and prediction. We observe that mathematical text is often organized into sessions, each consisting of mutually related expressions with an inherent progression. Examples of such session include a lengthy equation, a symbolic matrix, a solution to a problem, a list of problems in an exercise, and a set of related rules and axioms. Using this observation, we phrase a synthesis-from-example problem that can predict what sub-term the user is likely to input next. Such intellisense algorithms can be important components of human-computer interfaces for inputting mathematical text into a computer, be it through speech, touch, keyboard or multi-modal interfaces.

Consider the following lengthy equation:

EXAMPLE 13 (EQUATION).

$$\tan A \cdot \tan 2A \cdot \mathbf{\tan 3A} = \tan A + \mathbf{\tan 2A + \tan 3A} \quad (1)$$

The arguments of the trigonometric function in the LHS follow the sequence: $A, 2A, 3A$. Thus, if the user has already typed "$\tan A \cdot \tan 2A \cdot$", our tool [26] suggests "$\tan 3A$" as the next term, saving about 1/3 of typing time for the LHS.

A related, but different problem, arises in the RHS of the above equation. Assume that the user types the RHS after typing the LHS. The RHS of this equation is a transformation of LHS, where operator "$\cdot$" is replaced with the operator "$+$". Thus, after the user has typed "$\tan A +$", our tool suggests "$\tan 2A + \tan 3A$" as a choice for auto-completion, saving about 2/3 of typing time for the RHS.

Following are some examples of a symbolic matrix with an inherent progression pattern.

EXAMPLE 14 (SYMBOLIC MATRIX).

$$\begin{pmatrix} yz - x^2 & zx - y^2 & \boldsymbol{xy - z^2} \\ zx - y^2 & \boldsymbol{xy - z^2} & \boldsymbol{yz - x^2} \\ xy - z^2 & \boldsymbol{yz - x^2} & \boldsymbol{zx - y^2} \end{pmatrix} \quad (2)$$

$$\begin{pmatrix} A_1 \sin^3\alpha & B_1 \sin^3\beta & \boldsymbol{C_1 \sin^3\gamma} \\ A_2 \sin\alpha & \boldsymbol{B_2 \sin\beta} & \boldsymbol{C_2 \sin\gamma} \end{pmatrix} \quad (3)$$

The bold text denotes the terms that can be predicted by our tool.

Following is an example of a solved exercise taken from a textbook [23].

EXAMPLE 15 (SEMANTIC INTELLISENSE IN A SOLUTION).

Prove: $(\csc x - \sin x)(\sec x - \cos x)(\tan x + \cot x) = 1$

$$\begin{aligned} \textit{L.H.S.} &= \left(\frac{1}{\sin x} - \sin x\right)\left(\frac{\mathbf{1}}{\mathbf{\cos x}} - \mathbf{\cos x}\right)\left(\frac{\mathbf{\sin x}}{\mathbf{\cos x}} + \frac{\mathbf{\cos x}}{\mathbf{\sin x}}\right) \\ &= \left(\frac{1 - \sin^2 x}{\sin x}\right)\left(\frac{\mathbf{1 - \cos^2 x}}{\mathbf{\cos x}}\right)\left(\frac{\mathbf{\sin^2 x + \cos^2 x}}{\mathbf{\cos x \sin x}}\right) \\ &= \left(\frac{\cos^2 x}{\sin x}\right)\left(\frac{\mathbf{\sin^2 x}}{\mathbf{\cos x}}\right)\left(\frac{\mathbf{1}}{\mathbf{\cos x \sin x}}\right) \\ &= 1 \end{aligned}$$

The bold text denotes the terms that can be predicted by our tool.

User Interaction Model.

The user types the mathematical text in a normal left to right fashion. The synthesis tool learns the inherent pattern in the text using the surrounding context, and provides ranked suggestions for auto-completion.

Search Algorithm.

We note that mathematical text inside a session can be thought of as a collection of sequences. For instance, in Equation 1 both the LHS and RHS of the equation correspond to the sequence $\langle \tan A, \tan 2A, \tan 3A \rangle$ with different separators ($\cdot$ in the LHS and $+$ in the RHS). In Equations 2 and 3, we identify each row of the matrix as a sequence. We perform pre-processing on the input stream to identify such sequences. The prediction problem then reduces to predicting the next term of a sequence, given a few initial terms of the sequence and a relevant context (previous sequences). We define a language of term transformations, and present an efficient algorithm to synthesize term transformations from examples, and use this algorithm for sequence prediction [26]. The general approach here is similar to the one presented in §3.

4.4 Grading (for Programming)

Manually grading programming problems is very challenging because of the need to reason about semantic equivalence of the student's solution with that of a reference solution. Recent advances in testing technology have been used to automatically generate test inputs on which the student's solution yields an answer that is different from that produced by the reference implementation [3]. However, this technique still does not come close to providing direct feedback to the student about what exactly is wrong with a given program and how to fix it.

We have developed a program synthesis technology to automatically determine minimal fixes to the student's solution that will make it match the behavior of a reference

solution provided by the teacher [31]. This technology provides a basis for assigning partial credit for incorrect solutions by giving a quantitative measure of the degree to which the solution was incorrect, and makes it possible to provide students with precise feedback about what they did wrong. Our methodology involves generating abstract error models (manually or automatically) from few solutions graded by the teacher (which serve as "examples" of corrections), and then automatically grading other solutions using those error models.

EXAMPLE 16. *Suppose we obtain the following abstract error model from grading few students' solutions for the array reversal problem, where each correction rule indicates a potential replacement for the expression on the left side by one of the expressions on the right side.*

$$\begin{aligned}
\text{INDF}\quad & v[a] && \rightarrow && v[\{a\{+,-\}\{1,?a\}, v.\texttt{Length} - a - 1\}] \\
\text{INITF}\quad & v = \texttt{Int} && \rightarrow && v = \{n+1, n-1, 0\} \\
\text{CONDF}\quad & b_0\ op_b\ a_0 && \rightarrow && b_0'\ \widetilde{op}_b\ \{a_0\{+,-\}1, 0, 1, ?a_0\} \\
& \multicolumn{4}{c}{\text{where } \widetilde{op}_b = \{<, >, \leq, \geq, ==, \neq\}} \\
\text{INCF}\quad & v{+}{+} && \rightarrow && \{{+}{+}v, --v, v--\} \\
\text{RETF}\quad & \texttt{return}\ v && \rightarrow && \texttt{return}\ ?v
\end{aligned}$$

Consider the following incorrect solution submitted by a student (on the Pex4Fun website [3]) to the array reversal problem.

```
1. public static int[] Puzzle(int[] b) {
2.     for (int i=1; i<=b.Length/2; i++){
3.         int temp = b[i];
4.         b[i] = b[b.Length-i];
5.         b[b.Length-i] = temp;
6.     }
7.     return b;
8. }
```

Our tool [31] can produce the following fixes, each of which can convert the above attempt into a semantically correct solution.

- *Fix 1: Replace b[i] by b[i-1] (Rule* INDF*) in lines 3 and 4.*
- *Fix 2: Replace b[b.Length-i] by b[b.Length-i-1] (Rule* INDF*) in lines 4 and 5, and Replace loop initialization i=1 by i=0 (Rule initF) in line 2.*

Search Algorithm.

Given a set of correction rules (abstracted from example corrections), and a student's solution, the synthesizer needs to explore the space of all possible variations to the student's solution based on the correction rules. We reduce the problem of generating minimal fixes to that of completing a partial program (programs with holes that take values from a finite domain). We then use the Sketch synthesizer [32] that uses a SAT-based algorithm to complete partial programs based on a given specification.

User Interaction Model.

The teacher picks an ungraded answer script to grade. The (example) corrections on the graded version of that answer script are abstracted into more general correction rules, which are then added to the error model (which is initialized to the empty set). The synthesizer then attempts to grade as many answer scripts as possible with the current error model. The entire process is then repeated until no ungraded answer scripts remain.

5. CONCLUSION

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

We believe that the most interesting applications of the synthesis technology can be in the areas of end-user programming, and intelligent tutoring systems. In this paper, we focused on example based interaction models. Another effective form of interaction can be based on natural language. It remains an open research problem to design intelligent multi-modal interfaces that can take examples, natural language, speech, touch, etc. as input. The solution lies in bringing together various inter-disciplinary technologies that can combine user intent understanding, (possibly unstructured) knowledge bases, and logical reasoning.

Acknowledgments

I would like to thank all my co-authors on the papers that are referenced here: William Harris, Susmit Kumar Jha, Vijay Korthikanti, Oleksandr Polozov, Sriram Rajamani, Sanjit Seshia, Armando Solar-Lezama, Rishabh Singh, Rohit Singh, Ashish Tiwari.

6. REFERENCES

[1] Khan academy. http://www.khanacademy.org/.
[2] Mitx: Mit's new online learning initiative. http://mitx.mit.edu/.
[3] Pex4fun. `http://www.pex4fun.com/`.
[4] Stanford online courses. http://www.stanford.edu/group/knowledgebase/cgi-bin/2011/08/19/free-computer-science-courses-offered-online-by-stanford-engineering-school/.
[5] Udacity: Free online university classes for everyone. http://www.udacity.com/.
[6] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
[7] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
[9] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 2012. (To appear).
[10] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[11] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61, 2011.

[12] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

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

[14] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[15] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Synthesizing switching logic for safety and dwell-time requirement. In *ICCPS*, 2010.

[16] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314, 2002.

[17] M. L. Khanna. *IIT Mathematics*.

[18] D. E. Knuth. *The Art of Computer Programming, Volume IV*.

[19] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[20] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *VL/HCC*, 2004.

[21] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.

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

[23] S. L. Loney. *Plane Trigonometry*. Cambridge University Press.

[24] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[25] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.

[26] O. Polozov, S. Gulwani, and S. Rajamani. Structure and term prediction for mathematical text. Technical Report MSR-TR-2012-7, 2012.

[27] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[28] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012. (To appear).

[29] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012. (To appear).

[30] R. Singh, S. Gulwani, and S. Rajamani. Interactive simultaneous editing of multiple text regions. In *AAAI*, 2012. (To appear).

[31] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated semantic grading of programs. Technical report, 2012.

[32] A. Solar-Lezama. Program synthesis by sketching, 2008.

[33] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[34] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.

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

[36] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, 2009.

[37] J. Walkenbach. *Excel 2010 Formulas*. John Wiley and Sons, 2010.

[38] H. S. Warren. *Hacker's Delight*. Addison-Wesley, '02.

[39] Wiki. Integration Problems. `http://www.math10.com/en/university-math/integrals/2en.html`.