Programming by Demonstration Framework applied to Procedural Math Problems *

Erik Andersen
University of Washington
eland@cs.washington.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Zoran Popović
University of Washington
zoran@cs.washington.edu

Abstract

K-12 mathematics includes many procedures to be learned, such as addition and subtraction, and there are many "buggy" or incorrect procedures that students demonstrate during this learning process. Learning such procedures (both correct and incorrect) from demonstration traces has various applications in computer-aided education. We formalize mathematical procedures as spreadsheet programs, involving loops and conditionals over a given set of base operators, and present a novel algorithm for synthesizing such procedures from demonstrations. Our algorithm is based on dynamic programming and leverages ideas from version-space algebras and template-based program synthesis. Our implementation efficiently synthesized programs to solve 20 common math procedures and reproduce 28 different kinds of bugs that were demonstrated by real students across 9 procedures. Our implementation significantly outperforms SKETCH, a state of the art program synthesizer, on these tasks. We also demonstrate the applicability of our generic program synthesis technology to spreadsheet table transformations, an important domain in end-user programming.

1. Introduction

Many educational objectives in K-12 mathematics require the student to learn a step-by-step deterministic procedure. Examples of such algorithms include addition, subtraction, prime factorization, and finding the greatest common factor (GCF) of two numbers. Expressing such procedures as imperative programs allows for many applications. We can generate sample solutions for practice problems. We can use test-input generation tools like Pex [27] or FShell [8] to explore possible execution pathways and automatically generate problem sets with good coverage. We can also use procedural traces to assemble practice problems into progressions that start easy and grow more difficult [1].

In order for educators to provide accurate and individualized feedback for each student, they need to understand exactly what the student is and is not doing correctly. Thus, incorrect variations of a correct procedure are also important. Many students demonstrate systematic errors across multiple problems [2, 28]. These "bugs" range from small operator swaps, such as using multiplication instead of addition, to large-scale confusion where the structure of the student's algorithm differs greatly from the correct algorithm. If we can describe an incorrect or "buggy" process as a program, we can automatically identify these bugs in student data [28] and give feedback that is specific to that bug. We can identify classes of problems that the student will probably solve incorrectly and design progressions of practice problems in this space. We can demonstrate the error to a teacher in a step-by-step manner, to help the teacher understand exactly what the student is doing wrong.

Gathering all of these programs, both correct and incorrect, is challenging. Educators who interact with students most closely often lack programming skills. There are many variations of K-12 math algorithms found in practice, including differences in notation and graphical representation such as underlining and coloring, variation within an algorithm such as whether carries in addition are written explicitly, and multiple approaches for the same task such as the three different procedures for GCF that we encountered (Figure 1). The space of "buggy" algorithms is massive and continually expanding; Van Lehn has identified over 100 bugs that students demonstrate in subtraction alone [28]. Error diagnosis is typically done by hand and is repeated individually for each student. Therefore, automating the synthesis of such programs is useful.

There have been many attempts to use computation to automate the diagnosis of student errors [23, 28]. Such approaches typically begin with the procedure to be learned and model errors as small preturbations of that procedure. However, a buggy procedure can potentially involve any combination of base concepts and arithmetic operators, assembled into any possible configuration of control structures such as loops and conditionals. In order to capture all of these bugs, we need an effective and automatic way to build hypotheses from the ground up. There is some work on learning "buggy" production rules from incorrect student behavior in intelligent tutors [17], but this cannot capture all of the important characteristics of full imperative procedures.

Ideally, we would be able to synthesize K-12 math programs from a set of representative demonstrations provided by a teacher and "buggy" variations from a set of demonstrations produced by a student who misunderstands the correct procedure. The most relevant techniques in this area are version space algebra [13, 19] and template-based approaches [24–26]. However, they cannot efficiently synthesize programs with nested loops and conditionals, necessitating a novel approach. Existing version space algebras can only handle regular loops without any conditionals and require the user to indicate loop boundaries. Existing template-based techniques require templates to have first-order holes (i.e. integers) so they can be reduced to SAT/SMT constraints.

In contrast, we introduce a novel framework that can synthesize programs with (possibly nested) loops and conditionals. Our algorithm uses dynamic programming and several novel ideas to efficiently search the state space of all programs whose loop-free substructures match a given set of templates, which can be expanded iteratively and automatically until synthesis succeeds. Our system is semi-automatic in the sense that it requires the user to provide the base operators and predicates, although we have found that a small set of operators is sufficient to capture all of our testing benchmarks. Given the correct operators, our system can synthesize programs with arbitrarily nested loop and conditional structures.

The goal of our system is to synthesize *any* program in our procedural language that is consistent with a given set of examples. It is always possible that the synthesized program is not what a teacher intended to demonstrate or what a confused student is really doing, but this is still useful. A teacher can know if the program is correct through testing. If our system learns an incorrect program from a set of correctly-solved practice problems, this is evidence that the practice problem set is not sufficiently complete to resolve ambiguity, possibly leading to misconceptions. If there are multiple

* Microsoft Research Technical Report
Number: MSR-TR-2014-61, Year: 2014

explanations for a student's error, then any of these explanations are possible, and this knowledge is valuable for the teacher.

We present a set of benchmarks showing that our algorithm is able to synthesize 20 correct procedures ranging from 2-14 lines of code. We also demonstrate that our system can synthesize 28 "buggy" procedures that capture 28 different bugs across 9 topics identified by Ashlock [2] from real student data. We show the generality of our method by applying it to spreadsheet table transformations [7], an important domain in end-user programming [6].

The paper makes the following contributions:

- We diagnose misconceptions in student data through program synthesis. Explaining a misconception as a program enables the use of test input generation techniques to generate progressions of practice problems that highlight differences between the buggy program and the correct program.
- We present a generic programming by demonstration framework that can synthesize programs with arbitrary nested loops and conditionals over a given set of operators.
- We present experimental evidence showing that our program synthesis technique can synthesize both correct procedures and buggy procedures demonstrated by real students for a variety of mathematical concepts, and that it outperforms SKETCH [24], a state-of-the-art program synthesis tool, for these examples.
- We apply our generic program synthesis technology to spreadsheet table transformations, an important domain in end-user programming.

2. Motivating Examples

We want to learn both correct and "buggy" K-12 math procedures.

2.1 Correct Procedures

Figure 1 shows three different algorithms for finding the greatest common factor (GCF) of two numbers. These programs vary in terms of their program structure. The first algorithm, Euclid's Algorithm, can be implemented as a single loop with a conditional inside. The true and false branches of this conditional both have two statements. The second algorithm, which we found in an Indian math textbook and we refer to as Successive Division, consists of a single loop with four statements. The third algorithm, which is an adaptation of a different algorithm found in that same textbook that we refer to as Simultaneous Division, has an outer loop and an inner loop. The inner loop consists of a single statement, and the outer loop consists of a computation and the inner loop.

Alought these algorithms contain complex structures such as nested loops and conditionals, the body of any particular loop or conditional is not very complex and typically only contains a few statements. If nested loops are treated as a single statement, then the control-flow skeletons of each loop contain four statements or fewer. This property is true for the majority of grade-school mathematical procedures that we studied. We exploit this structural similarity by defining a small set of template loop skeletons, such as "one statement," "four statements," and "a conditional with two statements in both the true and false branches."

2.2 "Buggy" Procedures

Student errors fall into multiple categories, including careless mistakes, incorrect fact recall, and systematic misconceptions in which the wrong algorithm is used [2, 28]. We focus only on this last class of systematic errors. As our definition of correctness is to synthesize any program in our procedural language that is consistent with all provided examples, we assume that the student has solved all of the provided demonstrations in exactly the same way. Being robust to inconsistent student behavior is certainly important for identification of bugs in the wild, but is beyond the scope of this paper.

```
GCF: Euclid's Algorithm(Sheet T, int I1, int I2)
  Assume T[0,0] contains I1 and T[0,1] contains I2.
1 for (j := 0; T[j,0] ≠ T[j,1]; j := j + 1):
2   if (T[j,0] > T[j,1]):
3     T[j + 1,0] := T[j,0] − T[j,1]; T[j + 1,1] := T[j,1];
4   else:
5     T[j + 1,0] := T[j,0]; T[j + 1,1] := T[j,1] − T[j,0];
6 return T[j,0];

GCF: Successive Division(Sheet T, int I1, int I2)
1 Assume T[0,0] contains I1 and T[0,1] contains I2.
2 for (j := 0; T[2j, j] ≠ 0; j := j + 1):
3   T[2j, j + 2] := Floor(T[2j, j + 1] ÷ T[2j, j]);
4   T[2j + 1, j + 1] := T[2j, j + 2] × T[2j, j];
5   T[2j + 2, j + 1] := T[2j, j + 1] − T[2j + 1, j + 1];
6   T[2j + 2, j + 2] := T[2j, j];
7 return T[2j, j + 1];

GCF: Simultaneous Division(Sheet T, int[] I1)
  Assume that T[0,(0,m)] contains I1[0],..,I1[m].
1 for (j := 0; ¬Coprime(T[j,(0, I1.right)]); j := j + 1):
2   T[j,−1] := LeastPrimeDivisor(T[j,(0, I1.right)]);
3   for (i := 0; i < I1.right; i := i + 1):
4     T[j + 1, i] := T[j, i] ÷ T[j,−1];
5 T[LastRow,−1] := Multiply(T[(0, LastRow − 1),−1]);
6 return T[LastRow,−1];
```

Figure 1. Example algorithms showing variety. These algorithms compute the greatest common factor of two numbers or a set of numbers. Euclid's Algorithm has a conditional inside a loop, with each branch of the conditional having two statements. The Successive Division algorithm has one loop with four statements. The Simultaneous Division algorithm has a nested loop and uses two spreadsheet properties: the rightmost column and the bottom row.

Even if a student is inconsistent, we can examine subsets of practice problems solved by the student to identify if there is a error pattern common to that particular subset.

Ashlock [2] identifies a set of buggy computational patterns for a variety of algorithms that are based on real student data, and this dataset formed the basis for our experiments. For each bug, Ashlock provides a set of 5-8 demonstrations that show the error. We attempted to synthesize a program that could solve all of the provided demonstrations in the way that is shown in the book. We do not seek to explain *why* the student made this error.

We will describe here the four bugs that Ashlock describes for addition. These bugs are defined for problems in which two addends, a_1 and a_2, are added.

- A_W_1 (page 34 in [2]): Add each column and write the sum below each column, even if it is greater than nine.
- A_W_2 (page 35): Add each column, from left to right. If the sum is greater than nine, write the tens digit beneath the column and the ones digit above the column to the right.
- A_W_3 (page 36): Only applies to problems in which a_1 has two digits and a_2 has either two digits or one digit. If a_2 has one digit, then add all three visible digits and write the sum. If a_1 and a_2 both have two digits, add each column normally.
- A_W_4 (page 37): Only applies to problems in which a_1 has two digits and a_2 has one digit. Add in a manner similar to multiplication. For each column, moving from right to left, add the digit of a_1 in that column to a_2. Carry if the sum is greater than nine and include in the next sum.

All of these bugs have a clear procedural meaning and can be captured as a program. They all use the same operators as the correct addition algorithm (add, add and take the ones digit, add and take the tens digit) but differ in terms of their control

structure. A_W_3 clearly involves a conditional at the outer level, while A_W_2 involves conditionals inside a loop.

3. Formalism

In this section, we formalize a mathematical problem in §3.1, its solution in §3.2, a procedural language to automatically compute such solutions in §3.3, and the inductive synthesis problem to automatically synthesize such procedures from examples in §3.4.

3.1 Mathematical Problem Instance

Our examination of textbooks for K-12 math has revealed that many topics can be expressed as computation between cells of a spreadsheet. Although this abstraction does not cover problems that include text, geometric shapes, and symbolic computation (such as algebra), we found it to be very useful as a basis for synthesis and widely applicable. Therefore, we abstract a mathematical problem as a spreadsheet T partitioned into a tuple of rectangular regions $(I_1, \cdot\cdot, I_m)$, each of which contain the corresponding original inputs formatted appropriately. A spreadsheet T is a two-dimensional array of integers that stretches infinitely in all directions. A region I has four integral attributes: top, $left$, $bottom$, $right$. These attributes denote the row and column coordinates of the top-left corner and bottom-right corner respectively. We fix the origin of the spreadsheet to be the top-left corner of the first input region I_1.

These regions have one of the following types depending on the kind of content they hold. A 0-dimensional region (a single cell) has the type "int" and holds an integer. A 1-dimensional region can either be of type "array int" if it holds an array of integers or "digits int" if it holds the digits of an integer. A 2-dimensional region can either be of type "matrix int" if it holds a matrix of integers or "digits array int" if it holds the digits of the integers from an array. Regions are typically aligned with each other; for example, addition problems have two right-justified 1-dimensional regions.

Although copying inputs to appropriate regions in the spreadsheet is an important part of the problem solving experience, we omit this phase for simplicity of presentation. We note that we can extend our framework to incorporate it by learning straight-line code that uses array copy operators and type convertors to convert an integer into an array of digits and vice-versa.

For example, if we use GCF: Successive Division to find the GCF of 762 and 1270, the input is a spreadsheet with the following two regions (of type "int") indicated in blue and green:

762	1270

The variable values of these regions are as follows:

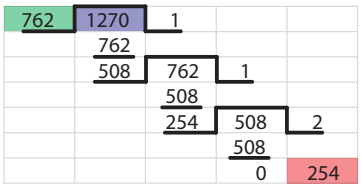
Input	top	$left$	$bottom$	$right$
762	0	0	0	0
1270	0	1	0	1

3.2 Solution to a Mathematical Problem Instance

In our framework, the solution to a mathematical problem instance is expressed as a trace `Tr` and a highlighted region J. A *trace* is a demonstration of the steps required to generate the final result. It is sequence of steps in which each step identifies a spreadsheet cell, a value to be written in that cell, and any enumerated tags associated with that value, such as font color, emphasis, or animation. More formally, a trace is an array of tuples (row, $column$, $value$, $emphasis$). The index of each tuple in the array represents its timestamp. The trace for the above problem is as follows:

Time	1	2	3	4	5	6	7	8	9	10	11	12
$place$	(0,2)	(1,1)	(2,1)	(2,2)	(2,3)	(3,2)	(4,2)	(4,3)	(4,4)	(5,3)	(6,3)	(6,4)
$value$	1	762	508	762	1	508	254	508	2	508	0	254
$emph$	Div		Sub		Div		Sub		Div		Sub	

The highlighted region J is an indication of where the answer was written in the spreadsheet. This is similar to "circling" the answer, a common practice when practice problems are worked out on paper. This output region is defined in the same way as input regions in §3.1. The state of the spreadsheet after the above problem has been solved is shown below, with the answer highlighted in red:



$$
\begin{aligned}
\text{Program } P &:= \texttt{Sequence}(S_1, S_2, \cdot\cdot, S_m) \\
\text{Statement } S &:= \texttt{Update}(k_1, k_2, e, w) \mid \texttt{Loop}(j, b, P) \mid \texttt{Cond}(b, P_1, P_2) \\
\text{Integer Expression } e &:= \texttt{F}(a_1, \cdot\cdot, a_m) \\
\text{Argument } a &:= \texttt{Select}(k_1, k_2) \mid \texttt{SelectRow}(k_1, k_2, k_3) \\
&\mid \texttt{SelectColumn}(k_1, k_2, k_3) \\
\text{Boolean Expr. } b &:= j \texttt{ relop } k \mid j \texttt{ relop Select}(k_1, k_2) \mid \texttt{G}(a_1, \cdot\cdot, a_m) \\
\text{Emphasis } w &:= \text{Enumerated type} \\
\text{Integer Linear Expression } k &:= c + c_1 v_1 + \cdot\cdot + c_m v_m
\end{aligned}
$$

Figure 2. Syntax of Programs.

3.3 Procedure

In this section, we introduce a fairly general-purpose language that can be used to compute the kind of solutions specified in §3.2, when given as input the kind of problem specification specified in §3.1. The types in our language are scalars such as digits and numbers, and vectors such as arrays of digits and arrays of numbers.

Our language includes a set of base operators that take as input either an integer, an array of integers, or a matrix of integers and output an integer or a Boolean. The values of array types are constructed using a standard spreadsheet range construct so that all arguments to base operators are integers. We provide a set of base operators that are common across many mathematical procedures such as addition and subtraction. Once our system learns a new procedure, this procedure can become an operator for learning future procedures. The basic constructs of our language are the operators mentioned above, as well as conditionals, loops, and emphasis such as color, annotations, and animations.

Figure 2 describes the syntax of our language. Programs P contain a `Sequence` of statements S. Statements can take one of three forms. One is $\texttt{Update}(k_1, k_2, e, w)$, which writes the value computed by e into the grid cell at row k_1 and column k_2 with emphasis w. w is an enumerated type such as "make bold", "draw line below and over and italicize", "flash red", etc. Another is $\texttt{Loop}(j, b, P)$, which represents a loop over program body P that continually iterates and increments a loop iterator variable j, which starts at 0, until Boolean expression b evaluates to false. Finally, it can be $\texttt{Cond}(b, P_1, P_2)$, which represents a conditional branch that executes P_1 if b evaluates to true or P_2 if b evaluates to false.

Integer expressions e are defined as $\texttt{F}(a_1, \cdot\cdot, a_m)$ where `F` is a function that returns an integer and takes $a_1, \cdot\cdot, a_m$ as input. Examples of these functions include addition and subtraction. An argument a reads from a spreadsheet cell and has three forms. The first, $\texttt{Select}(k_1, k_2)$, reads the value from the spreadsheet T in scope located at $T[k_1, k_2]$. $\texttt{SelectRow}(k_1, k_2, k_3)$ reads an array of integers from T in row k_1 from column k_2 to k_3. Similarly, $\texttt{SelectColumn}(k_1, k_2, k_3)$ reads an array of integers in column k_3 from row k_1 to k_2. Boolean expressions b have three types. The first type is j `relop` k, where `relop` is a relational operator $\in \{\leq, <, =, >, \geq\}$, and this operator compares a loop itera-

$$
\begin{array}{rcl}
\text{Angelic Program } \tilde{P} & := & \texttt{Sequence}(\tilde{S}_1, \tilde{S}_2, \cdot\cdot, \tilde{S}_m) \\
\text{Angelic Statement } \tilde{S} & := & \{\tilde{V}_1, \cdot\cdot, \tilde{V}_m\} \mid \tilde{W} \\
\text{Angelic Conditional St. } \tilde{V} & := & \texttt{Cond}(\tilde{b}, \tilde{P}_1, \tilde{P}_2) \\
\text{Angelic Non-conditional St. } \tilde{W} & := & \{\tilde{R}_1, \cdot\cdot, \tilde{R}_m\} \mid \bot \\
\text{where } \tilde{R} & := & \texttt{Update}(\tilde{k}_1, \tilde{k}_2, \tilde{E}, w) \\
& \mid & \texttt{Loop}(j, \tilde{b}, \tilde{P}) \\
\text{Angelic Boolean Expression } \tilde{b} & := & \{\tilde{h}_1, \cdot\cdot, \tilde{h}_m\} \mid \bot \\
\text{Angelic Boolean Constant } \tilde{h} & := & \{(\sigma_1, d_1), \cdot\cdot, (\sigma_m, d_m)\} \\
\text{Angelic Expression } \tilde{E} & := & \{\tilde{e}_1, \cdot\cdot, \tilde{e}_m\} \\
\text{where } \tilde{e} & := & \texttt{F}(\tilde{a}_1, \cdot\cdot, \tilde{a}_m) \\
\text{Angelic Argument } \tilde{a} & := & \texttt{Select}(\tilde{k}_1, \tilde{k}_2) \\
& \mid & \texttt{SelectRow}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3) \\
& \mid & \texttt{SelectColumn}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3) \\
\text{Angelic Integer Constant } \tilde{k} & := & \{(\sigma_1, c_1), \cdot\cdot, (\sigma_m, c_m)\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{Sequence}(\tilde{S}_1, \cdot\cdot, \tilde{S}_m)]\!] & = & \{\texttt{Sequence}(S_1, \cdot\cdot, S_m) \mid S_1 \in [\![\tilde{S}_1]\!], \cdot\cdot, S_m \in [\![\tilde{S}_m]\!]\} \\
[\![\{\tilde{V}_1, \cdot\cdot, \tilde{V}_m\}]\!] & = & \bigcup_{i=1}^{m} [\![\tilde{V}_i]\!] \\
[\![\texttt{Cond}(\tilde{b}, \tilde{P}_1, \tilde{P}_2)]\!] & = & \{\texttt{Cond}(b, P_1, P_2) \mid b \in [\![\tilde{b}]\!], P_1 \in [\![\tilde{P}_1]\!], P_2 \in [\![\tilde{P}_2]\!]\} \\
[\![\{\tilde{R}_1, \cdot\cdot, \tilde{R}_m\}]\!] & = & \bigcup_{i=1}^{m} [\![\tilde{R}_i]\!] \\
[\![\texttt{Update}(\tilde{k}_1, \tilde{k}_2, \tilde{E}, w)]\!] & = & \{\texttt{Update}(k_1, k_2, e, w) \mid k_1 \in [\![\tilde{k}_1]\!], k_2 \in [\![\tilde{k}_2]\!], e \in [\![\tilde{E}]\!]\} \\
[\![\texttt{Loop}(j, \tilde{b}, \tilde{P})]\!] & = & \{\texttt{Loop}(j, b, P) \mid b \in [\![\tilde{b}]\!], P \in [\![\tilde{P}]\!]\} \\
[\![\{\tilde{e}_1, \cdot\cdot, \tilde{e}_m\}]\!] & = & \bigcup_{i=1}^{m} [\![\tilde{e}_i]\!] \\
[\![\texttt{F}(\tilde{a}_1, \cdot\cdot, \tilde{a}_m)]\!] & = & \{\texttt{F}(a_1, \cdot\cdot, a_m) \mid a_1 \in [\![\tilde{a}_1]\!], \cdot\cdot, a_m \in [\![\tilde{a}_m]\!]\} \\
[\![\texttt{Select}(\tilde{k}_1, \tilde{k}_2)]\!] & = & \{\texttt{Select}(k_1, k_2) \mid k_1 \in [\![\tilde{k}_1]\!], k_2 \in [\![\tilde{k}_2]\!]\} \\
[\![\{(\sigma_1, c_1), \cdot\cdot, (\sigma_m, c_m)\}]\!] & = & \{\texttt{M} \mid \texttt{M} \text{ is a linear function over loop iterators in} \\
& & \sigma \text{ such that } \texttt{M}(\sigma_i) = c_i \text{ for all } 1 \le i \le m\} \\
[\![\{(\sigma_1, d_1), \cdot\cdot, (\sigma_m, d_m)\}]\!] & = & \{\texttt{N} \mid \texttt{N} \text{ is a Boolean function over loop iterators in} \\
& & \sigma \text{ such that } \texttt{N}(\sigma_i) = d_i \text{ for all } 1 \le i \le m\}
\end{array}
$$

Figure 3. Syntax & semantics of angelic programs. $[\![\texttt{SelectRow}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3)]\!]$ & $[\![\texttt{SelectColumn}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3)]\!]$ look like $[\![\texttt{Select}(\tilde{k}_1, \tilde{k}_2)]\!]$.

tor variable j and an integer linear expression k. The second is j `relop` $\texttt{Select}(k_1, k_2)$, which compares a loop iterator variable j to a value read from the spreadsheet. The last type is $\texttt{G}(a_1, \cdot\cdot, a_m)$, in which `G` is a function that returns a boolean value and takes $a_1, \cdot\cdot, a_m$ as input. Note that `G` also includes binary `relop` operators. Integer linear expressions k are linear functions over the integer variables in scope, expressed as a sum of an integer constant c and a set of variables v_i with integer coefficients c_i. These variables can be either a loop iterator variable like j or a property of one of the input regions described in §3.1 such as $I.right$.

To make our examples easier to read, we write them in pseudocode instead of the syntax described in Figure 2. We write $\texttt{Loop}(j, b, P)$ as `for (`$j := 0$`; `b`; `$j := j + 1$`) { `P` }`. Similarly, we write $\texttt{Cond}(b, P_1, P_2)$ as `if (`b`) { `P_1` } else { `P_2` }`. $\texttt{Select}(k_1, k_2)$ is written as $T[k_1, k_2]$, where T is the `Sheet` in scope. $\texttt{SelectRow}(k_1, k_2, k_3)$ and $\texttt{SelectColumn}(k_1, k_2, k_3)$ are written as $T[k_1, (k_2, k_3)]$ and $T[(k_1, k_2), k_3]$, respectively. We write $\texttt{Update}(k_1, k_2, e, w)$ as $T[k_1, k_2] := e$. If `F` is addition, we write $\texttt{F}(a_1, a_2)$ as $a_1 + a_2$, and do the same for similar operators. Figure 1 contains statements of the form $\texttt{Return}(e)$; these are written as $\texttt{Update}(0, 0, \texttt{Return}(e), none)$.

3.4 Synthesis Problem

The previous sections formally defined an input problem (§3.1) and an output solution (§3.2) for a mathematical procedure (§3.3). We now seek to synthesize such procedures from input-output examples. More formally, given a set of examples $\{Z_1, \cdot\cdot, Z_m\}$, where each example consists of an input tuple $(\texttt{Sheet}, I_1, \cdot\cdot, I_m)$ and an output tuple $(\texttt{Tr}, J)$, the goal is to synthesize a procedure P that is consistent with each of these examples. More formally, the procedure P should map the input tuple in each example to the corresponding output tuple. The synthesizer also needs two other inputs: a set of base operators that apply as a single step in the computation, and a set of loop-free control structure templates (§4.2). As templates can be enumerated in an iterative manner, the user *does not* need to provide them. The user *does* need to provide the operators or select from many that we have implemented.

4. Synthesis

We present an algorithm for the synthesis problem introduced in §3.4. We first describe the key ideas before providing details.

Efficient Data Structure For each example, our algorithm learns the set of all programs that are consistent with that example. Our algorithm then intersects all these sets. Since the number of programs in these sets is typically huge, Section §4.1 introduces a data structure that can succinctly represent these programs and support an efficient `Intersect` operation. The key idea behind this data structure is the sharing of common fragments between programs.

Learning Conditionals Section §4.2 describes how our algorithm restricts the set of all consistent programs to those that fit a given set of *templates*, which are loop-free skeletons with explicit control flow and holes for statements (`Update` or `Loop`). Our algorithm iteratively expands the set of templates until it finds a valid program that is consistent with all examples. This approach is motivated by the observation that loop-free templates tend to have a small size, even inside large procedures. A good example of this is the GCF: Simultaneous Division algorithm, described in Figure 1. Although this procedure has complex stuctures like nested loops, the loop-free skeleton only consists of a single statement in the case of the inner loop, and two statements in the case of the outer loop (note that the inner loop is treated as a single statement).

Learning Loops Our dynamic programming algorithm learns the set of all `Loop` statements that are consistent with a given example, as described in Section §4.3. We first learn the set of all loop-free programs, then the set of all programs that have loops of depth at most 1, followed by the programs that have loops of depth at most 2, and so forth. We compute this set for all contiguous subsequences of the example trace; we compute the set of all programs that are consistent with a given subsequence of the output trace after having computed the set of all programs that are consistent with smaller subsequences. In particular, the key inductive step of the synthesis algorithm is to compute the set of all programs of a given loop depth v and that are consistent with a given subsequence of the output trace, after having computed programs of loop depth $v - 1$ for each subsequence of the output trace.

Learning integer linear expressions and Boolean expressions As described in §4.1, our algorithm learns integer linear expres-

$$
\begin{aligned}
\texttt{Intersect}(\texttt{Sequence}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m),\texttt{Sequence}(\tilde{S}'_1,\cdot\cdot,\tilde{S}'_m)) &= \texttt{Sequence}(\texttt{Intersect}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m),\cdot\cdot,\texttt{Intersect}(\tilde{S}'_1,\cdot\cdot,\tilde{S}'_m))\\
\texttt{Intersect}(\texttt{Cond}(\tilde{b},\tilde{P}_1,\tilde{P}_2),\texttt{Cond}(\tilde{b}',\tilde{P}'_1,\tilde{P}'_2)) &= \{\texttt{Cond}(\texttt{Intersect}(\tilde{b},\tilde{b}'),\texttt{Intersect}(\tilde{P}_1,\tilde{P}'_1),\texttt{Intersect}(\tilde{P}_2,\tilde{P}'_2)),\\
&\quad \texttt{Cond}(\texttt{Intersect}(\tilde{b},\tilde{b}'),\texttt{Intersect}(\tilde{P}_1,\tilde{P}'_2),\texttt{Intersect}(\tilde{P}_2,\tilde{P}'_1))\}\\
\texttt{Intersect}(\{\tilde{V}_1,\cdot\cdot,\tilde{V}_m\},\{\tilde{V}'_1,\cdot\cdot,\tilde{V}'_{m'}\}) &= \bigcup_{1\le j\le m,1\le j'\le m'} \texttt{Intersect}(\tilde{V}_j,\tilde{V}'_{j'})\\
\texttt{Intersect}(\{\tilde{R}_1,\cdot\cdot,\tilde{R}_m\},\{\tilde{R}'_1,\cdot\cdot,\tilde{R}'_{m'}\}) &= \bigcup_{1\le j\le m,1\le j'\le m'} \texttt{Intersect}(\tilde{R}_j,\tilde{R}'_{j'})\\
\texttt{Intersect}(\texttt{Update}(\tilde{k}_1,\tilde{k}_2,\tilde{E},w),\texttt{Update}(\tilde{k}'_1,\tilde{k}'_2,\tilde{E}',w)) &= \texttt{Update}(\texttt{Intersect}(\tilde{k}_1,\tilde{k}'_1),\texttt{Intersect}(\tilde{k}_2,\tilde{k}'_2),\texttt{Intersect}(\tilde{E},\tilde{E}'),w)\\
\texttt{Intersect}(\texttt{Loop}(j,\tilde{b},\tilde{P}),\texttt{Loop}(j,\tilde{b}',\tilde{P}')) &= \texttt{Loop}(j,\texttt{Intersect}(\tilde{b},\tilde{b}'),\texttt{Intersect}(\tilde{P},\tilde{P}'))\\
\texttt{Intersect}(\{\tilde{e}_1,\cdot\cdot,\tilde{e}_m\},\{\tilde{e}'_1,\cdot\cdot,\tilde{e}'_{m'}\}) &= \bigcup_{1\le j\le m,1\le j'\le m'} \texttt{Intersect}(\tilde{e}_j,\tilde{e}'_{j'})\\
\texttt{Intersect}(\texttt{F}(\tilde{a}_1,\cdot\cdot,\tilde{a}_m),\texttt{F}(\tilde{a}'_1,\cdot\cdot,\tilde{a}'_m)) &= \texttt{F}(\texttt{Intersect}(\tilde{a}_1,\tilde{a}'_1),\cdot\cdot,\texttt{Intersect}(\tilde{a}_m,\tilde{a}'_m))\\
\texttt{Intersect}(\texttt{Select}(\tilde{k}_1,\tilde{k}_2),\texttt{Select}(\tilde{k}'_1,\tilde{k}'_2)) &= \texttt{Select}(\texttt{Intersect}(\tilde{k}_1,\tilde{k}'_1),\texttt{Intersect}(\tilde{k}_2,\tilde{k}'_2))\\
\texttt{Intersect}(\tilde{k},\tilde{k}') &= \text{Let } temp := (\tilde{k}\cup\tilde{k}') \text{ in if } (\llbracket temp \rrbracket = \emptyset) \text{ return } \top \text{ else return } temp
\end{aligned}
$$

Figure 4. Intersect function. $\texttt{Intersect}(\texttt{SelectRow}(\tilde{k}_1,\tilde{k}_2,\tilde{k}_3),\texttt{SelectRow}(\tilde{k}'_1,\tilde{k}'_2,\tilde{k}'_3))$ and $\texttt{Intersect}(\texttt{SelectColumn}(\tilde{k}_1,\tilde{k}_2,\tilde{k}_3),\texttt{SelectColumn}(\tilde{k}'_1,\tilde{k}'_2,\tilde{k}'_3))$ are very similar to $\texttt{Intersect}(\texttt{Select}(\tilde{k}_1,\tilde{k}_2),\texttt{Select}(\tilde{k}'_1,\tilde{k}'_2))$. For any other case not listed here, Intersect returns $\top$. Additionally, in cases in which Intersect would return $\emptyset$, it will return $\top$ instead.

sions and Boolean expressions in a lazy manner since a required loop variable may not be in scope until the dynamic programming based algorithm reaches the corresponding loop depth. For example, in the GCF: Simultaneous Division algorithm described in Figure 1, the inner loop statement $T[j+1,i+1] := T[j,i+1] \div T[j,0]$ depends on both j and i. The linear functions in this statement cannot be fully realized until the synthesis algorithm has made two full passes. Therefore, we learn integer expressions lazily instead of eagerly. At all times, we maintain the set of constraints imposed by previous invocations of that integer expression, for example, that it evaluated to 0 when $i = 0$ and 1 when $i = 1$.

Since the number of consistent Boolean expressions can be large and maintaining them can be expensive, we delay learning them until the very end. Every time a potential Boolean expression is evaluated, we store the program context, the state of the spreadsheet, and whether the expression evaluated to true or false. In cases where the Boolean expression depends on values in the spreadsheet, the arguments for these expressions are typically found within the substructures of the corresponding loop or conditional. For example, consider the Boolean expressions in GCF: Euclid's Algorithm described in Figure 1: $T[j,0] \neq T[j,1]$ and $T[j,0] > T[j,1]$. The arguments for both of these expressions, $T[j,0]$ and $T[j,1]$, are found in the substructures of the loop and conditional. Therefore, we restrict the search for arguments of Boolean expressions to those that occur in the body of the loop or branches of the conditional. This intuition makes sense: if program branching behavior depends on spreadsheet values, the program probably reads or writes to those locations with `Update` statements.

We define a materialization procedure to find a solution for integer and Boolean constraints. The goal of this process is to take these constraints and generate a program that satsfies them, if one exists. For integer constraints, we use Gaussian elimination to solve the system of linear equations represented by these constraints. For Boolean expressions, we use brute force search that is restricted to arguments that occur in conditional or loop substructures, as described in the previous paragraph. In particular, for $\texttt{Loop}(j,b,P)$, we learn Boolean expressions b by gathering all of the arguments a found in P, and trying all available Boolean operators `G` for all variations of those arguments. For $\texttt{Cond}(b,P_1,P_2)$, we learn b in the same way by looking inside P_1 and P_2.

4.1 Angelic Programs

We present a data structure that allows for succinct representation of a set of programs that share various fragments at multiple levels. We refer to such a set representation as *angelic programs*, motivated by the fact that all programs in this set are consistent with the example observations that induced them and are candidates for the final result. The syntax of the angelic program structure is described in Figure 3. The syntax of angelic programs is similar to the syntax of programs, with a few key differences. First, as the angelic program structure represents a huge set of programs instead of a single program, several angelic structures contain sets of substructures in order to maintain these sets as efficiently as possible. This sharing occurs at four levels: angelic statement $\tilde{S}$ contains a set of $\tilde{V}$, angelic non-conditional statement $\tilde{W}$ contains a set of $\tilde{R}$, angelic boolean expression $\tilde{b}$ contains a set of $\tilde{h}$, and angelic expression $\tilde{E}$ contains a set of $\tilde{e}$. $\top$ represents no program and $\bot$ represents any possible program.

The semantics of angelic programs are described in Figure 3, which precisely formalizes the set of structures that are represented by a corresponding angelic structure. This semantics shows how the angelic program structure represents huge sets of programs efficiently. For example, a sequence of angelic statements $\texttt{Sequence}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m)$ represents the set of sequences that can be constructed by taking one statement each from $\tilde{S}_1,\cdot\cdot,\tilde{S}_m$, which themselves represent sets of programs. Thus, if $\tilde{S}_1$ contains n_1 statements, $\tilde{S}_2$ contains n_2 statements, etc., then $\texttt{Sequence}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m)$ represents a set of size $n_1 \times n_2 \times \cdot\cdot \times n_m$ but uses space proportional to $n_1 + n_2 + \cdot\cdot + n_m$.

Two particularly important components are the angelic integer constant $\tilde{k}$ and angelic boolean constant $\tilde{h}$, which represent a not-yet-determined linear or Boolean expression as a set of constraints over possible expressions. Each $\tilde{k}$ holds a set of constraints (σ_i, c_i) that record key information for a particular invocation of that expression. For each invocation, σ_i is the set of variables in scope and their values, including loop iterators and properties of the input regions I, and c_i is the integer value that the function returned. For example, if $\tilde{k} = (i = 0, 2), (i = 1, 3)$, $\tilde{k}$ represents the set of linear functions `M` that evaluate to 2 when $i = 0$ and 3 when $i = 1$, which contains the function $i + 2$. Angelic Boolean constants are defined similarly, except that d_i is of type Boolean.

Program Template $\mathcal{P}$	:=	`Sequence`$(\mathcal{S}_1, \mathcal{S}_2, \cdot\cdot, \mathcal{S}_m)$
Statement Template $\mathcal{S}$	:=	`<p>` $* W$ | `<p> Cond`$(*b, \mathcal{P}_1, \mathcal{P}_2)$

Figure 5. Syntax of Templates

This data structure supports an efficient `Intersect` operation shown in Figure 4. `Intersect`$(\tilde{P}_1, \tilde{P}_2)$ takes two angelic programs $\tilde{P}_1$ and $\tilde{P}_2$ and returns an angelic program $\tilde{P}'$ such that $\tilde{P}' = \tilde{P}_1 \cap \tilde{P}_2$. Intersect is similarly defined for all of the other substructures in the angelic program syntax. Note that two conditionals can be semantically the same in two cases: when the boolean expression and the two branches match exactly, and when they match after the true and false branches of one of the conditionals are flipped and the boolean expression of that conditional is negated. `Intersect`$(\texttt{Cond}(\tilde{b}, \tilde{P}_1, \tilde{P}_2), \texttt{Cond}(\tilde{b}', \tilde{P}_1', \tilde{P}_2'))$ checks both of these cases. In cases where an initialized angelic structure is intersected with an uninitialized structure $\perp$, as is the case in `Intersect`$(\{\tilde{R}_1, \cdot\cdot, \tilde{R}_m\}, \perp)$, `Intersect` returns the initialized structure. In any case not listed here, Intersect returns $\top$. Additionally, in cases in which Intersect would return $\emptyset$, it returns $\top$ instead.

When we intersect two angelic integer constants $\tilde{k}$ and $\tilde{k}'$, we take the union of the set of constraints (σ_i, c_i) contained within each constant. Similarly, when we intersect two angelic Boolean constants $\tilde{h}$ and $\tilde{h}'$, we take the union of the constraints (σ_i, d_i). As an optimization, we check if it is still possible to materialize the constants after intersection. If not, then `Intersect` returns $\top$.

4.2 Templates and Co-templates

A template is a program structure whose `Update` and `Loop` instructions have been replaced by non-conditional statement holes $*W$ and whose Boolean expressions have been replaced by Boolean holes $*b$. Thus, a template is a loop-free skeleton with explicit control flow and holes `Hole` for each non-conditional statement (`Update` and `Loop`) and for each Boolean expression. More formally, a template is recursively defined as shown in Figure 5. There are two kinds of templates: Program templates $\mathcal{P}$ and Statement templates $\mathcal{S}$. Program templates $\mathcal{P}$ have a sequence of statement templates $\mathcal{S}$. Statement templates $\mathcal{S}$ have two forms. The first, `<p>` $* W$, contains a program location `<p>` that immediately precedes a non-conditional statement hole $*W$. The second, `<p> Cond`$(*b, \mathcal{P}_1, \mathcal{P}_2)$, is a program location `<p>` that immediately precedes a conditional `Cond`$(*b, \mathcal{P}_1, \mathcal{P}_2)$ where $*b$ is a Boolean hole and program templates $\mathcal{P}_1, \mathcal{P}_2$ represent the true and false branches of that conditional, respectively.

We observe that the template structure of a program or of any loop body in that program in the mathematical procedural examples that we consider usually consists of very small number of statement holes, and the number of such templates is also relatively small. For example, the template for the loop of GCF: Euclid's Algorithm in Figure 1 is `Cond`$(*b, \texttt{Sequence}(*W, *W), \texttt{Sequence}(*W, *W))$. In GCF: Successive Divison it is `Sequence`$(*W, *W, *W, *W)$, and in GCF: Simultaneous Division the inner loop is `Sequence`$(*W)$ and the outer loop is `Sequence`$(*W, *W)$. The algorithm thus restricts its search to programs that fit a set of templates that is provided as input. As enumeration of possible templates is straightforward, this set of templates can be iteratively and automatically increased until the synthesis algorithm succeeds.

A Co-Template, Val, of a given template $\mathcal{P}$ is a mapping from holes in $\mathcal{P}$ to a set of angelic statements. We define the function $\mathcal{P}[Val]$, which returns an angelic program $\tilde{P}$ obtained by replacing each `Hole` $hole$ in $\mathcal{P}$ by $Val(hole)$. Statement holes $*W$ are mapped to angelic non-conditional statements $\tilde{W}$, and Boolean holes $*b$ are mapped to angelic Boolean expressions $\tilde{b}$. We use the notation $Val[*b \leftarrow \tilde{b}]$ to indicate when a Boolean hole $*b$ in a Co-template is filled with an angelic Boolean expression $\tilde{b}$. Similarly, $Val[*W \leftarrow \tilde{W}]$ indicates that the statement hole $*W$ is filled with an angelic non-conditional statement $\tilde{W}$.

```
SynthesizeFromExample(Trace Tr, Maximum
  loop depth k:int, Templates Ps)
1 Let ProgA be a three-dimensional array of R̃.
2 Let ProgA[v,i,j] contain all programs with loop depth
  ≤ v that can generate trace Tr from node i to node j.
3 Initialize every entry of ProgA[v,i,j] to ⊥.
4 for i ← 1 to Length(Tr):
5   ProgA[0,i−1,i] := {Convert(Update(k̃1,k̃2,Ẽ,w),
    State(Tr,i)) s.t. Update(k̃1,k̃2,Ẽ,w) computes the i^th
    element in Tr from previous entries in T}.
6 for v ← 1 to k:
7   foreach P ∈ Ps:
8     for n ← 0 to Length(Tr) − 1:
9       AddLoopPrograms(ProgA, v, n, P, Tr);
10 return ProgA[k];
```

Figure 6. Procedure `SynthesizeFromExample`.

4.3 Dynamic Programming

Now, for each subsequence of an example trace, we learn the set of all loops with bounded depth, represented as angelic structures, that fit one of the given templates. The procedure `SynthesizeFromExample`, defined in Figure 6, performs this task using a dynamic programming based approach. This procedure takes as input a `Trace Tr`, a maximum desired loop depth k, and a set of loop-free templates $\mathcal{P}s$, and outputs a two-dimensional array $Prog$ such that $Prog[n_1, n_2]$, for $0 \leq n_1 \leq n_2 \leq$ `Length(Tr)`, contains the set of all loop programs of depth $\leq k$ that are consistent with the subsequence of `Tr` from index n_1 to index n_2. This array represents a directed acyclic graph (`DAG`) in which nodes are timestamps and edges are programs that explain the changes to the spreadsheet between two nodes. Internally, `SynthesizeFromExample` uses a three-dimensional array, $ProgA$, to represent this `DAG`. The added third dimension is used to store the loops of a particular loop depth in the learning process. All entries of $ProgA$ are initialized to $\perp$ on line 3.

For each step in the trace, `SynthesizeFromExample` tries to explain the value written to the spreadsheet on that step, by computing the set of operations that produce that value from the input and values generated in previous steps (lines 4-5). It does this with the `Convert` function, which takes a program structure and a state σ as input and it attaches that state to all the constants that occur in that program, in order to convert those integer constants into angelic integer constants. `State` calculates the state σ of the trace `Tr` at a particular timestamp. `SynthesizeFromExample` then stores each set of operations in $ProgA$. The operations learned in this initial phase can only compute one step ahead; therefore, after this phase the only edges in $ProgA$ that are not $\perp$ are edges of the form $ProgA[0, i, i+1]$, $0 \leq i <$ `Length(Tr)`.

We learn loops on lines 6-9 by iterating over the set of templates and learning the set of all programs that fit each template. In each synthesis pass, we call a separate procedure `AddLoopPrograms` in order to find the set of loops that fit a particular template and that start at a particular timestamp. Within a single loop-learning pass, for each template in the set of templates, we call `AddLoopPrograms` for all possible start times m_i, $1 \leq i \leq$ `Length(Tr)`. We then learn a second set of loops by iterating again over the set of templates, including loops learned in the previous step as statements in this outer-loop learning process.

```
AddLoopPrograms(DAG ProgA, current loop depth v:int,
  start time n:int, Template P, Trace Tr)
1  Let j be a fresh variable.
2  Let p_0 and p_end denote the start and end of P.
3  Let *b_0 be a fresh boolean expression hole.
4  Let Val_0 map every hole in P to ⊥.
5  b̃_0 := Extend(Convert(true, State(Tr, n)), j, 0);
6  Worklist := {(Val_0[*b_0 ← b̃_0], 0, p_0, n)}; Processed := ∅;
7  while Worklist ≠ ∅:
8    Pick and remove (Val, z, p, m) from Worklist;
9    Switch (Successor(p)):
10     case (*W, p'):
11       foreach m' where ProgA[v − 1, m, m'] ≠ ∅:
12         W̃_1 := Extend(ProgA[v − 1, m, m'], j, z);
13         W̃_1 := Intersect(Val(*W), W̃_1);
14         if W̃_1 ≠ ⊤ then:
15           add (Val[*W ← W̃_1], z, p', m') to Worklist;
16     case (*b, p_1, p_2):
17       b̃_1 := Extend(Convert(true, State(Tr, m)), j, z);
18       b̃_1 := Intersect(Val(*b), b̃_1);
19       if b̃_1 ≠ ⊤ then:
20         add (Val[*b ← b̃_1], z, p_1, m) to Worklist;
21       b̃_2 := Extend(Convert(false, State(Tr, m)), j, z);
22       b̃_2 := Intersect(Val(*b), b̃_2);
23       if b̃_2 ≠ ⊤ then:
24         add (Val[*b ← b̃_2], z, p_2, m) to Worklist;
25     case (p = p_end):
26       b̃_1 := Extend(Convert(true, State(Tr, m)), j, z + 1);
27       b̃_1 := Intersect(Val(*b_0), b̃_1);
28       if b̃_1 ≠ ⊤ ∧ m < Length(T) then:
29         add (Val[*b_0 ← b̃_1], z + 1, p_0, m) to Worklist;
30       b̃_2 := Extend(Convert(false, State(Tr, m)), j, z + 1);
31       b̃_2 := Intersect(Val(*b_0), b̃_2);
32       if b̃_2 ≠ ⊤ then:
33         add (Val[*b_0 ← b̃_2], z, p, m) to Processed;
34 foreach (Val, z, p, m) ∈ Processed:
35   Prog[v, n, m] := Prog[v, n, m] ∪ {Loop(j, Val(*b_0), P[Val])};
```

Figure 7. Procedure `AddLoopPrograms`.

4.3.1 AddLoopPrograms

`AddLoopPrograms`, shown in Figure 7, takes as input an integer n that indicates the timestamp from which we are trying to learn loops, a template $\mathcal{P}$, and a `Trace Tr`. The goal is to compute the set of programs that match $\mathcal{P}$ and explain the trace starting at time n. `AddLoopPrograms` does this by stepping through $\mathcal{P}$, continually filling the statement and Boolean holes in the template with angelic statements that represent the programs that match that hole. `AddLoopPrograms` maintains a worklist of tuples (Val, z, p, m) that contain a Co-template Val, the value of the loop iteration variable z, a program location variable p that indicates the current position in the template, and a timestamp m representing the current location in `Trace Tr`. Each time the main loop (line 7) iterates, `AddLoopPrograms` calls a successor function `Successor` on line 9 that returns the hole in the template that immediately follows p. `Successor` can return a non-conditional statement hole $*W$ (line 10), a Boolean hole $*b$ (line 16), or signal that it has reached the end of the template (line 25).

If `Successor` returns a non-conditional statement hole $*W$ (line 10), we consider all previously computed statements in the `DAG` $ProgA$ that could fill that hole (lines 11-15). We therefore consider every m' for which $ProgA[v, m, m'] \neq \bot$ and hypothesize that this statement might be the next statement in the loop. On line 12, we use the `Extend` function to update the angelic integer constants within that loop to reflect that this statement is being called with the loop iterator variable set to the value of that iteration. This function takes as input an angelic program $\tilde{P}$, a fresh loop iterator j that does not occur in $\tilde{P}$, a non-negative integer z, and returns another angelic program where the state σ in each angelic constant occuring in $\tilde{P}$ is extended with the assignment $j := z$. We then intersect this program with anything stored for previous invocations of $*W$ and add it to the worklist (lines 13-15).

If `Successor` returns a Boolean hole $*b$ (line 16), then we add two hypotheses to the worklist: one in which execution goes into the true branch (lines 17-20), and one in which execution goes into the false branch (21-14). On lines 17 and 21, we use the `Extend` and `Convert` functions to create new angelic Boolean expressions $\tilde{b}$ that represent both possible hypotheses for the behavior of the Boolean expression represented by this hole: true and false. After intersecting this angelic Boolean expression with previous invocations of this expression (lines 18 and 22), we add these hypotheses to the worklist (lines 29 and 33). Hypothesizing that conditionals can branch either way leads to a combinatorial explosion in the number of entries in the worklist; however, since `AddLoopPrograms` uses the `Intersect` function to ensure that there actually is a solution to the set of linear constraints over loop iterator variables that we maintain in the angelic integer constants $\tilde{k}$, many infeasible loop hypotheses quickly die out.

If `Successor` signals that we reached the end of the template (line 25), then we maintain two hypotheses: that the loop continues (lines 26-29), and that it ends now (lines 30-33). To hypothesize that the loop continues, we record that the loop continuation Boolean expression evaluated to true (lines 26-27) and add a tuple to the worklist in which the template location is returned to the initial program location of the template (line 29). To hypothesize that the loop ends, we record that the loop continuation Boolean expression evaluated to false (lines 30-31) and add the filled co-template to the $Processed$ list to be added to the `DAG` (line 33).

```
Synthesize(Examples {Z_1, ·, Z_m}, Maximum loop depth
  k:int, Templates Ps)
1  for (i := 0 to m):
2    Prog_i := SynthesizeFromExample(Z_i.Trace, k, Ps);
3  temp_2 := ∅;
4  foreach (P in Ps):
5    Let Val_0 map every hole in P to ⊥;
6    result := {Val_0};
7    for(i := 0 to m):
8      newResult := ∅; temp := Unify(Prog_i, P, Z_i);
9      foreach (Val' in temp):
10       foreach (Val in result):
11         fail := false;
12         foreach (Hole hole in P):
13           Val''[hole] := Intersect(Val'[hole], Val[hole]);
14           if(Val''[hole] = ⊤):
15             fail := true; break;
16         if (not fail):
17           newResult := newResult ∪ Val'';
18     result = newResult;
19   foreach (Val in result):
20     P := P[Val]; temp_2 := temp_2 ∪ {P};
21 Materialize all integer and Boolean angelic constants
   in temp_2.
22 return temp_2;
```

Figure 8. Procedure `Synthesize`.

4.4 Overall synthesis algorithm

Figure 8 defines the overall synthesis algorithm, `Synthesize`, which takes as input a set of examples $\{Z_1, \cdot, Z_m\}$, a maximum loop depth k, and a set of templates $\mathcal{P}s$, and returns a (non-angelic) program P. The key idea is to compute the set of angelic programs for each example and then intersect all of these programs. However, we cannot just naïvely intersect them because we need to

Procedure	L	C	S	Templates	T(s)	ST(s)
Addition: Count On	1	0	2	[S]	<1	6
Addition: Standard	1	1	6	[C{2S}{S}],[S]	21	fail
Division: Repeated Subtraction	1	0	3	[2S],[S]	2	32
Div.: Repeated Subt. Remainder	1	0	2	[S]	<1	4
Fraction Multiplication	0	0	2	[2S]	<1	1
Fraction Division	0	0	4	[4S]	<1	25
Fraction Reduction	1	0	5	[4S],[S]	8	fail
Fraction Reciprocal	0	0	2	[2S]	<1	1
GCF: Euclid's Algorithm	1	1	7	[C{2S}{2S}],[S]	7	fail
GCF: Simultaneous Division	2	0	5	[S],[2S]	4	fail
GCF: Successive Division	1	0	5	[4S],[S]	14	fail
Matrix Addition	2	0	3	[S]	112	fail
Matrix Subtraction	2	0	3	[S]	52	fail
Matrix Scalar Multiplication	2	0	3	[S]	6	fail
Pattern Continuation: Addition	1	0	2	[S]	<1	3
Pattern Continuation: Subtraction	1	0	2	[S]	<1	3
Pattern Contin.: Explicit Add.	1	0	2	[S]	<1	3
Pattern Contin.: Explicit Subt.	1	0	2	[S]	<1	3
Prime Factorization	1	0	3	[2S],[S]	1	fail
Subtraction: Count Back	1	0	2	[S]	<1	4

Figure 9. Summary of target algorithm benchmarks. L, C, S, show the number of loops, conditionals, and statements, respectively, for each intended procedure. The next column shows the templates used to construct the target procedure. We abbreviate templates here; for example, 2S is a template with two statements and C{2S}{2S} is a conditional with two statements in each branch. T shows the number of seconds taken by our algorithm to generate a program solving all of the provided demonstrations. ST reports the number of seconds taken by SKETCH to synthesize the program *when given the exact supertemplate* (see Section §7). "fail" indicates that no program was synthesized within 10 minutes.

Bug	P	L	C	S	Templates	T(s)	ST(s)
Addition	34	1	0	2	[S]	<1	1
Addition	35	1	1	6	[C{2S}{S}],[S]	15	fail
Addition	36	1	1	6	[S],[C{2S}{S}]	<1	fail
Addition	37	1	1	8	[C{2S}{2S}],[2S]	<1	fail
Subt.	38	1	1	5	[C{S}{S}],[S]	<1	fail
Subt.	39	1	0	4	[S],[3S]	<1	fail
Subt.	40	1	1	5	[C{S}{S}],[S]	7	fail
Subt.	41	1	2	10	[C{C{S}{S},2S}{S}],[S]	24	fail
Subt.	42	1	1	7	[C{S}{S}],[3S]	<1	fail
Mult.	44	1	1	6	[C{2S}{S}],[S]	<1	fail
Mult.	45	0	0	4	[4S]	<1	fail
Division	47	1	1	5	[C{S}{S}],[S]	<1	62
Frac. Red.	51	0	0	2	[2S]	<1	1
Frac. Red.	52	1	3	11	[C{S}{C{S}{C{S}{S}}}],[S]	4	32
Frac. Red.	53	0	1	6	[C{2S}{2S}]	<1	4
Frac. Add.	54	0	0	2	[2S]	<1	1
Frac. Add.	55	1	0	5	[2S],[3S]	<1	fail
Frac. Add.	56	1	0	5	[2S],[3S]	<1	49
Frac. Add.	57	1	0	5	[2S],[3S]	<1	fail
Frac. Subt.	58	0	3	14	[C{S}{S},C{2S}{C{2S}{2S}}]	<1	fail
Frac. Subt.	59	1	1	12	[3S],[S,C{3S}{S},2S]	127	fail
Frac. Subt.	60	1	0	3	[S],[2S]	<1	2
Frac. Subt.	61	2	1	10	[S],[3S],[S,C{S}{S},S]	41	fail
Frac. Mult.	63	1	0	6	[S],[5S]	<1	fail
Frac. Mult.	64	1	0	2	[S]	<1	1
Frac. Div.	65	1	0	2	[S]	<1	1
Frac. Div.	66	1	0	6	[S],[5S]	<1	2
Dec. Add.	67	0	1	5	[C{2S}{S}]	<1	2

Figure 10. Summary of "buggy" benchmarks from Ashlock [2]. P shows the page number on which each bug appeared. See Figure 9 for description of other columns. These results show that our algorithm can efficiently learn programs to describe students' errors.

learn Boolean conditionals. To do this we make use of our templates. The initial loop on lines 1-2 computes the `DAG` for each example. Then, for each template, we initialize a set of co-templates $result$ to a single co-template in which all holes are mapped to $\perp$ (lines 4-6). We iterate through each trace and compute the set of all programs that fit that template and are consistent with that trace. We do this with the `Unify` procedure on line 8, which is not included in the paper but follows the same basic process found in `AddLoopPrograms`: step through the templates and return the set of matches, a set of co-templates. The algorithm then intersects these co-templates with everything we have found so far in $result$ (line 14) to see if anything is in common. If so, this co-template survives to the next iteration (lines 16-18). The result is the union of all intersections for each template (lines 19-20). The last step is to materialize the integer and Boolean constants (line 21).

The following theorem holds:

THEOREM 1. *Our algorithm synthesizes all programs in our language for the set of templates $\mathcal{P}$s consistent with provided examples $\{Z_1, \cdot\cdot, Z_m\}$, integer operators `F`, and Boolean operators `G`.*

4.5 Optimizations

We apply a set of heuristics to cut down on the number of loops. We delete a loop if we have computed another loop that starts or ends at the same time but iterates longer. We delete conditionals in which both branches are the same. We delete a co-template if it is a subset of another co-template. These heuristics can delete correct programs; however, they greatly improved efficiency.

5. Evaluation

We evaluated the effectiveness of our data structure and synthesis algorithm by testing it on correct demonstrations of 20 K-12 math procedures and 28 buggy demonstrations of some of those procedures. For each benchmark, we selected a set of operators and predicates related to that algorithm. These operators represent concepts that are taught in previous chapters and are expected to be applied as a single step. Many of these operators were simple, like addition, subtraction, and less than. Some were more complex, such as finding the lowest prime divisor of a set of numbers.

We picked a set of templates for learning inner and outer loops by studying the examples and trying to come up with a representative set. The inner loop templates were all templates with no more than 5 statements, 3 conditionals, and 3 statements per conditional. The conditionals could be nested. The outer loop templates were `Sequence`$(*W_1)$ and `Sequence`$(*W_1, *W_2)$. The templates provided to the `Unify` procedure were a set of 164 templates, representing all possible templates with a maximum of 8 statements, 1 conditional and 3 statements per conditional branch. Instead of trying all templates at once, we used an iterative, phased strategy that tried various subsets of templates until synthesis succeeded.

5.1 Correct programs

For each correct procedure, we collected a set of problems from a variety of textbooks and tried to pick a set that explored the full range of pathways through the procedure. For each procedure, we first provided a single example to the synthesizer. If the synthesized program solved all of the example problems, we stopped. If there were examples that the learned solution procedure did not solve correctly, we added the first such incorrect problem and tried again. We continued this process until the synthesized program was able to solve all of the programs correctly.

Our results are listed in Figure 9. For each problem, we report a few metrics: the number of loops, conditionals, statements, and the set of loop templates needed for the synthesized program. Note that the same set of templates were used for all of our benchmarks; this column reports the templates needed to construct the program

Figure 2 in [7]. Example input table:

	Qual 1	Qual 2	Qual 3
Andrew	01.02.03	27.06.08	06.04.07
Ben	31.08.01		05.07.04
Carl		18.04.03	09.12.09

Example output table:

Andrew	Qual 1	01.02.03
Andrew	Qual 2	27.06.08
Andrew	Qual 3	06.04.07
Ben	Qual 1	31.08.01
Ben	Qual 2	
Ben	Qual 3	05.07.04
Carl	Qual 1	
Carl	Qual 2	18.04.03
Carl	Qual 3	09.12.09

Figure 8 in [7]. Example input table:

Name	Color	Price
Toyota	Red	2000
Nissan	White	4000

Example output table:

Toyota	Red
Toyota	2000
Nissan	White
Nissan	4000

Figure 9 in [7]. Example input table:

3099	905	A4CA				
NO.14	NO.14	Full Copies	6.78	2	* *	0
3200	906	AHG				
9-Jun	9-Jun	Covers Only	4.74	1	* *	0

Example output table:

3099	905	A4CA	NO.14	Full Copies	6.78	2
3200	906	AHG	9-Jun	Covers Only	4.74	1

Figure 11. Our system learned a program to compute these three spreadsheet table transformations from Harris and Gulwani [7].

we intended to find, not the set of templates that were tried. This set of examples shows considerable variety in terms of loop and conditional structures. We report the time it took to synthesize the procedure that solves 100% of the practice problems.

5.2 Buggy procedures

Ashlock [2] identifies a set of 40 buggy computational patterns for a variety of algorithms. We focused on a large subset of these algorithms: addition, subtraction, multiplication, division, fraction reduction, fraction addition, fraction subtraction, fraction multiplication, and fraction division. Our bug results are listed in Figure 10. Our system is able to synthesize programs consistent with all of the examples provided for 28 of the 40 bugs in the book (excluding those in the appendix), which is about 70% coverage.

The bugs that we were not able to capture fell into three categories. First, some of the bugs involved base operators that were quite unrelated to the operators used in the correct algorithm. For example, one of the bugs for multiplying two fractions f_1 and f_2 required a base operator defined as $f_1.num * f_2.denom + (10 * f_1.denom + f_2.num)$. Although our system could capture such a bug if provided such a non-standard operator, this was too impractical to include as a successful benchmark. Second, some algorithms, particularly those for division, involved traces that were too long and complicated for our system to handle. Better heuristics for pruning operators and template search strategies would likely help us capture such bugs. Third, some bugs involved word problems.

6. Other Applications

Although our system was designed to learn K-12 mathematical procedures, we believe it can advance the state-of-the-art for programming by demonstration in other domains as well. One such domain is layout transformations on spreadsheet tables [6, 7]. In this domain, the input is a 2D table of entries with type *string*. The output is another table containing a rearrangement of the cells in the input. Figure 11 shows three motivating transformations from [7] that from an online help forum for Excel macro programming.

We used our synthesizer to learn a program for each of these table transformations. Since our algorithm requires a full step-by-step demonstration, we simulated entering information into the spreadsheet in a way that seemed natural. We provided a single operator, "move". For each of the three transformations, we provided the demonstration shown in Figure 11, and another similar example of a different size. In all three cases, the synthesizer took about 10 seconds to learn the correct program. These examples show how our general approach can apply to domains other than math. In contrast, the technique presented in [7] is specialized for table layout transformations and cannot synthesize any of our math programs.

7. Comparison to SKETCH

We compared our tool to SKETCH [24], a state-of-the-art general-purpose program synthesis tool that is the closest existing system to our work. SKETCH takes as input an incomplete program with first-order holes (integers, Booleans) and tries to fill in these holes in a way that satisfies all assertions for all possible inputs. We encoded our demonstration-based specifications by asserting "if the input is X, then the output is Y." We used version 1.6.4, released on May 15th, 2013.

Harris and Gulwani [7] reported that SKETCH could not successfully solve their table transformation benchmarks. Therefore we tried SKETCH on all our math benchmarks, as shown in Figures 9 and 10. For each benchmark, we report how long SKETCH took to synthesize a program or "fail" if no program was synthesized within 10 minutes. We found that SKETCH could synthesize some of the simpler benchmarks. However, it failed to efficiently solve larger programs with more complex control structures because it cannot process the large number of required holes. We gave SKETCH the full conditional and loop structure for each benchmark, with all other statements and Boolean expressions replaced by holes. We refer to this as a *supertemplate*. We note that the timings reported for SKETCH are very optimistic because in reality one would have to try out all possible supertemplates of which there is a very large number. We conclude that, since SKETCH is designed to synthesize programs when most of the program is known, it is not so well suited for our benchmarks in which the entire program is essentially unknown. This makes our template-based dynamic programming approach necessary.

8. Related Work

We review related work in two areas: program synthesis and educational technology.

8.1 Program Synthesis

Recent approaches to program synthesis have focused on two areas: version space algebras and template-based synthesis techniques. Neither approach can synthesize most of our benchmarks, as existing version space algebras are not designed to learn nested loops and conditionals, and existing template-based approaches cannot scale. Our framework draws inspiration from both these areas: it borrows the idea of manintaining multiple hypotheses from the former, and templates from the latter. It combines these general ideas in a non-trivial manner using a novel dynamic programming algorithm and angelic inference of atomic expressions.

Version Space Algebras Version space algebra based techniques have been used to synthesize programs from trace demonstrations or input-output examples. The key idea is to efficiently compute and succinctly represent a large number of hypothesis in an underlying domain-specific language. The original concept was pioneered by Mitchell [19] for refinement-based learning of Boolean functions. It was later extended by Lau et.al. [13] to learn simple loops in the SMARTedit system, which learns text editing commands with base operations such as moving the cursor to a new position or inserting and deleting text. Recently, Gulwani extended

these ideas to learn functions with simple loops in a more sophisticated Programming by Example setting [3]. Version space algebras have been applied to a wide variety of application domains including text manipulation [13], string manipulation [18], table transformations [9], repetitive robot programs [20], shell scripts [14], and Python programs [15].

These techniques cannot solve our math benchmarks because they are specialized for individual domains. More significantly, they are not sufficiently robust to learn programs with complex loop and conditional structures. Prior techniques can only handle simple loops, without nested loops or conditionals inside loops. They also require the user to explicitly indicate each iteration of a loop inside a demonstration. These kind of restrictions make such systems less useful and usable [12]. Our approach does not have these restrictions. It can learn nested loops, thanks to the novel dynamic programming based algorithm that iteratively learns loops of increasing depth and uses angelic expressions to deal with yet unknown loop iterators. Furthermore, the use of templates facilitates learning of conditionals, even inside loops. Our technique is general purpose and is parameterized by a set of base operators.

Template-based Program Synthesis Since program synthesis is a hard combinatorial problem, many program synthesis techniques require the user to specify the control-flow structure of a program with templates [24–26]. These methods use a variety of techniques for the underlying combinatorial search, such as brute-force search with A*-style heuristics [5], SAT solvers [24], SMT solvers [4, 11], and probabilistic inference. Most of these techniques are inapplicable to our setting since they are either specialized to specific domains such as loop-free programs [4], geometry constructions [5], progam inverses [26], or require complete functional specifications [25]. The most closely related work to our setting is that of SKETCH [24], which is a general purpose template-based program synthesizer and can accept various forms of specifications. SKETCH, which is based on reducing the synthesis problem to solving SAT/SMT constaints, works well in an interactive setting where programs are mostly complete (with insight from the programmer) and have a small number of holes. SKETCH is not well suited for our domain becuse it does not fully utilize specifications expressed as demonstrations and fails to synthesize most of our benchmarks. In contrast, our template-based technique can fully take advantage of user demonstrations, and it scales better because it leverages insights from version space algebras and uses a novel dynamic programming algorithm.

8.2 Educational Technology

Training of procedural tasks has been realized in intelligent tutoring systems [10, 21] primarily through manual crafting of production rules. These production rules check if a particular condition is true about the problem state and perform an operation. This process is time-intensive, motivating automatic approaches. Recently, Li et al. showed how a machine learning agent, SimStudent, can learn production rules for a target educational procedure from input data [16]. In our work, we extend the expressive capability of synthesized procedures to full imperative programs. Recently, program synthesis was applied to automatically generate algebra proof problems [22] and solutions to geometry construction problems [5], but these approaches only work on restricted domains.

9. Conclusion

We have presented a novel programming language and synthesis framework that uses programming by demonstration to learn K-12 mathematical procedures, both correct and incorrect. Our framework can learn complex structures such as loops and conditionals by defining a set of template loop skeletons and learning sets of programs that match each of these templates. We successfully used our system to synthesize programs from demonstrations of 20 correct procedures and 28 "buggy" versions of 9 procedures.

References

[1] E. Andersen, S. Gulwani, and Z. Popović. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI '13*.

[2] R. Ashlock. *Error Patterns in Computation: A Semi-Programmed Approach*. Merrill Publishing Company, 1986.

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

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

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

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

[7] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[8] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite. ASE '10, pages 407–416, 2010.

[9] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[10] K. Koedinger and A. Corbett. Cognitive tutors: Technology bringing learning science to the classroom. *The Cambridge handbook of the learning sciences*, 2006.

[11] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

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

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

[14] T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI*, 2004.

[15] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43, 2003.

[16] N. Li, W. Cohen, K. Koedinger, and N. Matsuda. A machine learning approach for automatic student model discovery. In *EDM*, 2011.

[17] N. Matsuda, A. Lee, W. W. Cohen, and K. R. Koedinger. A computational model of how learner errors arise from weak prior knowledge. *Annual Conference of the Cognitive Science Society*, 2009.

[18] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2001.

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

[20] M. Pardowitz, B. Glaser, and R. Dillmann. Learning repetitive robot programs from demonstrations using version space algebra. In *RA '07*.

[21] K. Schulze, J. Shapiro, R. Shelby, D. Treacy, and M. Wintersgill. The andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15:147–204, 2005.

[22] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[23] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.

[24] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

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

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

[27] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.

[28] K. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. MIT Press, Cambridge, MA, USA, 1991.