Automatic Scaffolding for Procedural Learning

Erik Andersen

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Zoran Popović, Chair

Sumit Gulwani, Chair

Anna Karlin

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Automatic Scaffolding for Procedural Learning

Erik Andersen

Co-Chairs of the Supervisory Committee:
Professor Zoran Popović
Computer Science and Engineering

Affiliate Professor Sumit Gulwani
Computer Science and Engineering

A key challenge in education is how provide support that is tailored to the learner's individual needs. Effective teachers and curricula typically provide such support, often referred to as *instructional scaffolding*, through the development of progressions of practice problems, step-by-step demonstrations, and strategies for diagnosing misconceptions. This process is often tedious and time-consuming. Furthermore, it typically requires a large amount of design by experts and little can be reused across educational domains. As a result, creating adaptive educational content often remains prohibitively difficult.

This thesis presents a general framework for constructing instructional scaffolding for procedural learning, a key domain of learning in which the goal is to learn a step-by-step procedure. In contrast to previous approaches that require a large amount of domain-specific authoring, this framework takes as input only the procedure to be learned and produces scaffolding automatically. Directly encoding procedural knowledge in this way allows us to leverage techniques from the software engineering community. The framework uses test input generation tools to synthesize systematic progressions of practice problems that start easy, grow more difficult, adapt to the learner, and ultimately cover all of the important pathways through the procedure. When provided with a mapping from programming language elements to visual objects in a user interface, the framework can also automatically generate step-by-step demonstrations for any practice problem associated with a given pro-

cedure. Finally, the framework can synthesize many K-12 mathematical algorithms by demonstration, enabling it to learn "buggy" algorithms from student data. More generally, this thesis explores how the natural hierarchies that exist within cognitive models of target concepts can suggest pragmatic and automatic ways to teach them.

TABLE OF CONTENTS

Page

LIST OF FIGURES

ACKNOWLEDGMENTS

I am particularly grateful to my advisors, Zoran and Sumit, for their ideas, dedication, energy, and support. I would also like to thank the collaborators, developers, and artists who helped me with this thesis and other projects: Yun-En Liu, Eric Butler, Eleanor O'Rourke, Adam M. Smith, Seth Cooper, François Boucher-Genesse, Kathleen Tuite, Marianne Lee, Brian Britigan, Happy Dong, Yanko Yankov, Aaron Whiting, Tim Pavlik, David Yamanoha, Stephen Sievers, Roy Szeto, Mai Dang, Christian Lee, Ethan Apter, Emma Lynch, Justin Irwen, Carmen Petrick Smith, and Taylor Martin.

DEDICATION

to Ryoko

Chapter 1

INTRODUCTION

Education is one of the most important things we do as a society. It is one of our best ways to alleviate poverty and improve the future of each child. However, despite the efforts of so many passionate educators, teaching every topic to every child in an effective and engaging way remains extraordinarily challenging. A key difficulty is that each child has a unique set of skills, preferences, and interests. Children improve at different speeds and require varying levels of reinforcement. As they improve, they continually require material that is at the right level for them. Children can become confused in many ways, and each misconception must be addressed individually.

Humans learn most effectively when they receive support that is directly tailored to their individual needs. This support is often referred to as *instructional scaffolding* [70]. There are many forms of scaffolding, such as instructional resources, practice problems, problem progressions, step-by-step walkthroughs, and strategies for the diagnosis and correction of misconceptions.

Scaffolding is necessary because of the idiosyncrasies of the human learning process. Humans have limited short-term memory, need to be continually engaged, and must repeatedly practice skills in order to internalize them. With these constraints in mind, researchers in education and psychology have suggested several guiding principles for the design of effective scaffolding. For example, Csikszentmihalyi's theory of flow [17] suggests that educational content can keep the learner in a state of maximal engagement by continually increasing difficulty to match the learner's increasing skill. Vygotsky's zone of proximal development [88] stipulates that there is a set of concepts that a learner can acquire next with some guidance, and educational content should continually target this set. Reigeluth and Stein's Elaboration Theory [68] argues that the simplest version of a task should be taught first, followed by progressively more complex tasks that elaborate on the original task.

The creation of high-quality scaffolding is challenging and time-consuming. It is typically done by hand, by experts, and requires a great deal of knowledge of particular educational subdomains. Although there is a rich body of educational theory to draw from, designers often apply these principles in an ad hoc manner when designing curricula. We still lack pragmatic and automatic ways to bridge the gap between educational theory and practice, especially at the scale of an entire curriculum.

The difficulty of designing scaffolding also limits the ability of educational curricula to adapt to the needs of each student. Although human teachers frequently help individual students by answering questions, providing feedback on homework, and giving additional help or practice problems, the majority of textbooks and instructional materials that are currently in use cannot easily respond to students who are faster or slower than expected. By definition, adaptive curricula are more complex than static curricula. One of the key challenges is that the space of possible understandings and misconceptions is massive. Since creating static educational content is already hard, designing a curriculum that can deal effectively with every possible combination of knowledge and misconception is even harder.

Recent changes in the educational ecosystem have made it even more challenging and important to build scaffolding on a large scale. Massively Open Online Courses (MOOCS) have recently become popular, spawning multiple companies such as Coursera[1] and Udacity[2]. Khan Academy[3] provides instructional videos, for free, on a wide range of subjects. Games for learning such as *Refraction* (Center for Game Science 2010) and *DragonBox* (We Want to Know 2012) attempt to make learning more fun and engaging. In all of these domains, since there is minimal interaction between a teacher and a learner, *all* feedback must be automated.

There have been many attempts to use technology to provide and enhance instructional scaffolding, including systems that can automatically adapt to the student. Intelligent tutoring systems such as Cognitive Tutors [16, 40] have shown considerable success, even approaching a level of effectiveness comparable to a human tutor [86]. However, they are

[1] http://www.coursera.com

[2] http://www.udacity.com

[3] https://www.khanacademy.org/

also very expensive to design; Cognitive Tutors require as many as 200-300 hours of expert design effort to create one hour of consumable content [2]. As a result, the startup cost to build a tutor for a new domain remains prohibitively large.

In order to increase the ability of educators to design effective instructional content for new domains, and to create adaptive curricula that can respond to the needs of a student in a profound way, I believe that we need to increase the role that automation can play in this process. We need a framework that can automatically reason about the space of possible scaffolding for a set of concepts. Ideally, the input to such a system would be a set of educational objectives and the output would be scaffolding for those objectives in the form of practice problems, demonstrations, help, etc.

In order for a system to produce scaffolding automatically, educational objectives must be presented in a format that the system can understand and utilize to produce educational content. The natural hierarchies that exist within cognitive models of target concepts can suggest pragmatic, automatic, and general ways to teach them. By deconstructing such models into smaller building blocks, and focusing on how to assemble and visualize these blocks, we can create theoretically-grounded instructional scaffolding. This contrasts with most previous approaches to automatically creating educational content, which typically require designers to write low-level domain-specific algorithms and are not very general.

1.1 Approach

One of the most important domains of human learning is procedural task learning, which spans a wide range of human activities. Humans learn to execute procedures that range from a simple list of actions, such as a cooking recipe, to more complex procedures involving loops and conditionals, such as prime factorization, long division, and solving systems of equations. What is common to all of these domains is that there is an algorithm that we want the student to learn. This algorithm provides a natural cognitive model to work with.

Thesis Statement: *By encoding any procedural thought process as a computer program, we can leverage techniques from software engineering and programming languages to automatically generate layered instructional scaffolding for teaching that algorithm. In particular, we can use (i) test input generation tools to create progressions of practice problems, (ii)*

debuggers to create step-by-step demonstrations, and (iii) programming-by-demonstration to model misconceptions. Although some effort is required to encode the thought process as a program, much less effort is required to create the scaffolding itself.

In Chapter 3, I will describe a framework for reasoning about the space of possible progressions of practice problems as defined by a procedural task. Solving such problems allows a learner to develop an internal model of the procedural algorithm over time, so that ultimately it can be applied correctly to all possible inputs for that procedure. I propose categorizing a procedural task based on features of the program trace obtained by executing the procedure on that task. I will show how this trace-based measure can be used to measure the quality of a progression and compare the relative difficulty of two problems. I will also show how we can use test input generation tools to create practice problems that exercise a desired procedural trace, perform a breadth-first exploration of practice problems and their corresponding traces, and assemble the discovered problems into systematic progressions. I will demonstrate the effectiveness of this framework for generating practice problems for many procedures in K-12 math and show how it scales to the level of producing an entire progression of hundreds of levels for a commercially-available learning game for algebra, replicating hundreds of hours of design effort in a single automatic process.

In Chapter 4, I will describe a framework for creating step-by-step demonstrations of a procedural task. I will show how linking programming language elements to a visual user interface can enable a debugger to step through a procedure line-by-line and demonstrate each line. I will show how consideration of procedural traces can minimize the amount of explanation that is presented to the learner by automatically omitting discussion of decisions that are not relevant for the current problem. I will present a prototype system that implements these ideas for 2D-table-style problems in K-12 math.

In Chapter 5, I will propose a method for diagnosis of incorrect understandings of a procedure, or "buggy" algorithms. This programming-by-demonstration framework attempts to synthesize all of the algorithms reproducing a student's behavior that can be expressed in a fairly general-purpose programming language that we define. I will show that this system can learn 70% of the "bugs" in a well-known collection of students' errors [9]. Furthermore, this system can also learn the correct (intended) algorithms, enabling teachers to

create programs without knowing how to program. Our dynamic programming approach advances the state-of-the-art of program synthesis because previous work cannot efficiently synthesize complex control flow structures like nested loops and conditionals inside loops, which are needed to capture some misconceptions.

Chapter 2

RELATED WORK

This chapter presents related work. First, I will include a discussion of intelligent tutoring systems, educational technology, and educational video games in Sections 2.1 and 2.2. Then, I will present work related to the three primary investigations of this thesis: progressions (Section 2.3), demonstrations (Section 2.4), and misconceptions (Section 2.5). Finally, I will present related work on programming-by-demonstration in Section 2.6.

2.1 Intelligent Tutoring Systems

There is a long history of using technology in education. Some of the most advanced systems are intelligent tutoring systems (ITS), which are computer programs that provide customized instruction and feedback for each student. They have been shown to improve test scores by as much as one standard deviation [87, 40].

Some of the most well-known intelligent tutoring systems are the Cognitive Tutors [8]. These systems use a cognitive model of the student to provide tailored instruction. One of the major theoretical ideas behind Cognitive Tutors is knowledge tracing [16], which models knowledge as a set of production rules of the form "if the goal is X, and the student is in state Y, then the student should do Z." They then use a Hidden Markov Model (HMM) in which the hidden states are whether or not the student knows each production rule. Cognitive Tutors keep giving problems until this HMM has achieved 95% confidence that the student has learned every production rule.

Cognitive Tutors are difficult to construct. It has been estimated that as much as 300 hours of expert design effort are required to design a single hour of content [2]. This is because the entire system must typically be programmed by hand - every production rule, every practice problem, every explanation, every action a student might take, and every response to that action. Therefore, although these systems are known to be effective, they

have not reached their full potential in part because they are so hard to build.

ITS creators are aware of the challenges in creating such systems, and they have tried to make this easier. A survey of this effort can be found in [61]. For example, the Cognitive Tutoring Authoring Tools [2] provide a series of widgets that allow non-programmers to input all of the relevant information described above, such as what the problem state should look like on the screen, what to say if the student makes a mistake, etc. They facilitate the creation of example-tracing tutors [3]. In such tutors, a designer creates a *behavior graph* in which nodes represent problem states and edges encode both correct and incorrect answers that a student can give at a particular problem state. When the tutor is run, and the student makes one of the indicated mistakes, the tutor displays the message associated with that link. This strategy still requires the designer to specify the entire behavior graph.

There is some work on learning production rules from data. For example, Li et al. showed how a machine learning agent, SimStudent, can learn production rules for a target educational procedure from input data [48]. However, this technique has not been used to generate tutors themselves.

My work builds on these approaches by using models of procedural knowledge to generate many kinds of instructional scaffolding automatically. While previous work has certainly made it easier for experts to create effective educational technology, including adaptive systems, the scalability of these systems remains poor. Experts must still predict and program a large space of possible interactions. In contrast, I argue that it is more efficient to explicitly model the procedure knowledge we want to teach, and use these models to create robust systems that can respond to a wide variety of possible interactions (for example, misconceptions) without having to predict all of them ahead of time.

2.2 Educational Video Games

Although much of educational technology research has focused on more traditional settings, such as intelligent tutoring systems, recently some of this attention has shifted to video games [72, 32]. Video games are particularly exciting because they have the potential to be highly engaging. However, many of the design challenges that make it hard to scaffold learning in intelligent tutors also make it difficult to build good educational games. For

example, in the case of *Refraction* (Center for Game Science 2010), a level designer spent over 100 hours creating the level progression of the game, which contains 61 levels. Changing a game mechanic often requires substantial changes to this progression.

Most research on educational game design has focused on specification of guidelines for designing good games. Linehan et al. [49] provide a series of guidelines for designing effective educational games. Isbister et al. [34] report insights gained from interviewing game designers. In previous work, I found that video game tutorials are only effective when the game is too complex for players to learn through experimentation [7] and that changing a progression by introducing secondary game objectives can negatively impact engagement and retention [5, 6].

However, there has been much less technical work on engineering games so that they are both educational and engaging and reducing the amount of design effort that is required. This is a primary goal of this thesis. In particular, Chapter 3 describes a practice problem generation framework that can generate video game levels in addition to more traditional practice problems. In future work, I hope to apply the techniques described in Chapters 4 and 5 to games as well. In fact, an early prototype of the step-by-step demonstrations framework has been implemented for *Refraction*.

2.3 Practice Problem Progressions

The primary way that people learn how to execute procedures is by solving progressions of practice problems. Since these are tedious to write by hand, there is a long line of work in generating them automatically. Some of the earliest approaches involve grammars. For example, McArthur et al. [56] automatically generated algebra problems as part of an intelligent tutor for algebra. They first collected and annotated problems by what skills they use, such as isolating positive terms and multiplying both sides of an equation by -1. They then indicated relationships between these skills, such as that one is a prerequisite for another, or that one is a generalization of another. Since each of these skills is essentially a production rule, it was straightforward to specify a grammar for how these rules can transform a problem state into another problem state. They used random values for constants to generate problems. Sleeman [75] used a similar approach of specifying a grammar for gen-

erating algebra problems. This approach works for algebra, and can control some aspects of difficulty, but it does not work for all domains. Furthermore, it requires the designer to write algorithms that are specific to a particular domain, and very little can be reused for a different procedural skill.

Another approach is to specify a template of what the problem should look like, and use generation techniques to fill in the holes in the template. For example, Jurkovic [36] created templates for algebra problems and then used random integers for constant terms and coefficients. Unfortunately, this method makes it very hard to control difficulty. For many domains in K-12 math, the choice of constants makes a huge difference in the difficulty level of a problem. For example, 1000 minus 1 is a very different problem from 1001 minus 1 because the first problem involves a complex kind of borrowing.

Further work in template-based approaches has allowed for holes that are not just constants, but also include features of the solution process. For instance, in the domain of quadratic equations, some interesting features could be whether the equation is "simple factorable", "difficult factorable, where the leading coefficient is not 1", or "requires use of general quadratic formula". Another interesting feature is whether or not it has imaginary solutions. The Microsoft Math-Worksheet Generator automatically infers such features from a problem instance [57]. However, each domain has its own set of features that needs to be defined separately. In contrast, I argue that a procedural description of the problem domain leads to a more natural and automatic definition of such features. Furthermore, it allows for more complicated features, such as n-grams, that relate to a sequence of decisions required to solve a problem as opposed to a single decision.

Singh et al. [74] proposed a semi-automatic template-based approach to problem generation for the domain of algebra proof problems. The teacher semi-automatically generalizes a given seed problem into an abstract template. However, not all instantiations of those templates are valid proof problems. The underlying system performs a brute-force enumeration over all possible instantiations and uses novel results from randomized algebraic identity testing [30] to filter those instantiations that yield valid problems. They can generate impressive and non-trivial problems; however, there is no guarantee of the difficulty level associated with each problem.

Cerny et. al. [13] also proposed a template-based approach to problem generation for the domain of automata theory problems. A given seed problem is abstracted into a template, and most instantiations, if not all, are actually correct problems. The system then uses a sophisticated solution generation technology based on novel results in automata theory to compute the solutions for various instantiations and then partitions the problems into different equivalence classes based on some user-defined similarity metric. However, no attempt is made to compare the problems across different equivalence classes or different seed problems.

All of the work discussed so far is focused on generating individual problems. However, most practice problem sets involve a progression of problems that gets more difficult over time. There is little public information on how, exactly, Cognitive Tutors decide which problem to give next. The SQL-tutor [60] selects problems with constraints that students have violated before or have not practiced to give next.

Li et al. studied problem orderings by examining them with a machine learning agent called SimStudent. They found that interleaved problem orderings led to faster learning than blocked orderings [47]. I extend this work by creating a framework in which interleaved or blocked problem progressions that get more difficult over time can be created for any procedural task.

Most previous techniques for problem generation involve writing some kind of infrastructure that is specific to a particular problem type or procedural skill. In contrast, my work tries to generate problems using as little domain-specific authoring as possible. Chapter 3 will introduce a framework that uses test-input generation tools to automatically create practice problems through static analysis of procedural knowledge encoded as a computer program. While there is some effort required to write this program, I believe that this is a more natural and scalable approach because an educational technologist only has to write it once, and all important practice problems can be generated directly. Furthermore, Chapter 3 will present ideas for organizing problems into progressions in ways that systematically introduce new components of a procedural skill and clearly grow more difficult over time, by specifying partial orderings that apply generally across procedural domains and are not specific to any domain.

2.3.1 Procedural Content Generation in Video Games

Although practice problems and progressions are important to education, they have also been studied from the perspective of video games. Many video games contain challenges of increasing difficulty that are intended to develop a particular skill. Therefore, there have been some interesting investigations into automatically generating video game levels. For example, Smith et al. used answer-set programming to generate levels for *Refraction* that adhere to constraints written in first-order logic [77]. Similar approaches have also been used to generate levels for platform games [79]. Butler et al. [12] designed a mixed-initiative tool that combines a level generator with an interface that the designer can use to inspect a progression and modify it at a high level.

In the majority of existing approaches, designers must explicitly specify constraints that the generated content must reflect, for example, "the tree needs to be near the rock and the river needs to be near the tree". There has been much less work in guiding the generation based on a description of what the player needs to do. Dormans [21] generated levels for puzzle-platform games through grammars that could build a "mission" for the desired tasks the player needed to perform. I expand on these methodologies by generating not only levels but also level progressions directly from a procedure that solves those levels. Although previous work has typically considered level solvers and level generators to be separate entities, my framework unifies these two ideas, which I believe is a particularly exciting direction.

2.4 Demonstrations

Providing guides and tools that allow students to complete problems that would otherwise be beyond their reach can improve learning outcomes [35, 23, 91]. Therefore, most intelligent tutoring systems provide demonstrations that teach the student how to solve a problem. These are typically written by hand for each problem.

Learnability is a central component of software usability [26, 62, 71] and using complex software often requires mastery of procedural tasks. Therefore, researchers in Human Computer Interaction (HCI) have explored procedural learning in this context and developed

scaffolding to support it. Rieman [69] examined how humans learn to perform procedural tasks through exploratory learning. Dong et al. developed Jigsaw, a game for teaching Adobe Photoshop that allows users to discover how to use the software functionality through a series of puzzles [20].

Much of this work has focused on step-by-step demonstrations. Kelleher and Pausch designed Stencils, an interactive tutorial for the Alice computer programming language that restricts user input to help new users follow specific procedures [38]. Bergman et al. designed DocWizards, a system for creating and following step-by-step instructions by exposing the steps in the procedure and highlighting relevant interface elements [10]. Gupta et al. present a real-time system that tracks and guides the assembly of snap-together block models using a Kinect camera [31]. This system can be used to both author and follow step-by-step tutorials for how to build certain block models.

There is also work that focuses on authoring video tutorials. Pongnumkul et al. developed Pause-and-Play, a system that links events in the video to the user's corresponding events in the target application and pauses the tutorial when the user lags behind [65]. Chi et al. developed DemoCut, a semi-automated system for editing and cutting videos authored by amateurs to improve their quality [15].

While step-by-step demonstrations are valuable instructional tools, authoring this type of tutorial content can be very time consuming [10, 25, 67]. As a result, a number of researchers have explored methods of improving the tutorial authoring process. Some have focused on improving the reusability of existing tutorials. Ramesh et al. developed ShowMeHow, a system that automatically translates step-by-step tutorials between similar applications such as Photoshop and GIMP [67]. Others have explored methods of using programming-by-demonstration to generate programs that are used to create tutorials automatically. Grabler et al. developed a system for automatically generating text and image tutorials from demonstrations in an instrumented version of GIMP [25]. Chi et al. developed MixT, a system for automatically generating step-by-step tutorials with both textual and video content from user demonstrations in Photoshop [14]. While these systems can reduce the time to author tutorials, they provide no support for re-using tutorial content across problems or application domains. Furthermore, both systems only support linear procedures.

The most closely related work is DocWizards [10], which automatically generates interactive tutorials by learning the underlying procedure from multiple demonstrations and supports more complex procedures that include conditional statements. The DocWizards interface exposes a human-readable version of the learned procedure that the user can step through line-by-line. At each step, the current line is highlighted and the widget involved in the next action is circled in the user interface. However, DocWizards was not designed to explain the problem-solving thought process; the learner is expected to read and understand $if - else$ statements, which significantly reduces the usability of this system. Furthermore, the procedure must be demonstrated for many distinct problems before a general underlying program can be learned, and the complexity of the procedures that can be represented is limited.

Chapter 4 introduces a framework that automatically generates step-by-demonstrations directly from analysis of a procedural skill expressed as a computer program. It supports complex non-linear procedures and can produce tutorials for every practice problem used to teach a given procedure. This is useful because there are many practice problems; for example, there are over 55 million unique four-digit subtraction problems, which have 73 unique procedural traces (explained in Chapter 3). It would be a massive amount of work to write templates for the 73 different types of problems. Furthermore, this system facilitates reuse of infrastructure; a small amount of domain-specific authoring allows creation of a tutorial system that can explain a large number of two-dimensional spreadsheet procedures in K-12 math.

2.5 Misconceptions

Many researchers have explored the errors that students make while learning procedural skills. Ashlock [9] identified bugs for multiple algorithms such as addition, subtraction, multiplication, division, and operations with fractions. VanLehn [84] extensively studied subtraction and thoroughly analyzed the bugs that children display while learning this procedure. He also built programs that could identify if a student displayed a certain error from a list of errors. This work led to the development of Repair Theory, which tries to explain why students make certain errors.

There is some work associated with learning models of student knowledge in intelligent tutoring systems, both for model-tracing tutors [11] and Cognitive Tutors [52, 53, 48]. These systems learn production rules by demonstration. The authors first specify a set of possible operators and predicates, and the system then observes a human solve each problem and tries to learn production rules that describe its behavior. Such an approach can synthesize most of the production rules for algebra. Matsuda et al. [54] used a production-rule-learning framework to learn errors; however, they found that this system was unable to learn and predict student errors effectively because the system learned production rules that were too general or too specific.

In Chapter 5, I build on this work by focusing on a larger domain of procedural tasks, two-dimensional spreadsheet problems in K-12 mathematics. I seek to build a system that can identify and classify a very large space of errors that a student could possibly make. In contrast to previous work that learns production rules, my goal is to create a system that can synthesize full algorithms, with complex control flow structures like nested loops and conditionals. Accomplishing this requires a designer to supply a set of base operators and language structures that is sufficiently large to capture all bugs. However, this requires less effort than programming every single error by hand.

2.6 Programming-by-demonstration

In Chapter 5, I will present a programming-by-demonstration system capable of learning complex programs from demonstrations of their behavior, for both the "correct" knowledge that we want to teach and also "buggy" understandings of that knowledge. This system builds on a large volume of work in programming-by-demonstration. As I will discuss, previous techniques cannot efficiently handle programs complex control flow such as nested loops and conditionals within loops, and this is a key way in which my method improves over previous work.

Recent approaches to program synthesis have focused on two areas: version space algebras and template-based synthesis techniques. Neither approach can synthesize some algorithms in K-12 math because existing version space algebras are not designed to learn nested loops and conditionals, and existing template-based approaches cannot scale. My

framework draws inspiration from both these areas: it borrows the idea of maintaining 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.

2.6.1 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 [59] for refinement-based learning of Boolean functions. It was later extended by Lau et.al. [44] 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 [27]. Version space algebras have been applied to a wide variety of application domains including text manipulation [44], string manipulation [58], table transformations [37], repetitive robot programs [64], shell scripts [43], and Python programs [45].

These techniques cannot scale to learn K-12 spreadsheet math procedures 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 [42]. My 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. My technique is general purpose and is parameterized by a set of base operators.

2.6.2 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 [80, 82, 81]. These methods use a variety of techniques for the underlying combinatorial search, such as brute-force search with A*-style heuristics [30], SAT solvers [80], SMT solvers [29, 41], and probabilistic inference. Most of these techniques are inapplicable to K-12 math since they are either specialized to specific domains such as loop-free programs [29], geometry constructions [30], progam inverses [81], or require complete functional specifications [82]. The most closely related work is that of SKETCH [80], 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 K-12 math because it does not fully utilize specifications expressed as demonstrations. Furthermore, I tried to use SKETCH to learn both correct and incorrect programs and it could not do so efficiently. In contrast, my 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.

Chapter 3

PRACTICE PROBLEM PROGRESSIONS

A fundamental challenge of teaching procedural tasks is determining the optimal sequence of training problems. Textbooks for elementary and middle school mathematics typically start with problems that only require a few steps to solve and grow to more complex multi-step problems that vary based on the input. These progressions vary widely and many of them are likely suboptimal. The quality of a training sequence depends on many factors, such as the structure of the target procedure, cognitive processes that lead to the creation of procedural models in the mind, level of engagement towards the task, learner background, and learning preferences.

There are a number of guiding principles for practice problem progressions. Reigeluth and Stein's Elaboration Theory [68] argues that the simplest version of a task should be taught first, followed by progressively more complex tasks that elaborate on the original task. Csikszentmihalyi's theory of flow [17] suggests that we can keep the learner in a state of maximal engagement by continually increasing difficulty to match the learner's increasing skill. By considering Vygotsky's zone of proximal development [88], we can avoid overloading the learner by introducing so many concepts at the same time that the learner cannot create a consistent internal representation. Nevertheless, many important details of optimal progression design are not covered by general principles.

In this chapter, I present a framework that I developed with Sumit Gulwani and Zoran Popović for reasoning about the space of possible progressions as defined by the procedural task itself. Our goal is to create a representation of the space of progressions using only a specification of the algorithmic procedure, defined directly as a computer program. We propose categorizing a procedural task based on features of the program trace obtained by executing the procedure on that task. We show how this trace-based measure can be used

---

Much of the work described in this chapter was originally presented as a paper [4] at CHI 2013.

to measure the quality of a progression and compare the relative difficulty of two problems.

The use of a trace-based framework for characterizing procedural tasks allows us to borrow well-established techniques from the software engineering community related to software testing. In particular, it allows us to use test input generation tools [83] for generating problems that have certain trace features. It also allows us to use notions of procedure coverage [89] to evaluate the comprehensiveness of a certain progression. Furthermore, we are able to borrow techniques from the sequence comparison literature to compare two different problems, in particular n-gram models [90].

We demonstrate our method by analyzing the space of progressions in three different domains. One domain is early math procedures, such as addition, adding fractions, and comparing integers. The second domain is puzzle games, in which we apply our framework to *Dragonbox* and *Refraction*, two well-known interactive math puzzle games. The third domain is phonology, in this case the Thai alphabet. We will show how we can generate large sets of practice problems to build mastery of procedural knowledge in each of these domains. We also show how our framework can be used to analyze and compare expert-designed progressions in terms of thoroughness and aggressiveness. Determining which kinds of progressions are more effective is beyond the scope of this thesis. Our goal is to automatically discover the coverage achieved by a progression, and generate additional problems to supplement the areas that it covers sparsely.

To assess our partial ordering of problem difficulty, we conducted a pilot user study in which we generated several levels for an algebra-learning game and asked participants to compare these levels to those in the expert-designed progression. We found that our model was better able to predict participant responses than other baseline metrics.

This chapter is organized as follows. Sections 3.1, 3.2, 3.3 introduce the formalism of the framework. Section 3.4 describes how we used our framework to analyze math problems. Section 3.5 provides an overview of our problem synthesis techniques. Then, Sections 3.6, 3.7, 3.8 and 3.9 describe how we used our framework to generate practice problem progressions for math, *Dragonbox*, the Thai alphabet, and *Refraction*, respectively.

Algorithm 1 Addition: Given as input a sequence of sequences of digits $n_i = [n_{i_q}, ..., n_{i_0}]$, add them:

```
1:  procedure ADD(n_1, ..., n_m)
2:      a ← 0
3:      maxLen ← max(len(n_1), ..., len(n_m))
4:      for i ← 0, maxLen − 1 do                          ▷ Loop over digits (D)
5:          k ← 0
6:          for all n_j do                                ▷ Loop over inputs (N)
7:              if len(n_j) > i then
8:                  k ← k + n_j[i]                        ▷ If digit exists (A)
9:              end if
10:         end for
11:         if c[i] ≠ null then
12:             k ← k + c[i]                              ▷ If carry exists (C)
13:         end if
14:         if len(k) = 2 then
15:             c[i + 1] ← k[1]                           ▷ If we need to carry (O)
16:         end if
17:         a[i] ← k[0]
18:     end for
19:     if c[maxLen] = 1 then
20:         a[maxLen] ← c[maxLen]                         ▷ If final carry (F)
21:     end if
22:     return a
23: end procedure
```

3.1 Procedural Traces

This section is the first of three that will present the formalism for characterizing practice problems and estimating their difficulty. To begin, consider the standard addition algorithm, shown in Algorithm 1. We believe that one of the best categorizations of the difficulty of a particular addition problem is the pathway or the trace that the procedure takes while solving that problem. A quick analysis of addition problems in any textbook will likely show many such traces. We will use sequences of letters to indicate traces; these letters are output when the program executes commented lines in the above procedure.

Valid inputs to this problem have two or more input numbers and these numbers can have any number of digits. The simplest possible trace under these constraints is the trace corresponding to a one-digit number plus a one-digit number where the sum is less than 10:

"DNANA". In this trace, the carry branch "C" does not execute because there is no existing carry, the "O" overflow branch does not execute because there is no overflow, and there is no final carry so the final carry branch "F" also does not execute. The digit existence branch must execute because otherwise the addend would be invalid.

The following table shows several problems and their traces:

Problem	**Trace**
$1+1$	DNANA
$11+1$	DNANADNAN
$1+11$	DNANADNNA
$11+11$	DNANADNANA
$9+1$	DNANAOF
$19+1$	DNANAODNANC
$1+2+3$	DNANANA
$1+2+3+7$	DNANANANAOF
$333+444$	DNANADNANADNANA

Characterizing problems by their execution traces has several implications. It allows us to measure the program coverage of a problem, which can be used to evaluate the comprehensiveness of a progression. It also allows us to compare problems, a notion more formally captured in the next section, based on how the procedure executes. It does *not* take into account differences in perceived difficulty that may arise from differences in the input values. For example, students may be much more comfortable with adding $1+1$ than $5+4$, and my model will not take this into account. However, such differences are generally domain-specific and cannot be identified without gathering data. Our model can easily grow to accommodate such distinctions. If a teacher or designer wants to distinguish two classes of inputs, he or she can add a conditional branch to the program that will separate these two classes.

3.2 Partial Orderings Over Traces

In order to assemble practice problems into progressions that begin simply and grow more difficult, we define a methodology for comparing two different problems. To do this, we define partial orderings over problems based on features of their corresponding traces. We can use this partial order relationship to compare two problems and determine if they are similar in difficulty, if one problem is more difficult than the other, or if no determination can be made using our model. Once we have such a partial ordering, we can identify simpler or more complex variations of a particular problem, and we can use it to sequence multiple problems together into a progression.

There are many possible ways to define a partial ordering over execution traces. Here, we define two partial orderings that we have found useful in analyzing and synthesizing progressions.

3.2.1 Path-based Partial Ordering

We use the following trace features to define our first partial order over traces:

- the number of times a certain loop in the procedure was executed
- whether or not the non-skip branch of a certain exceptional conditional was executed; we define a conditional to be *exceptional* if one of its branches contains no instructions and is equivalent to "skip".

We now use the following recursive definition to define a partial ordering:

Definition 1a: *A trace T_1 that contains loops is at least as complex as trace T_2 if for every loop L in the procedure, the trace T_1 has at least as many iterations of loop L as in trace T_2, and for each subtrace T_2' in T_2 that corresponds to an iteration of loop L, there exists a subtrace T_1' in T_1 that is at least as complex as T_2'.*

We also define the following base case:

Definition 1b: *A non-loopy trace T_1 is at least as complex as trace T_2 if for every exceptional condition in the procedure, if the non-skip branch of the exceptional condition was executed in T_2, then it was also executed in T_1.*

We now give some examples of trace comparisons based on the above-defined partial order.

Example 1: *Loops that execute longer are more complex.* If we compare the traces "DNANA" and "DNANANA", corresponding to the problems $1 + 2$ and $1 + 2 + 3$, respectively, the only difference is that the inner "N" loop executed one more time. Therefore, "DNANANA" subsumes "DNANA".

Example 2: *A conditional branch that executes some statements is more complex than a conditional branch that executes no statements.* If we compare "DNANA" and "DNANAOF", corresponding to the problems $1 + 2$ and $9 + 1$, respectively, "DNANAOF" contains all of the "work" of "DNANA", plus the additional step that there is an overflow and the 1 must be carried over to the next digit. Therefore, "DNANAOF" also subsumes "DNANA".

Example 3: Since loops can contain conditionals and other loops, each iteration of a loop that executes multiple times may vary in complexity. Our partial order also allows us to compare such traces. For example, consider the two traces "DNANADNANA" and "DNANADNAN", corresponding to the problems $11 + 11$ and $11 + 1$, respectively. In both cases, the outer "D" loop executes twice. The first iteration of this loop is the same but the second iteration is different. The second iteration for "DNANADNANA" executes a final "A" statement in an exceptional condition that "DNANADNAN" skips. Therefore "DNANADNANA" subsumes "DNANADNAN".

The notion of partial order also allows us to take a given trace and build more complex variants. For example, consider again the trace "DNANA" from the above addition example. There are three obvious ways in which this trace can expand. The first is by flipping conditional branches that executed no statements so that they do execute statements. One such trace is "DNANAOF", corresponding to problems in which there is a carry such as $1 + 9$. Another possible expansion is to increase the number of iterations of the "N" loop by 1, yielding the trace "DNANANA" that corresponds to problems with three one-digit numbers and no carries like $1+2+3$. Another way to expand the problem is to increase the "D" loop by one iteration, perhaps yielding the trace "DNANADNANA" corresponding to two-digit addition problems with no carries like $12 + 34$.

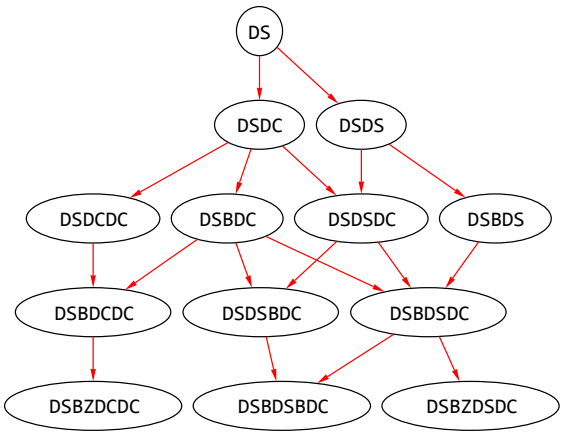


Figure 3.1: A partial ordering of problem traces for the subtraction algorithm in Algorithm 2 based on 1-grams. This is shown with the additional constraint that each 1-gram in P_1 must appear fewer or equal times in P_2 for P_1 to be considered easier than P_2.

3.2.2 N-gram-based Partial Ordering

We now define a second partial ordering by utilizing n-grams [90]. This method defines a family of partial orderings depending on the value of positive integer n. n-gram models have been used extensively in natural language processing and search engines, where the meaning of a word can be uniquely defined by just looking at a small context around it in the parent sentence. In the case of procedural execution traces, the intuition is that students can only remember a certain amount of context, and it is most useful to test students on their ability to execute small sequences of algorithmic decisions within a larger execution trace.

The n-gram abstraction of a trace also provides a nice continuum between the standard notions of *statement coverage*, which corresponds to a uni-gram model, and *path coverage*, which corresponds to ∞-gram model in the software engineering literature [89].

Definition 2: *Let n be any positive integer. We say that a trace T_1 is at least as complex as trace T_2 if every n-gram of trace T_2 is also present in trace T_1.*

3.3 Constructing Full Progressions

The previous sections introduced the idea of using traces to classify practice problems and specified two different partial orderings that use traces to rank the difficulty of problems.

Algorithm 2 Subtraction: Given as input a minuend p and subtrahend q, both sequences of digits $n_m, ..., n_0$:

```
1:  procedure SUBTRACT(p, q)
2:      for i := 0 to len(p) − 1 do                          ▷ Each digit (D)
3:          if i < len(q) then                               ▷ Subt. digit present (S)
4:              if q[i] > p[i] then                          ▷ Must borrow (B)
5:                  p[i] := p[i] + 10
6:                  j := 0
7:                  while p[i + j] = 0 do                    ▷ Zero (Z)
8:                      p[i + j] := 9
9:                      j := j + 1
10:                 end while
11:                 p[i + j] := p[i + j] − 1
12:             end if
13:             a[i] := q[i] − p[i]
14:         else                                             ▷ Copy down (C)
15:             a[i] := q[i]
16:         end if
17:     end for
18: end procedure
```

In this section, we extend these ideas into an end-to-end system that can generate a full progression, allowing control of the pacing and providing a platform for adaptation.

Consider the algorithm for subtraction detailed in Algorithm 2. Given a problem and its trace, the *n-grams* of a trace are the n-length substrings of the trace. For example, in the trace $ABABCABAB$, the 1-grams are $\{A, B, C\}$, the 2-grams are $\{AB, BA, BC, CA\}$, and the 3-grams are $\{ABA, BAB, ABC, BCA, CAB\}$. Intuitively, 1-grams represent fundamental procedural concepts, and higher-value n-grams represent the interactions between concepts.

3.3.1 Queries over partial ordering

We can use n-grams to create a partial ordering graph, as shown in Figure 3.1. Once we have such a graph, we can define useful queries over that graph.

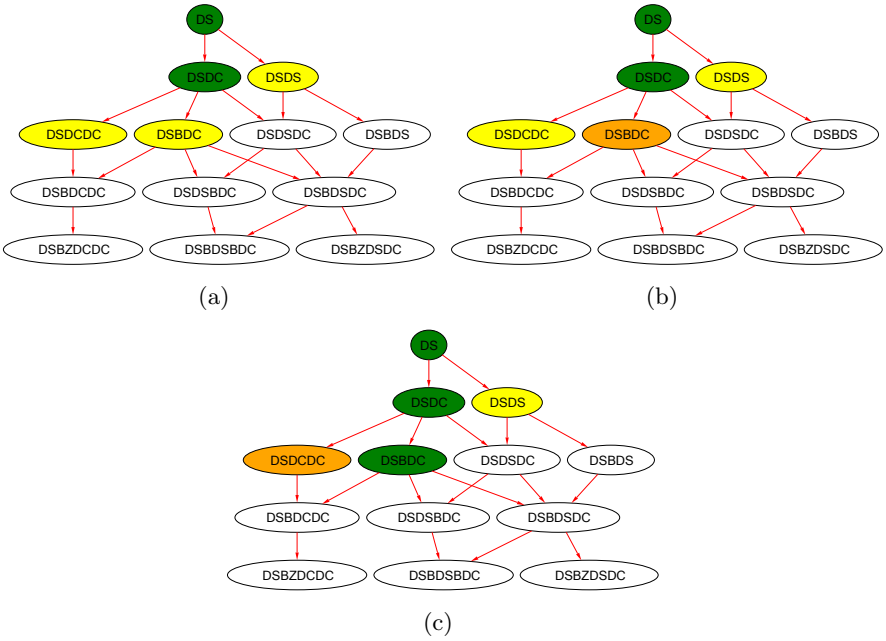


Figure 3.2: If the player has completed the two green nodes in (a), then the three yellow nodes are good candidates for the next problem. We can then arbitrarily pick one to give next. (b) shows the selected node in orange. Once this node is completed, we recalculate the fringe, and select another node arbitrarily, as shown in (c).

Make problems harder or easier

We can do this by simply looking at the ancestors and descendants of nodes in the graph.

Identify good problems to give next

This depends on what problems we have seen the player successfully solve in the past. One way to do this is by tracking success by coloring nodes in the partial-ordering graph. We mark nodes as either unknown or known, then define the set of good nodes to give next as the set of nodes such that all predecessors of that node have been completed. Figure 3.2 illustrates an example.

Breaking a problem into subparts

The partial ordering graph also provides suggestions for how we can provide remediation if the player fails a task. If the player fails a problem with a particular trace, and we have no further information to indicate how the user failed, one possible strategy is to mark the current problem as "unknown" and reevaluate all of the predecessors of that node. If the player also fails one of those predecessors, this strategy can continue until the problematic node is found.

Simplest trace that preserves some substructure

If we have information about where in the trace the player failed, we can utilize this information to find a problem that removes as much complexity unrelated to the error as possible. For example, suppose we know with which n-gram the player had trouble. We can search over problems whose traces contain that n-gram for the one with the least additional complexity by finding the minimum cost, as determined by our weighted-sum cost function over n-grams.

3.4 Analysis Of Progressions

In order to test the ability of our framework to analyze progressions, we gathered problems from three different workbooks for elementary and middle school mathematics:

- Math Sprints series from Singapore Math Inc.. These books include many worksheets that are intended to be completed in rapid one-minute sessions. They tend to involve lots of repetition and not too much difficulty.

- JUMP Math curriculum. This Canadian curriculum was able to raise at-grade-level performance from 12% to 60% in a study in Lambeth, England [1].

- Skill Sharpeners: Math by Evan-Moor Educational Publishers.

`http://www.singaporemath.com/Math_Sprints_s/184.htm`

`http://jumpmath1.org/`

`http://www.evan-moor.com/Product.aspx?SeriesID=122`

Algorithm 3 Fraction Computation: Given as input a set of fractions $f_1, ..., f_n$ and a set of + and − operations $o_1, ..., o_{n-1}$, execute these operations:

```
1:  procedure FCompute(f_1, ..., f_n, o_1, ..., o_{n-1})
2:      if ∃f_i, f_j : f_i.d ≠ f_j.d then
3:          lcm ← max(d ∈ f_i.d)                          ▷ Denom. diff. (D)
4:          while ∃f_i : lcm mod f_i.d ≠ 0 do
5:              lcm ← lcm + max(d ∈ f_i.d)                ▷ (M)
6:          end while
7:          for all f_i do
8:              s ← lcm ÷ f_i.d
9:              f_i.n ← f_i.n * s
10:             f_i.d ← f_i.d * s
11:         end for
12:     end if
13:     n ← 0
14:     for all o_i do                                    ▷ For each operation (O)
15:         if o_i = + then
16:             n ← n + f_{i+1}.n                         ▷ Add (A)
17:         else if o_i = − then
18:             n ← n − f_{i+1}.n                         ▷ Subtract (S)
19:         end if
20:     end for
21: end procedure
```

3.4.1 Evaluating Progressions

We can use our framework to explore interesting patterns in a progression. Figure 3.3 shows progressions from the Singapore Math Sprints workbooks for the the addition algorithm (Algorithm 1) and an algorithm for performing multiple addition and subtraction operations on fractions, shown in Algorithm 3. These worksheets are organized into pairs of "A" and "B" worksheets. The instructions state that the "A" progression is intended for weaker students and the "B" progression is intended for stronger students. For most of these pairs, the two worksheets cover the same set of concepts. However, we can see that the "A" progressions spend more time moving back and forth between simple traces and the "B" progressions move into more complicated traces that subsume the early traces.

Figure 3.4 shows a larger-scale analysis of progressions from all of these books. For each progression that we chose to analyze in each book, we first categorized all of the problems

Figure 3.3: We can use our trace-based framework to compare progressions. The Singapore Math Sprints books organize worksheets into pairs of "A" worksheets and "B" worksheets, stating that the B side is "intended for more advanced students". The above figures compare the "A" and "B" progressions for two different problem domains. In both figures, the green solid arrows correspond to the "A" progression, and the blue dashed arrows correspond to the "B" progression. Self-edges are removed for clarity. The left side shows a worksheet pair for addition, corresponding to Algorithm 1. The right figure shows a different worksheet pair for fraction computation, corresponding to Algorithm 3. In both cases, we see that the "advanced" progression spends less time in the easier regions and quickly moves into longer pathways that require additional steps. In the case of addition, the "B" problems have more addends, and in the case of fraction computation, the "B" problems have more input fractions.

by trace. We can see that for most of these problem types, the number of traces is much less than the number of problems. However, there are differences between progressions; for example, the Singapore Sprints progression for addition has many more unique traces than the Skill Sharpeners progression.

3.4.2 Making progressions more systematic

We can use our framework to identify pathways a progression does not cover and suggest problems to fill that gap, if desired. Algorithm 4 defines an algorithm for comparing two integers. Figure 3.5 compares the Skill Sharpeners and JUMP Math progressions for Algorithm 4. Since there are too many nodes in the JUMP Math progression to visualize here, only the first third of the progression is shown. We can see that there is a considerable difference in how these progressions proceed. The JUMP Math progression, indicated in green, goes more quickly into more complex traces. The Skill Sharpeners progression spends more

Problem	Book	Problems	Unique Traces	% generated
Addition	Skill Sharpeners: Math	584	17	88%
	Singapore Math Sprints	576	48	64%
Fraction Computation	Skill Sharpeners: Math	66	13	100%
	Singapore Math Sprints	248	15	100%
	JUMP Math	131	11	95%
Fraction Reduction	Skill Sharpeners: Math	66	13	100%
	JUMP Math	56	11	100%
Integer Comparison	Skill Sharpeners: Math	24	6	100%
	JUMP Math	88	12	100%

Figure 3.4: Summary of textbook progression analysis. We analyzed progressions from three textbooks for four different problem domains. We partitioned problems into groups based on their execution trace. The final column shows how many of these traces we were able to generate using Pex, a test input generation tool.

Algorithm 4 Integer Comparison: Given as input two sequences of digits $a = [a_0, ..., a_m]$ and $b = [b_0, ..., b_n]$, determine if $a > b$, $a < b$, or $a = b$:

```
1:  procedure COMPARE(a, b)
2:      if len(a) > len(b) then
3:          return more                          ▷ More digits (H)
4:      else if len(a) < len(b) then
5:          return less                          ▷ Fewer digits (L)
6:      end if
7:      for i ← 0, len(a) − 1 do                 ▷ For each digit (D)
8:          if a_i > b_i then
9:              return more                      ▷ Digit is larger (G)
10:         else if a_i < b_i then
11:             return less                      ▷ Digit is smaller (S)
12:         end if
13:     end for
14:     return equal                             ▷ Equal (E)
15: end procedure
```

time going back and forth between simple problems, and never reaches some of the longer traces. We can also see that the first third of the JUMP Math progression omits a class of problems indicated by the traces "H" and "L". The JUMP progression eventually includes a problem in the "L" class but not until late in the progression. These traces correspond to problems in which a number is greater than another number because the number of digits

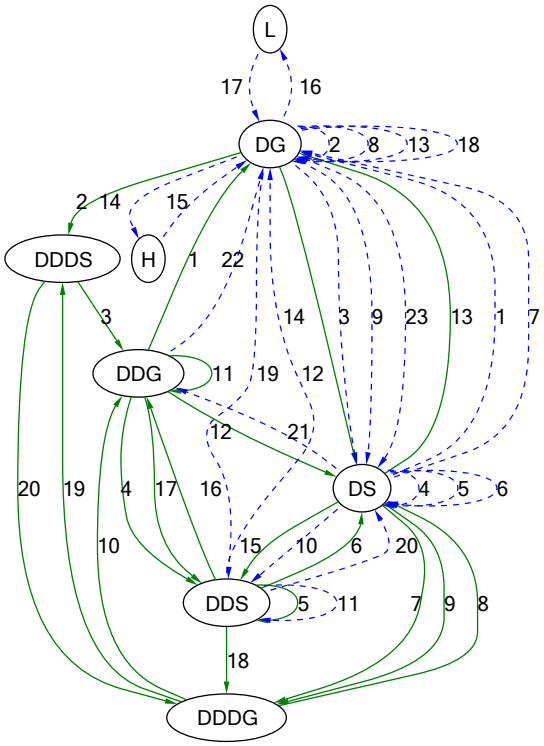


Figure 3.5: Filling in holes in progressions. This figure compares two progressions for integer comparison (Algorithm 4). “H” corresponds to the trace that gets executed when the first number has more digits than the other and is therefore greater. “L” corresponds to the trace that gets executed when the first number has fewer digits. “D” signifies that the number of digits is the same and that the execution path moves down the digits from left to right. “G” signifies that the digit under examination was smaller and “S” means that the digit under examination was bigger. Therefore, “DG” means that the first number is greater than the second because the first digit was greater. The blue dashed arrows indicate the Skill Sharpeners: Math progression and the green solid arrows indicate the JUMP Math progression. The JUMP Math progression quickly reaches more difficult problems than the Skill Sharpeners progression. The Skill Sharpeners progression never reaches traces longer than “DDG” and “DDS”. It also omits problems in which the first number is greater or less than the second number because the number of digits is different. We can “repair” such holes by borrowing problems from other progressions or generating them automatically.

Problem	Book	n-gm.	Cov.	Missing Traces
Frac. Comp.	JUMP	1	100%	
		2	100%	
		3	93%	SOA
		4	65%	MOAO, OSOA, MOSO, DOSO, SOSO, AOSO, SOAO
	S.S.	1	100%	
		2	75%	OA, OS
		3	50%	DMO, OAO, AOA, OSO, SOA, AOS, SOS
		4	30%	DMOS, DMOA, MOAO, OAOA, OSOA, MOSO, DOAO, AOAO, OSOS, DOSO, SOSO, AOSO, SOAO
Int. Comp.	JUMP	1	80%	H
		2	100%	
		3	100%	
		4	100%	
		5	100%	
		6	100%	
	S.S.	1	100%	
		2	100%	
		3	67%	DDD
		4	0%	DDDG, DDDS, DDDD
		5	0%	DDDDG, DDDDS, DDDDD
		6	0%	DDDDDG, DDDDDS, DDDDDD

Figure 3.6: We can use the n-gram model to evaluate progressions. This table shows an analysis of progressions from JUMP Math and Skill Sharpeners (abbreviated as S.S.) for both fraction computation (Algorithm 3) and integer comparison (Algorithm 4). The table lists what percentage of the feasible n-grams, for various values of n, were covered by all problems together in the progression. The table also lists the missing n-grams. Integer comparison traces involving "E" are omitted because the goal of the exercise was to determine which number is larger.

is larger. Whether or not this omission is desirable is up to the educator; however, this example shows how a procedural analysis can find holes in progressions.

We can use the n-gram model presented in the partial ordering section to look for missing traces even more deeply. Figure 3.6 shows an analysis of the n-grams for four progressions for different problems. We compute all of the trace n-grams for each level for each progression, and identify missing n-grams by comparing them to a complete progression. Note that not all combinations of all letters are possible. For example, the integer comparison trace "H" cannot combine with any other letters because the program returns immediately after that statement.

3.5 Synthesis Of Progressions

We employ four different strategies for generating and gathering problems: (i) conducting brute force enumeration over inputs, (ii) using test input generation tools to explore the space of execution paths, (iii) using test input generation tools to generate problems that lead to the execution of a desired path by writing a straight-line program with assertions corresponding to that path, and (iv) collecting problems from textbooks.

3.5.1 Large-scale exploration of execution paths

Manual software testing, in general, and test input generation, in particular, are labor-intensive processes. The need to reduce the cost of software testing and maintenance has led to development of automated tools for generating test inputs that achieve a high level of coverage. Pex [83] is one such tool that automatically produces a small test suite with high code coverage for a .NET program. It performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths, following the idea of dynamic symbolic execution [24]. Pex uses the theorem prover and constraint solver Z3 [19] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems.

We applied Pex to generate problems for several elementary school mathematics procedures, shown in Figure 3.7. For each of these problems, we let Pex run for a certain length of time that we varied depending on how many paths we wanted to generate for a particular problem. We then analyzed the execution pathways of each of the problems that Pex generated and partitioned them by trace.

3.5.2 Generating problems to exercise a specific pathway

Pex's goal is to achieve high code coverage. While this allows Pex to generate a large variety of pathways quickly and efficiently, it also creates the possibility that it may not find certain pathways because its goal is statement coverage, not path coverage. However, we can still use Pex to generate problems for a specific trace by converting the procedure to a straight-line procedure. Essentially, we take the desired path and instrument the procedure with

Problem	Time	Traces	Constraints
Addition	64 min	1433	digits ≤ 4, addends ≤ 6
Fraction Computation	16 min	72	fractions ≤ 4, numerator ≤ 20, denominator ≤ 20, operations $\in$ addition, subtraction
Fraction Reduction	2 min	29	numerator ≤ 50, denominator ≤ 50, numerator $\leq$ denominator
Integer Comparison	2 min	26	digits ≤ 8
Subtraction	2 min	157	digits ≤ 6, difference ≥ 0
Prime Factorization	2 min	70	number ≤ 100000

Figure 3.7: We used Pex to generate many practice problems for these mathematical procedures. In each exploration, we let Pex run for the indicated length of time and it produced the indicated number of unique traces for that problem. Pex can clearly generate many interesting inputs by directly exploring traces through the procedure. Figure 3.4 shows the percentage of each textbook progression that this exploration was able to cover.

assertions that will constrain the execution of the procedure along that particular path.

Using this method on Algorithm 3, we were able to generate an input problem that creates the trace "DAAA" in 40 seconds and "DMMMMMMS" in 35 seconds.

3.6 Generating textbook math problems

Figure 3.7 shows how we were able to generate large numbers of unique traces with Pex for several math topics. This table indicates how long we let Pex run and how many unique traces it produced. Figure 3.4 shows what percentage of the set of traces found in the textbooks we were able to generate. It is not 100% for all problems, but we can use the straight-line program method to generate the missing buckets. These results show that using test input generation tools is sufficient for generating problems in this domain.

3.7 Generating Dragonbox Levels

DragonBox (WeWantToKnow 2012), shown in Figure 3.8, is a video game that became the most purchased game in Norway on the Apple App Store [50]. It features game mechanics that involve solving algebra equations. The game does not appear to be an algebra lesson, but the player cannot succeed without learning algebraic simplification and variable isolation. These topics are learned through the level progression. *DragonBox* represents al-

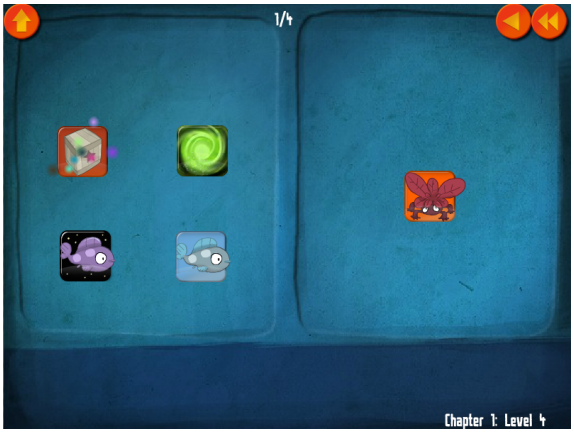


Figure 3.8: A level of DragonBox. The goal is to simplify an algebraic equation by isolating the variable, indicated by a box. Each half of the screen represents a side of the equation. The green spiral represents a zero and the two fish cards represent 5 and -5. The bug card represents 10. The player can solve this level by executing the following three rules: $+0 \to \emptyset$, $a + (-a) \to 0$, and $+0 \to \emptyset$. ©WeWantToKnow

gebraic expression trees visually as a set of cards that represent numbers and variables. On each level, the screen is divided into two halves that represent the two sides of an equation. By dragging cards around, the player can add, multiply, or divide both sides of the equation by a number or a variable, and perform algebraic simplifications. The goal for each level is to isolate the *DragonBox* card, which represents an unknown variable.

DragonBox levels are essentially expression trees that represent algebraic equations. Leaf nodes can be a variable "x" or an integer. Integers can have multiple forms. They can either be an animal card with a unique picture, a number, or a constant like "a" or "c". Since these forms are essentially equivalent, we only focused on generating integer values and rendered all of these values with an animal card. Internal nodes can be either addition, multiplication, or division. The *DragonBox* interface imposes an ordering on these operations: addition cannot appear below multiplication or division, and division cannot appear below multiplication.

The procedure we wrote to solve *DragonBox* levels is described in Algorithm 5. This algorithm is sufficient to solve all of the levels of the expert progression except for three at the end, which require a more complex algorithm that we leave for future work.

Because there are a huge number of possible expression trees for both the left and right-hand sides of the equation, we used Pex to rapidly explore pathways and generate a good coverage of test levels. Pex ran for two hours and generated 781 unique levels. These levels corresponded to 152 unique traces. These traces covered roughly 45% of the original progression traces, but represented a large number of unique and interesting levels.

3.7.1 Evaluating the path-based partial ordering

In order to evaluate the synthesized levels and determine whether or not they were equivalent to the expert-designed levels, we conducted a user study. Twenty participants were recruited through an internal email list at the University of Washington that is read by faculty, staff, and students.

Each participant played 30 levels of *DragonBox*. These levels were organized into fifteen pairs, each consisting of one automatically generated level and one expert-designed level from the original progression. After playing the two levels in the pair, each participant was asked to judge the difficulty of the first level compared to the second level. Five options were given: "much easier", "slightly easier", "about the same", "slightly harder", and "much harder". Participants played the pairs in a random order and the order of levels within each pair was also randomized.

We organized the levels such that five of the pairs contained levels that our model considered to be equivalent. For these pairs, the traces were exactly the same. In four of the level pairs, the trace of the original level subsumed the trace of the generated level according to our path-based partial ordering, and therefore our model considered the original level to be more difficult. In another three level pairs, the trace of the generated level subsumed the trace of the original level, and the original level was considered to be easier. For the final three pairs, we picked a generated level and an original level with the same trace length but different actions, indicating different conditional branching in the trace. As a result, our model specified no partial ordering for these pairs. Within each category, we tried to pick pairs so that the set would have a range of trace lengths and actions (like "add a to both sides"). For the "easier" and "harder" categories, the more difficult trace included one to

three more actions than the easier trace.

Partial ordering was better than random by factor of 3

We first tried to determine whether our partial ordering could predict whether the level pairs were equal in difficulty or not. We found that the percentage of participants who considered a level pair to be equal increased from 25% when our model predicted that the pair was unequal to 62% when our model predicted that the pair was equal. This was statistically significant, $\chi^2(1,\ 241) = 33.97$, $p < 0.0001$. This indicates that our model reflects some human intution about level equality.

We then compared the root-mean-square error of our model's predictions with that of other baselines. We first grouped the "slightly harder" and "much harder" responses together, giving them a value of 1, and we did the same for "slightly easier" and "much easier" giving those responses a value of -1. We assigned a value of 0 to an equal response. We then measured the average response for each level, and computed the root-mean-square error of our model's predictions for each level. The root-mean-square error of our model overall was 0.35, whereas the error for making random predictions each time was 1.00, roughly three times as much.

Just using the trace length is insufficient

We observed that a simpler metric, the trace length, makes many of the same predictions as our path-based partial ordering. Therefore, to motivate the need for our more complex partial ordering, we compared it with this simpler model. In Algorithm 5, whenever our model predicts that level L_1 is harder than level L_2, it is always the case that the trace of L_1 has more statements than the trace of L_2. Therefore, for this particular procedure, a simpler model that just compares the length of these traces makes similar predictions, and actually makes the same predictions as our path-based partial ordering in cases where one trace subsumes another.

However, our partial ordering does not make a prediction when one trace does not clearly subsume the other, indicating where more data is needed. One such case is when

Algorithm 5 Dragon Box

```
1:  procedure DragonBox(left, right)
2:      simplify(left, right)
3:      if x ∈ denom(left) ∨ x ∈ denom(right) then
4:          multiply(left, right, x)
5:          simplify(left, right)
6:      end if
7:      if x ∈ num(left) then
8:          variableSide ← left
9:          otherSide ← right
10:     else
11:         variableSide ← right
12:         otherSide ← left
13:     end if
14:     while variableSide ≠ x do
15:         isolate(variableSide, otherSide)
16:         simplify(otherSide, variableSide)
17:     end while
18: end procedure
19: procedure Simplify(node)
20:     if node = + ∧ child = 0 then
21:         child ← null                                  ▷ Remove +0
22:     else if node = + ∧ a ∈ child_1 ∧ −a ∈ child_2 then
23:         child_1 ← 0                                   ▷ Remove a + (−a)
24:         child_2 ← null
25:     else if node = × ∧ child = 1 then
26:         child ← null                                  ▷ Remove ×1
27:     else if node = ÷ ∧ a ∈ child_1 ∧ a ∈ child_2 then
28:         child_1 ← 1                                   ▷ Remove a ÷ a
29:         child_2 ← null
30:     end if
31: end procedure
32: procedure Isolate(varSide, otherSide)
33:     varChild ← varSide.child with variable
34:     nonVarChild ← other child
35:     if varSide = + then
36:         subtract(varSide, otherSide, nonVarChild)
37:     else if varSide = × then
38:         divide(varSide, otherSide, nonVarChild)
39:     else if varSide = ÷ then
40:         multiply(varSide, otherSide, nonVarChild)
41:     end if
42: end procedure
```

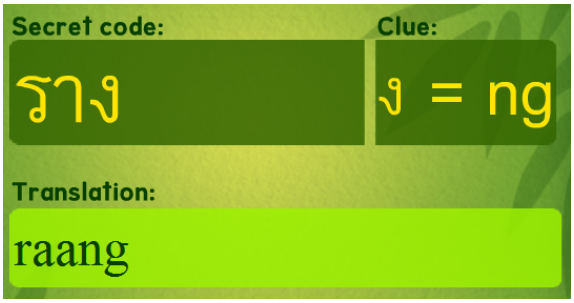


Figure 3.9: *Bpan Yaa* asks the player to transliterate words from Thai to English. It features a progression of over 1,000 words, each classified and ordered according to the execution trace obtained by running a transliteration algorithm on that word. Forgetfulness is a big problem in this domain. If the player gets a problem wrong, the system identifies the specific statement in the program he or she could not execute correctly, and selects a simpler word to help the player resolve that specific error.

two traces have the same number of actions but the actions themselves are different. In this event, a trace-length metric would predict that these levels are equal, although this is likely naïve because the actions themselves are different and those differences would likely cause a human to feel that the levels are different.

To examine this, we calculated the error for the trace-length metric on the three level pairs with the same trace length but different actions. This error was 0.51, which is higher than the 0.35 error that the path-based partial ordering achieved on the rest of the problems. This suggests that looking at trace length alone is not sufficient, and we need a model that carefully examines the sequence of statements within a trace.

3.8 A progression for learning the Thai alphabet

We used our framework to create a game for learning the Thai alphabet called *Bpan Yaa*, shown in Figure 3.9. There are multiple complexities of the Thai writing system that make it non-trivial to transliterate words from a Thai alphabet to a corresponding pronunciation using English letters, such as the following:

- Although letters are generally pronounced in a left-to-right order, this is not always

http://www.mochigames.com/games/bpan-yaa/

true. Certain vowels are written before, above, or below a consonant, even though they are pronounced after that consonant.

- Vowels are sometimes omitted and must be inferred.

- Consonants are often pronounced differently if they appear at the beginning or the end of a syllable.

- Although pronunciation depends on how a word is broken into syllables, these divisions are often not immediately clear, and must be determined from a set of rules.

These complexities necessitate a more structured learning approach than simple combinations of letters. There are multiple procedures for transliterating words from Thai to English, but one that captures all of the elements listed above is the "Thai-language.com Enhanced Phonemic Transcription" system. For this procedure, 1-grams range from simple operations like basic conversions from a Thai letter to a corresponding English sound, to more complex processes such as determining whether a consonant closes the current syllable as a final consonant or starts the next one as an initial consonant. Larger n-grams, such as 2-grams and 3-grams, start to capture syllables, and even larger n-grams capture full words.

To create a progression, we gathered a list of approximately 1000 of the most common words in Thai that appeared on the same website. We computed the trace for each word, identified n-grams, and calculated a partial ordering based on n-grams in a manner similar to Figure 3.2. We sorted 1-grams by how frequently they appeared in the word list. We then constructed a progression in the following manner:

1. Pick the most common 1-gram not already introduced.

2. Add the simplest word that contains this 1-gram and has no predecessors that have not been introduced.

---

http://www.thai-language.com

3. Continually traverse the graph in a manner similar to Figure 3.2. If there is a node that has all of its predecessors added to the progression, add that node.

4. When there are no more words available without introducing a new 1-gram, go back to step 1.

The Bpan Yaa game continually follows this progression and asks the player to transliterate words. If the player transliterates the word correctly, the game moves on to the next word in the progression. When new 1-grams are introduced, there is a small tutorial message that explains how to pronounce words with this new 1-gram. If the player answers incorrectly, the game identifies the point in the trace at which the answer deviated from the expected answer and marks that 1-gram as failed. The game then uses the partial ordering to identify the simplest word in the entire progression that contains the failed 1-gram. It gives this word next and provides an explanation for the failed 1-gram. When the player has relearned this 1-gram, the game returns to where it was in the progression and moves on.

3.9 Infinite Refraction

Refraction is a fractions-based puzzle game in which players solve spatial puzzles by splitting lasers into fractional amounts. Each level is played on a grid that contains laser sources, target spaceships, and asteroids, as shown in Figure 3.10. Each target spaceship requires a fractional amount of laser power, indicated by a yellow number on the ship. The player can satisfy the targets by placing pieces that change the laser direction and pieces that split the laser into two or three equal parts. All targets must be correctly satisfied at the same time to complete the puzzle. Although *Refraction* was built to teach fractions, it has found popularity with players of all ages and has been played about one million times. *Refraction* is freely available and can be played by anyone with a web browser and Adobe Flash.

The original version of *Refraction* contains 61 levels that were designed by an expert game designer who spent over 100 hours designing levels. As the player progresses, the levels gradually increase in difficulty through the introduction of new concepts, such as benders,

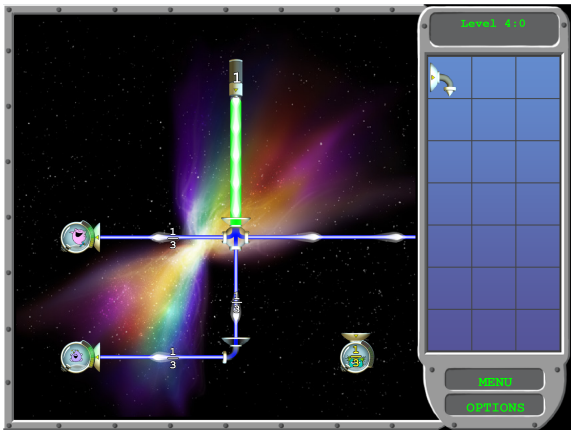


Figure 3.10: A level of *Refraction*. The goal is to use the pieces on the right to split lasers into fractional values and redirect them to satisfy the target spaceships. The user can pick up and put down pieces by clicking on them. The grid interface, pipe-flow game mechanics, and spatial reasoning puzzles are similar to many other puzzle games.

splitters, combiners, multiple sources, and multiple targets. The levels also increase in difficulty by combining base concepts together in way that necessitates more complex search processes, such as levels that require bending a laser clockwise and then counterclockwise to circumvent an obstacle, or levels that require splitting a laser into halves and then into thirds in order to produce one sixth. Together with Eric Butler and Adam M. Smith, I applied my framework to create a level progression for *Refraction*.

Describing a player's exact solution process for *Refraction* is difficult, as there are many different procedures that may lead to the correct answer. Furthermore, there are often multiple configurations of pieces that can solve a level correctly. Although most of these solutions are slight perturbations of other solutions, solutions can also differ in nontrivial ways.

To control the difficulty of *Refraction* levels, instead of specifying an exact procedure, we consider smaller procedural units that must show up somewhere in the solution process, for all valid solutions to the problem. We call these *solution features*. Since our solutions are not linear procedures, we cannot directly apply the n-gram analysis described previously. We must extend it to work with this non-linear structure.

To do this, we model features of the solution to each level. As an analogue to n-

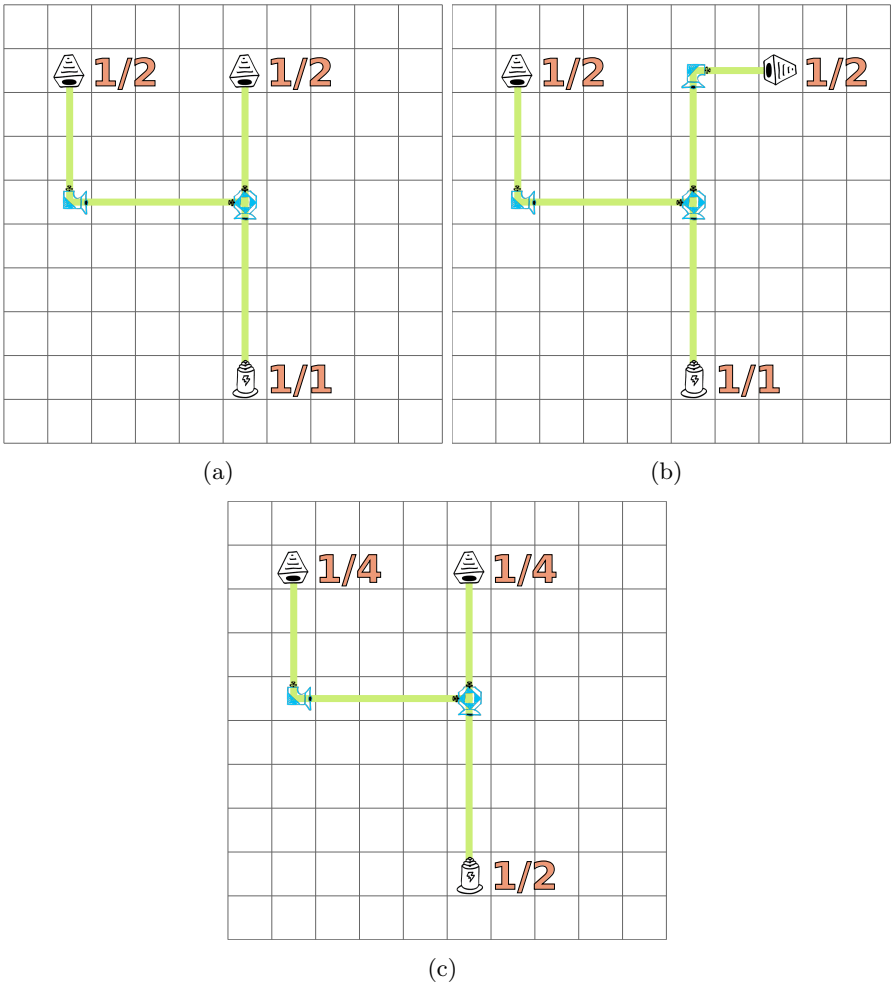


Figure 3.11: Three example levels for *Refraction*, to illustrate solution features and graphlets. When considering the *solution graph*, levels (a) and (b) have the same 1-graphlets. They differ in 2-graphlets: (a) has a splitter-target chain, while (b) does not. They differ on several 3-graphlets; notably, (a) has a splitter branching into a bender and target while (b) has a splitter branching into 2 benders. However, the solution graph is insufficient to distinguish (a) from (c). We need the additional *math graph* to capture differences between them with graphlets.

grams for procedures, we use *graphlets*, which are small connected non-isomorphic induced subgraphs of a larger graph [66]. Figure 3.11 shows some example *Refraction* levels to explain graphlets. In *Refraction*, graphlets on the solution graph capture ideas such as the following: an example of a 1-graphlet is that a two-splitter is used; an example 2-graphlet is

a laser edge linking a bender and a two-splitter; an example 3-graphlet is a length-3 chain including two two-splitters and a three-splitter. Since our level generation process ensured that all solutions had isomorphic graphs, we can be sure that every solution involves those graphlets.

To generate levels, we enumerate combinations of desired solution n-grams using an answer set program [22], following the patterns introduced in Variations Forever [78]. We then applied techniques introduced by Smith et al. [76] to generate levels with constraints ensuring that all desired solution n-grams are present for all possible solutions to the level. This process generated over 9,000 levels in one hour while running on a 60-node cluster of machines. Once we had a series of levels, we assembled them into a progression that continually introduced new n-grams.

3.9.1 Evaluation

Seeking to determine if our generated content was comparable to that produced by experts, we deployed both versions of *Refraction* and performed an A/B test to measure engagement. I have previously used A/B testing and multivariate testing to study the impact of game design on engagement [6, 5, 7]. We posted the game on a popular flash game website, Newgrounds, as well as through MochiMedia, which distributes through the main MochiGames website and dozens of other affiliated game websites. The game was released under the title *Infinite Refraction*. We gathered data from 2,377 players in this experiment. 1,221 randomly selected players played the automatically generated level progression and 1,156 players played a version that included the expert-designed level progression from the original *Refraction*.

Since this game was released "in the wild," we did not have direct access to players, only their play data. Therefore, we measured engagement by looking at the total play time for each player in a manner similar to [7, 51]. We first performed a series of statistical tests on the distribution of time played in each condition. Our data was not normally distributed,

---

www.newgrounds.com

www.mochimedia.com

www.mochigames.com

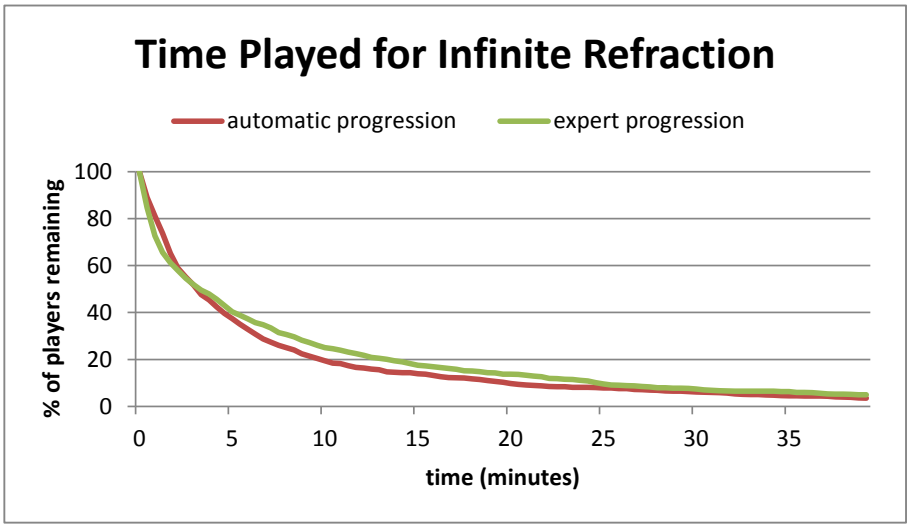


Figure 3.12: Comparison of our automatically-generated progression with the expert-generated progression from *Refraction*. The x-axis is time in minutes and the y-axis is the percentage of players who played for at least that much time. The median values are very similar: approximately 3 minutes. Although our framework's progression performs slightly worse, it is certainly *comparable* to the expert progression. These results suggest that our framework was able to replicate much of the wisdom and expertise contained in the original progression. In contrast to the original design process, which included many painstaking hours of crafting and organizing levels by hand, our framework only requires the designer to specify the rules of the game, a process for solving the game that our framework deconstructs into base conceptual units, and a few designed-tuned weights that control the order and speed of introduction of these units.

so we relied on a nonparametric test, the Wilcoxon-Kruskal-Wallis test, also known as the Mann-Whitney U-test. Despite having over one thousand players in each condition, the Mann-Whitney U-test showed no significant difference between the two populations on time played, with a median of 184 seconds for the automatically-generated progression and 199 seconds for the expert-crafted progression. As the median time is only 8% lower for the automatically generated progression, these result suggest that our automatic methods are capable of engaging players at a level that is comparable to expert-designed content. Figure 3.12 shows this in greater detail.

3.10 Discussion and Future Work

We have defined a trace-based framework for using execution traces to explore the space of problem progressions for any procedural task that can be specified as a computer program. We have shown how this trace-based framework can categorize problems according to their traces, define partial orderings that can compare two problems and assemble a set of problems into a progression, and analyze the thoroughness and depth of existing progressions.

Our framework has allowed us to borrow ideas from test input generation in software engineering and apply them to problem generation. Test input generation tools can automatically explore pathways through a procedure, systematically generate large sets of problems that correspond to various unique pathways through this procedure, and generate problems that follow a particular desired path. We applied the framework to two domains: early mathematics education, and level generation for a interactive puzzle game. Our framework synthesized hundreds of unique levels for a popular algebra puzzle game and we conducted a user study to confirm that these levels are similar to the expert-designed progression.

We applied these ideas to create implementations for two educational games, *Refraction* and *Bpan Yaa*. Our study with 2,377 *Refraction* players showed that the median player was engaged for 92% as long as in an expert-designed progression, demonstrating that our framework is capable of producing content on par with expert design.

Although most of grade-school mathematics education is procedural in nature, some of it also involves conceptual elements that our framework does not fully address. Future work will allow us to include some of these conceptual elements.

Chapter 4

STEP-BY-STEP DEMONSTRATIONS

In the previous chapter, I introduced a framework for automatically generating practice problem progressions. While practice problems are useful, we still need to be able to teach the learner how to actually solve a particular practice problem. In this chapter, I will present a system I developed with Eleanor O'Rourke, Sumit Gulwani, and Zoran Popović that is capable of rapidly producing step-by-step demonstrations for procedural skills.

Demonstrating procedures in a robust way is challenging because for any given procedure there can be hundreds of practice problems we might potentially want to explain. Furthermore, some domains of education, such as K-12 math, include dozens of procedures. Typically, designers write tutorials to explain a specific problem. By writing templates, they might be able to construct tutorials that work for multiple problems, but this will still only work for one procedure. Ideally, we would have an efficient technique for rapidly generating demonstrations for every practice problem for a given procedure, and be able to reuse large portions of this effort when demonstrating a different procedure.

We accomplish this by borrowing ideas from debuggers and Integrated Development Environments. Software engineers have spent many years demonstrating to programmers how their programs are behaving; we can leverage some of this effort in order to demonstrate a procedure to a student. To do this, the designer must encode the procedural skill as as computer program written in a domain-specific language, which we call the Thought Process Language (TPL). Our system then steps through this algorithm line-by-line, and uses mappings between programming language constructs and visual objects in the user interface to automatically explain each step. While encoding the thought process in TPL and creating the interface hooks may require more effort than creating a tutorial for a specific practice problem, our system is capable of producing systematic demonstrations for the entire space of problems that can be solved by the procedure. Furthermore, for some

domains, such as K-12 grid mathematics, a single set of interface mappings can be used to teach many procedural skills. This chapter will explain the key ideas behind the system and show that it can be applied to multiple procedures in K-12 math. Evaluating the usability of the authoring framework and the effectiveness of the tutorials themselves is beyond the scope of this work.

This chapter is organized as follows. Section 4.1 describes TPL. Sections 4.2 and 4.3 describe the architecture of the system. Section 4.4 describes how we can use procedural traces to introduce new information only when it is needed. Section 4.5 describes how we evaluated our system by trying it on multiple procedural skills in K-12 math.

4.1 Thought Process Language

In our system, procedures must be encoded in the Thought Process Language (TPL). An example of TPL algorithm for subtraction is given in Algorithm 6. The constructs used in this language are standard to many programming languages, and are not a contribution of this thesis. TPL includes only the key constructs required to cover a large portion of procedural knowledge. TPL includes basic primitives like variables and constants, as well as expressions that can include integer operators ($+$, $-$, $*$, $/$) or Boolean operators ($<$, $>$, $==$, $!=$). TPL has three types of statements: assignments, conditionals, and loops (for, while, do-while, for-each). We based TPL on object-oriented programming languages, so we also support objects, object fields, and object methods.

4.2 Compiler and Interpreter

Our system generates step-by-step explanations by stepping through a TPL procedure line-by-line in a manner similar to a debugger. This requires two system components: a compiler and an interpreter. These components can easily be reused across applications.

Figure 4.1 shows a diagram of the system. The TPL algorithm is passed to the compiler as input. The compiler processes this algorithm to produce a compiled program composed of programming language objects that can be understood by the interpreter. This program is passed to the interpreter, which steps through the language objects for the current example problem. The interpreter maintains the program's context: the set of variables that are

Algorithm 6 Given as input a 2D array *table* containing a minuend p, represented as a set of digits $p_0, p_1, ..., p_m$ located in cells (0,0) to (0, m), and a subtrahend q, represented as a set of digits $q_0, q_1, ..., q_n$ in cells (1,$m - n$) to (1, m), subtract:

```
 1: procedure SUBTRACT(table)
 2:     for each column in table.GetColsFromRight() do
 3:         top := column.GetCell(0)
 4:         bottom := column.GetCell(1)
 5:         if bottom.NotEmpty() then
 6:             if top.value < bottom.value then
 7:                 borrow := table.LeftOf(top)
 8:                 while borrow.value == 0 do
 9:                     borrow := table.LeftOf(borrow)
10:                 end while
11:                 borrow.value := borrow.value − 1
12:                 while borrow.NotNextTo(top) do
13:                     borrow := table.RightOf(borrow)
14:                     borrow.value := 9
15:                 end while
16:                 top.value := top.value + 10
17:             end if
18:             result := column.GetCell(2)
19:             result.value := top.value − bottom.value
20:         else
21:             result := column.GetCell(2)
22:             result.value := top.value
23:         end if
24:     end for
25: end procedure
```

currently stored in memory. As the interpreter steps through and executes each statement, it throws events that indicate when new variables are put into context and when assignment statements, loops, and conditionals are executed.

A full list of the events generated by the interpreter are given in Figure 4.2. Each event contains relevant information, such as the name of the variable that was put into context or the programming language object associated with the executed statement. These events are passed to a set of domain-specific interface hooks, which are mappings between events and visual objects in the application interface. These interface hooks are used to generate visual explanations of each step in the procedure.

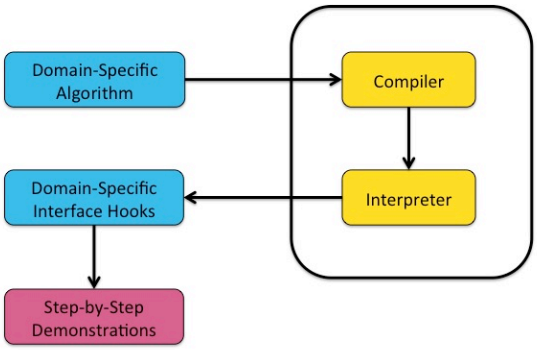


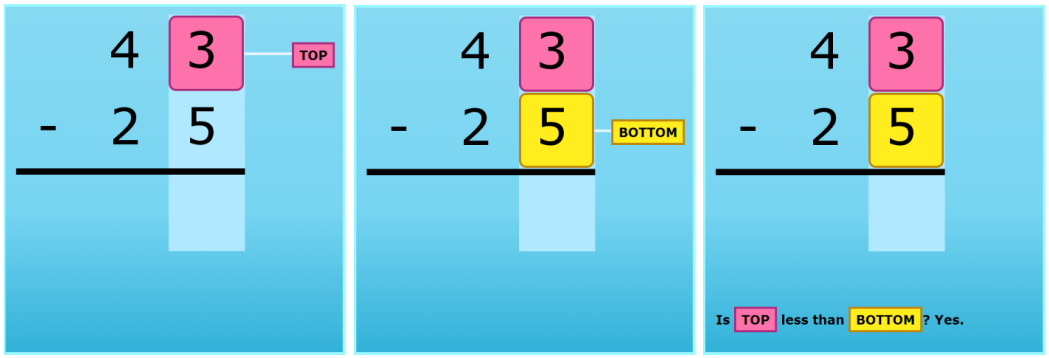
Figure 4.1: This diagram shows our framework workflow. The domain-specific TPL algorithm representing the problem-solving thought process is given as input. The algorithm is compiled and then the interpreter steps through it line-by-line. The interpreter throws events while executing the TPL program, which are passed to the domain-specific interface hooks. The interface hooks produce visual step-by-step explanations.

Event	Interface Hook
Put variable in context	Interface Highlighting
Remove variable from context	Interface Highlighting
Execute assignment statement	Textual Explanation
Execute conditional statement	Textual Explanation
Execute return statement	Textual Explanation

Figure 4.2: The left column lists the set of events that the interpreter throws while executing the TPL algorithm. The right column lists the type of visualization that is used to explain each type of event.

4.3 Interface Hooks

To automatically generate explanations for each step in the TPL algorithm, we use a set of domain-specific functions that map events thrown by the interpreter to visual explanations on the screen. We refer to these functions as *interface hooks*. These mappings are written a single time for a given domain of education (such as two-dimensional spreadsheet math problems) and can be reused for many procedures in that domain. We have implemented two different types of visual explanations: text messages and interface highlighting. These visualizations are triggered by events thrown by the interpreter, as defined in Figure 4.2.



(a) (b) (c)

Figure 4.3: Examples of explanations generated for the procedure to solve subtraction problems. Figure (a) shows the explanation of an assignment statement of the variable top to the top cell in a column. Figure (b) shows the explanation of an assignment statement of the variable bottom to the bottom cell in a column. Since the variable top is still in scope, it remains highlighted. Figure (c) shows the the explanation of the conditional statement that determines whether or not borrowing is needed, which references the variables top and bottom.

The next two sections describe how text messages and interface highlighting are generated automatically from interpreter events.

Text Explanations

Text-based explanations are commonly used to teach procedural tasks in both software tutorials and intelligent tutoring systems [14, 85]. These explanations are either written by hand for each practice problem or generated using detailed templates. This process can be time consuming [10, 25, 85]. We use text messages to describe assignment statements, loops, and conditionals. To produce these explanations automatically, we try to leverage linguistic information that is encoded within the procedure itself, such as variable names.

Interface hooks can generate text messages directly from the names of variables and functions that are defined in the algorithm. We also use general templates to describe more complex branching statements. For example, consider the procedure for solving subtraction problems that was described above. We explain the assignment statement defined in line 3, $top := column.GetCell(0)$, by displaying the name of the variable in a text box as shown in

Figure 4.3(a). We determine what object the text box should point to in the user interface by mapping the object stored in the variable *top*, in this case a cell in the grid, to a visual location on the screen.

We can specify different interface hooks for specific variables or variable types. For example, on line 19, $result.value := top.value - bottom.value$, we can create an interface hook that explains assignments to the "value" field of the "cell" object by converting the right-side expression to natural language text. We do this with a template that is filled with the variable names of the objects referenced in the expression and replacing operators with corresponding text. For example, this line becomes "set result to top minus bottom."

We also use templates to explain the thought process behind branching statements. For example, we can explain the conditional statement defined in line 6 of the subtraction algorithm, *if* $top.value < bottom.value$, by using a template that phrases the conditional's Boolean expression as a question. The template is filled with the variable names of the objects referenced in the Boolean expression, and the comparison operator $<$ is replaced with "less than". This produces the question "Is top less than bottom? Yes." as shown in Figure 4.3(c). We can add the answer to the question, either "Yes" or "No", based on whether the conditional statement evaluates to true or false.

In practice, we have found that a small number of interface hooks must be written for a given education domain. For example, 12 interface hooks are sufficient for explaining most procedures in K-12 spreadsheet math.

Interface Highlighting

Our system also highlights visual objects in the user interface. Many effective software tutorials present explanations in the application context and highlight relevant interface objects [10, 38]. This idea is supported by situated learning, a popular pedagogical model based around teaching concepts in the same context where they will be applied [46]. In order to highlight the set of interface objects that are relevant to the current line of the procedure without overwhelming the learner, we highlight objects based on the scope of variables within the procedure context.

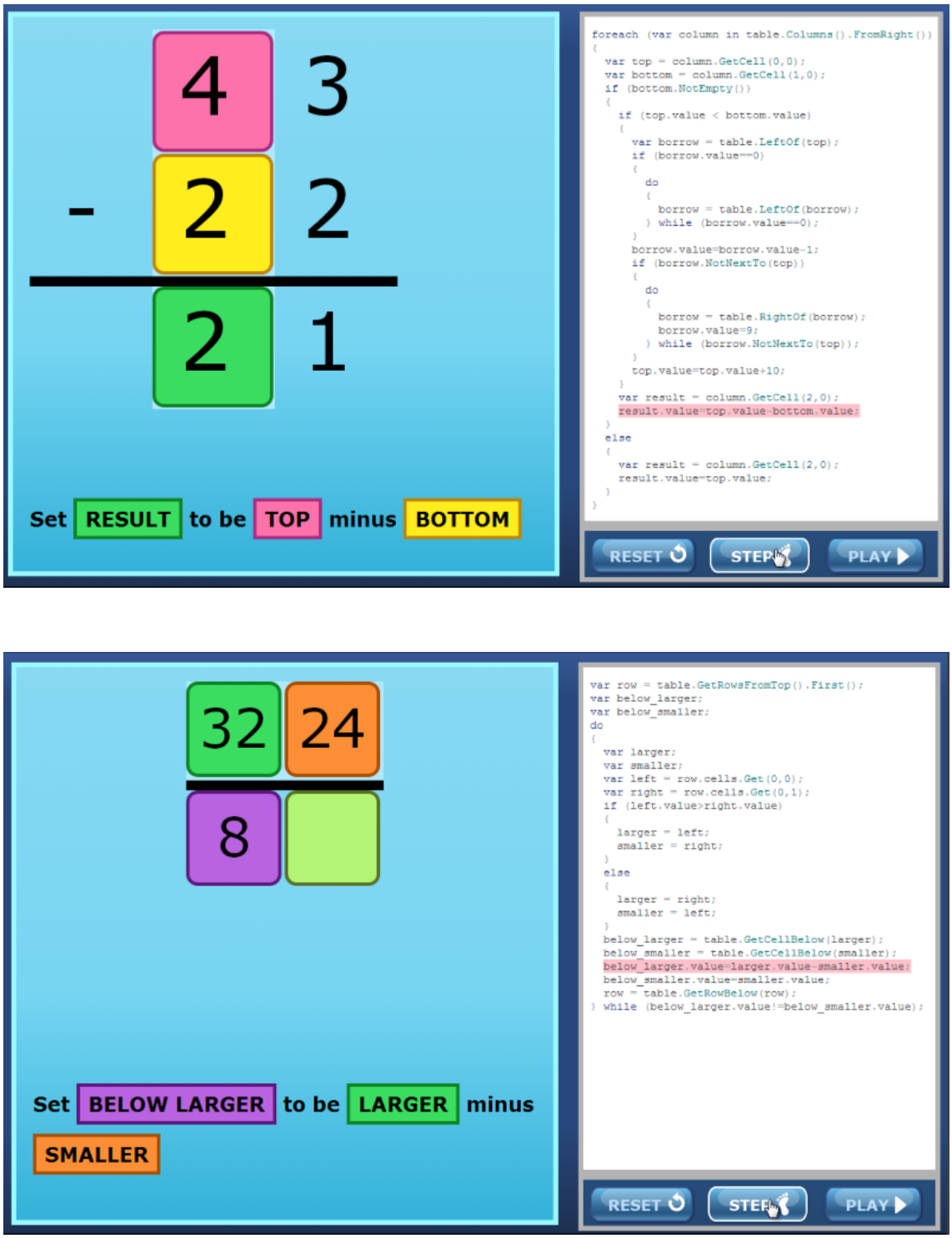


Figure 4.4: Screenshots of our K-12 Grid Mathematics Application. The top shows the application set up for a subtraction problem, and the bottom shows a greatest common factor problem. The window on the right displays the algorithm used to generate tutorials.

Many imperative languages have a sense of *variable scope*, or the context in which a variable is defined and can be referenced. A variable that is defined within a branch of a conditional statement, like the *borrow* variable in the subtraction procedure, will only be in scope while that branch is executing. We draw a connection between variable scope and working memory, by observing a parallel between the objects the learner needs to keep in working memory and the variables that are currently in scope.

We highlight objects by mapping variable types in the procedure and visual objects in the user interface. For example, variables like *top* and *bottom* in the subtraction example refer to grid cells that hold single digits of a number. The tutorial designer defines interface hooks that map grid cells to physical locations in the user interface. Using these mappings, our system highlights the objects referenced by the *top* and *bottom* variables as shown in Figure 4.3(b).

4.4 Layering

The previous chapter motivated the importance of practice problem progressions for learning a procedural skill. Teachers may wish to demonstrate a full progression of practice problems, introducing new skills when the progression becomes more challenging and new lines of the procedure are executed. For simple introductory problems, we may not want to explain the entire procedure. For example, if the learner is working on a subtraction problem like 5 minus 3, we should not explain borrowing because it is not required to solve the problem. To produce explanations with the correct level of detail, we identify which statements should be explained and which can be skipped.

We accomplish this goal by tracking which statements have been executed in the problems that the learner has solved so far. We only explain a conditional statement if both branches of the conditional have been encountered. If only one branch has been executed, then the learner does not need to be made aware of the decision. Similarly, we only explain loop statements if the loop body is executed multiple times. More generally, if a particular loop or conditional were removed and the procedure would have the same behavior for all problems solved up until this point, then that loop or conditional does not yet need to be explained. This allows us to describe only the steps required to solve the current problem.

4.5 Evaluation

We have built a prototype of our system that works for multiple procedures in K-12 spreadsheet mathematics. This refers to all types of problems that can be solved by writing integers on a two-dimensional grid. This covers a large portion of K-12 math, including addition, subtraction, long division, prime factorization, and Euclid's algorithm for calculating the greatest common factor of two integers.

Screenshots of the application interface that has been set up to solve subtraction problems and greatest common factor problems are shown in Figure 4.4. For this application, we implemented TPL in C#. The right-hand side of the application features a window in which the user can write TPL code. To view a step-by-step explanation of a problem, the user must input the problem into the two-dimensional grid in the main window of the user interface. To step through a demonstration for that example problem, the user repeatedly clicks the "Step" button in the bottom-right of the interface. At each step, the current line of the procedure is highlighted, and a visual explanation of that line is displayed in the main window.

The textual explanations and interface highlighting are produced from a set of interface hooks that map events thrown by the system's interpreter to visualizations in the user interface. For this prototype, we defined an interface hook to highlight cells in the grid in response to "put variable in context" events thrown by the interpreter. We also defined a set of interface hooks that display text messages that reference grid cells by coloring their variable names to match the highlighting. Variable names are used to generate explanations for assignment statements and general-purpose templates are used to generate explanations for conditional statements. Given one set of mappings between interpreter events and visual objects, we are able to explain all TPL procedures that solve spreadsheet math problems, and all practice problems for each procedure.

4.6 Discussion and Future Work

This chapter presented a new framework for authoring step-by-step demonstrations of procedural skills. Instead of writing tutorials for a specific practice problem, the author encodes

the thought process in TPL, and the system automatically generates demonstrations that explain each line of code in a manner similar to a debugger. This approach has a number of advantages. Given a set of mappings between programming language constructs and visual objects in the application interface, we can produce explanations for an entire domain of education, such as two-dimensional grid mathematics. Furthermore, we can gradually introduce new concepts to students as they become relevant. This framework has the potential to significantly reduce the effort of creating step-by-step demonstrations.

Constructing demonstrations in this way enables rapid development of other instructional modalities for a large space of procedures and practice problems. One possibility is just-in-time feedback that triggers only when the student does something wrong. Since traditional curricula often shift abruptly from fully demonstrating each problem to expecting students to solve problems completely on their own, another useful extension is a system that gradually *fades worked-out examples* by slowly reducing the amount of explanation. These avenues are left for future work.

One limitation of this framework is that educators who cannot program cannot write TPL procedures. However, if we could learn such procedures from demonstrations of their behavior, we could potentially make this framework far more accessible. The next chapter will introduce a system that *can* learn two-dimensional spreadsheet math programs by demonstration.

Chapter 5

DIAGNOSIS OF MISCONCEPTIONS

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 [84, 9]. 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 [84] 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 that we encountered for finding the greatest common factor (GCF) of two numbers (Figure 5.1). The space of buggy algorithms is massive and continually expanding; Van Lehn has identified over 100 bugs that students demonstrate in subtraction alone [84]. 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 [84, 73]. Such approaches typically begin with the procedure to be learned and

model errors as small perturbations 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 [55], 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 [59, 44] and template-based approaches [80, 82, 81]. 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, this chapter introduces a novel framework I created with Sumit Gulwani and Zoran Popović 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 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 [9] from real student data. We show the generality of our method by applying it to spreadsheet table transformations [33], an important domain in end-user programming [28].

The chapter 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 [80], 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.

5.1 Motivating Examples

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

5.1.1 Correct Procedures

Figure 5.1 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.

Although 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."

5.1.2 "Buggy" Procedures

Student errors fall into multiple categories, including careless mistakes, incorrect fact recall, and systematic misconceptions in which the wrong algorithm is used [84, 9]. 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 thesis. 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.

```
GCF: Euclid's Algorithm(Sheet T, int I_1, int I_2)
  Assume T[0,0] contains I_1 and T[0,1] contains I_2.
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];
4     T[j + 1,1] := T[j,1];
5   else:
6     T[j + 1,0] := T[j,0];
7     T[j + 1,1] := T[j,1] − T[j,0];
8 return T[j,0];

GCF: Successive Division(Sheet T, int I_1, int I_2)
1 Assume T[0,0] contains I_1 and T[0,1] contains I_2.
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[] I_1)
  Assume that T[0,(0,m)] contains I_1[0],··,I_1[m].
1 for (j := 0; ¬Coprime(T[j,(0,I_1.right)]); j := j + 1):
2   T[j,−1] := LeastPrimeDivisor(T[j,(0,I_1.right)]);
3   for (i := 0; i < I_1.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 5.1: Three algorithms for computing 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.

Ashlock [9] 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 [9]): 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.

5.2 Formalism

In this section, we formalize a mathematical problem in Section 5.2.1, its solution in Section 5.2.2, a procedural language to automatically compute such solutions in Section 5.2.3, and the inductive synthesis problem to automatically synthesize such procedures from examples in Section 5.2.4.

5.2.1 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:

The variable values of these regions are as follows:

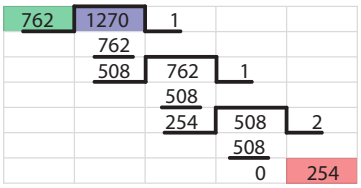
Input	*top*	*left*	*bottom*	*right*
762	0	0	0	0
1270	0	1	0	1

5.2.2 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 a 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 Section 5.2.1. The state of the spreadsheet after the above problem has been solved is shown below, with the answer highlighted in red:



5.2.3 Procedure

In this section, we introduce a fairly general-purpose language that can be used to compute the kind of solutions specified in Section 5.2.2, when given as input the kind of problem specification specified in Section 5.2.1. The types in our language are scalars such as digits and numbers, and vectors such as arrays of digits and arrays of numbers.

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

Figure 5.2: Syntax of Programs.

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 5.2 describes the syntax of our language. Programs P contain a `Sequence` of statements S. Statements can take one of three forms. One is $\texttt{Update}(k_1, k_2, e, h)$, which writes the value computed by e into the grid cell at row k_1 and column k_2 with emphasis h. h 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

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

Figure 5.3: Syntax of angelic programs.

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 iterator 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 Section 5.2.1 such as $I.right$.

$$
\begin{aligned}
[\![\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}, h)]\!] &= \{\texttt{Update}(k_1, k_2, e, h) \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]\!]\} \\
[\![\texttt{SelectRow}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3)]\!] &= \{\texttt{SelectRow}(k_1, k_2, k_3) \mid k_1 \in [\![\tilde{k}_1]\!], k_2 \in [\![\tilde{k}_2]\!], k_3 \in [\![\tilde{k}_3]\!]\} \\
[\![\texttt{SelectColumn}(\tilde{k}_1, \tilde{k}_2, \tilde{k}_3)]\!] &= \{\texttt{SelectColumn}(k_1, k_2, k_3) \mid k_1 \in [\![\tilde{k}_1]\!], k_2 \in [\![\tilde{k}_2]\!], k_3 \in [\![\tilde{k}_3]\!]\} \\
[\![\{(\sigma_1, c_1), \cdot\cdot, (\sigma_m, c_m)\}]\!] &= \{\texttt{G} \mid \texttt{G} \text{ is a linear function over loop iterators in} \\
&\quad \sigma \text{ such that } \texttt{G}(\sigma_i) = c_i \text{ for all } 1 \le i \le m\} \\
[\![\{(\sigma_1, d_1), \cdot\cdot, (\sigma_m, d_m)\}]\!] &= \{\texttt{H} \mid \texttt{H} \text{ is a Boolean function over loop iterators in} \\
&\quad \sigma \text{ such that } \texttt{H}(\sigma_i) = d_i \text{ for all } 1 \le i \le m\}
\end{aligned}
$$

Figure 5.4: Semantics of angelic programs.

To make our examples easier to read, we write them in pseudocode instead of the syntax described in Figure 5.2. 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 $\texttt{Update}(k_1, k_2, e, h)$ 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 5.1 contains statements of the form $\texttt{Return}(e)$; these are written as $\texttt{Update}(0, 0, \texttt{Return}(e), none)$.

5.2.4 Synthesis Problem

The previous sections formally defined an input problem (Section 5.2.1) and an output solution (Section 5.2.2) for a mathematical procedure (Section 5.2.3). 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 (Section 5.3.2). 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.

5.3 Synthesis

We present an algorithm for the synthesis problem introduced in Section 5.2.4. 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 5.3.1 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 5.3.2 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: Simul-

$$
\begin{aligned}
\texttt{IS}(\texttt{Sequence}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m),\texttt{Sequence}(\tilde{S}'_1,\cdot\cdot,\tilde{S}'_m)) &= \texttt{Sequence}(\texttt{IS}(\tilde{S}_1,\cdot\cdot,\tilde{S}_m),\cdot\cdot,\texttt{IS}(\tilde{S}'_1,\cdot\cdot,\tilde{S}'_m)) \\
\texttt{IS}(\texttt{Cond}(\tilde{b},\tilde{P}_1,\tilde{P}_2),\texttt{Cond}(\tilde{b}',\tilde{P}'_1,\tilde{P}'_2)) &= \{\texttt{Cond}(\texttt{IS}(\tilde{b},\tilde{b}'),\texttt{IS}(\tilde{P}_1,\tilde{P}'_1),\texttt{IS}(\tilde{P}_2,\tilde{P}'_2)), \\
&\quad \texttt{Cond}(\texttt{IS}(\tilde{b},\tilde{b}'),\texttt{IS}(\tilde{P}_1,\tilde{P}'_2),\texttt{IS}(\tilde{P}_2,\tilde{P}'_1))\} \\
\texttt{IS}(\{\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{IS}(\tilde{V}_j,\tilde{V}'_{j'}) \\
\texttt{IS}(\{\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{IS}(\tilde{R}_j,\tilde{R}'_{j'}) \\
\texttt{IS}(\texttt{Update}(\tilde{k}_1,\tilde{k}_2,\tilde{E},h),\texttt{Update}(\tilde{k}'_1,\tilde{k}'_2,\tilde{E}',h)) &= \texttt{Update}(\texttt{IS}(\tilde{k}_1,\tilde{k}'_1),\texttt{IS}(\tilde{k}_2,\tilde{k}'_2),\texttt{IS}(\tilde{E},\tilde{E}'),h) \\
\texttt{IS}(\texttt{Loop}(j,\tilde{b},\tilde{P}),\texttt{Loop}(j,\tilde{b}',\tilde{P}')) &= \texttt{Loop}(j,\texttt{IS}(\tilde{b},\tilde{b}'),\texttt{IS}(\tilde{P},\tilde{P}')) \\
\texttt{IS}(\{\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{IS}(\tilde{e}_j,\tilde{e}'_{j'}) \\
\texttt{IS}(\texttt{F}(\tilde{a}_1,\cdot\cdot,\tilde{a}_m),\texttt{F}(\tilde{a}'_1,\cdot\cdot,\tilde{a}'_m)) &= \texttt{F}(\texttt{IS}(\tilde{a}_1,\tilde{a}'_1),\cdot\cdot,\texttt{IS}(\tilde{a}_m,\tilde{a}'_m)) \\
\texttt{IS}(\texttt{Select}(\tilde{k}_1,\tilde{k}_2),\texttt{Select}(\tilde{k}'_1,\tilde{k}'_2)) &= \texttt{Select}(\texttt{IS}(\tilde{k}_1,\tilde{k}'_1),\texttt{IS}(\tilde{k}_2,\tilde{k}'_2)) \\
\texttt{IS}(\texttt{SelectRow}(\tilde{k}_1,\tilde{k}_2,\tilde{k}_3),\texttt{SelectRow}(\tilde{k}'_1,\tilde{k}'_2,\tilde{k}'_3)) &= \texttt{SelectRow}(\texttt{IS}(\tilde{k}_1,\tilde{k}'_1),\texttt{IS}(\tilde{k}_2,\tilde{k}'_2),\texttt{IS}(\tilde{k}_3,\tilde{k}'_3)) \\
\texttt{IS}(\texttt{SelectCol}(\tilde{k}_1,\tilde{k}_2,\tilde{k}_3),\texttt{SelectCol}(\tilde{k}'_1,\tilde{k}'_2,\tilde{k}'_3)) &= \texttt{SelectCol}(\texttt{IS}(\tilde{k}_1,\tilde{k}'_1),\texttt{IS}(\tilde{k}_2,\tilde{k}'_2),\texttt{IS}(\tilde{k}_3,\tilde{k}'_3)) \\
\texttt{IS}(\tilde{k},\tilde{k}') &= \text{Let } \mathit{temp} := (\tilde{k}\cup\tilde{k}') \text{ in if } (\llbracket \mathit{temp} \rrbracket = \emptyset) \text{ return } \top \text{ else return } \mathit{temp}
\end{aligned}
$$

Figure 5.5: The `Intersect` function, abbreviated here as `IS`. For any case not listed here, Intersect returns $\top$. Additionally, `Intersect` returns $\top$ instead of $\emptyset$.

taneous Division algorithm, described in Figure 5.1. 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 5.3.3. 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 Section 5.3.1, our algorithm learns integer linear expressions 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 5.1, 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 5.1: $T[j, 0] \neq T[j, 1]$ 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 con-

straints. The goal of this process is to take these constraints and generate a program that satisfies 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.

5.3.1 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 5.3. 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 5.4, 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 $n_1 + n_2 + \cdots + n_m$.

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 `G` 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.

This data structure supports an efficient `Intersect` operation shown in Figure 5.5. `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 $\top$.

$$\begin{array}{rcl} \text{Program Template } \mathcal{P} & := & \texttt{Sequence}(\mathcal{S}_1, \mathcal{S}_2, \cdot\cdot, \mathcal{S}_m) \\ \text{Statement Template } \mathcal{S} & := & \texttt{<p>} * W \;\mid\; \texttt{<p> Cond}(*b, \mathcal{P}_1, \mathcal{P}_2) \end{array}$$

Figure 5.6: Syntax of Templates

5.3.2 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 5.6. 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 5.1 is `Cond`$(*b, $`Sequence`$(*W, *W), $`Sequence`$(*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` h in $\mathcal{P}$ by $Val(h)$. 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 $\tilde{W}$.

5.3.3 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 5.7, 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

```
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,h), State(Tr,i)) s.t.  Update(k̃1,k̃2,Ẽ,h)
     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 5.7: Procedure `SynthesizeFromExample`.

$ProgA[0, i, i+1]$, $0 \leq i <$ `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

`AddLoopPrograms`, shown in Figure 5.8, 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

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

Figure 5.8: Procedure `AddLoopPrograms`.

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`

```
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 h in P):
13           Val''[h] := Intersect(Val'[h], Val[h]);
14           if(Val''[h] = ⊤) then fail := true; break;
15         if (not fail) then newResult := newResult ∪ Val'';
16     result = newResult;
17   foreach (Val in result):
18     P := P[Val]; temp_2 := temp_2 ∪ {P};
19 Materialize all integer and Boolean angelic constants in temp_2.
20 return temp_2;
```

Figure 5.9: Procedure `Synthesize`.

(line 33).

5.3.4 Overall synthesis algorithm

Figure 5.9 defines the overall synthesis algorithm, `Synthesize`, which takes as input a set of examples $\{Z_1, \cdot\cdot, Z_m\}$, 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 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`*.*

5.3.5 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.

5.4 Evaluation

In order to evaluate the effectiveness of our data structure and synthesis algorithm, we tested it on a dataset of 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, less than, and less than or equal to. Some were more complex, such as the `LeastPrimeDivisor` function found in GCF: Simultaneous Division, which finds the lowest

prime divisor of all of the inputs.

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 where we tried out different subsets of the set of templates until synthesis succeeded.

5.4.1 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 5.10. 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 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.

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
Division: Repeated Subtraction (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 Continuation: Explicit Addition	1	0	2	[S]	<1	3
Pattern Continuation: Explicit Subtraction	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 5.10: 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 5.6). "fail" indicates that no program was synthesized within 10 minutes.

Bug	Page	L	C	S	Templates	T(s)	ST(s)
Addition	34	1	0	2	[S]	<1	2
Addition	35	1	1	6	[C{2S}{S}],[S]	15	fail
Addition	36	1	1	6	[S],[C{2S}{S}]	<1	61
Addition	37	1	1	8	[C{2S}{2S}],[2S]	<1	38
Subtraction	38	1	1	5	[C{S}{S}],[S]	<1	3
Subtraction	39	1	0	4	[S],[3S]	<1	8
Subtraction	40	1	1	5	[C{S}{S}],[S]	7	48
Subtraction	41	1	2	10	[C{C{S}{S},2S}{S}],[S]	24	fail
Subtraction	42	1	1	7	[C{S}{S}],[3S]	<1	131
Multiplication	44	1	1	6	[C{2S}{S}],[S]	<1	fail
Multiplication	45	0	0	4	[4S]	<1	fail
Division	47	1	1	5	[C{S}{S}],[S]	<1	62
Fraction Reduction	51	0	0	2	[2S]	<1	1
Fraction Reduction	52	1	3	11	[C{S}{C{S}{C{S}{S}}}],[S]	4	32
Fraction Reduction	53	0	1	6	[C{2S}{2S}]	<1	4
Fraction Addition	54	0	0	2	[2S]	<1	1
Fraction Addition	55	1	0	5	[2S],[3S]	<1	fail
Fraction Addition	56	1	0	5	[2S],[3S]	<1	49
Fraction Addition	57	1	0	5	[2S],[3S]	<1	fail
Fraction Subtraction	58	0	3	14	[C{S}{S},C{2S}{C{2S}{2S}}]	<1	fail
Fraction Subtraction	59	1	1	12	[3S],[S,C{3S}{S},2S]	127	fail
Fraction Subtraction	60	1	0	3	[S],[2S]	<1	2
Fraction Subtraction	61	2	1	10	[S],[3S],[S,C{S}{S},S]	41	fail
Fraction Multiplication	63	1	0	6	[S],[5S]	<1	fail
Fraction Multiplication	64	1	0	2	[S]	<1	1
Fraction Division	65	1	0	2	[S]	<1	1
Fraction Division	66	1	0	6	[S],[5S]	<1	2
Decimal Addition	67	0	1	5	[C{2S}{S}]	<1	2

Figure 5.11: Summary of "buggy" benchmarks. Page reports the page number in Ashlock [9] on which the bug was found. See Figure 5.10 for description of other columns. These results show that our algorithm can efficiently learn programs to describe students' errors.

5.4.2 Buggy procedures

Ashlock [9] 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 5.11. 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 bizarre and 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, M_F_1, involved 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, we decided it was too impractical to include as a successful benchmark. Second, some of the 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 of the bugs involved text.

5.5 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 [33, 28]. 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 5.12 shows three motivating transformations from [33] that came 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 5.12,

Figure 2 in [33]. 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 [33]. 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 [33]. 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 5.12: Our system learned a program to compute these three spreadsheet table transformations from Harris and Gulwani [33].

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 [33] is specialized for table layout transformations and cannot synthesize any of our math programs.

5.6 Comparison to SKETCH

We compared our tool to SKETCH [80], 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 [33] reported that SKETCH could not successfully solve their table transformation benchmarks. Therefore we tried SKETCH on all our math benchmarks, as shown in Figures 5.10 and 5.11. 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.

5.7 Discussion and Future Work

This chapter 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.

Chapter 6

CONCLUSION

This thesis presents a new framework for generating instructional scaffolding directly from procedural knowledge. In my thesis statement, I claimed that we can leverage techniques from software engineering and programming languages to create instructional scaffolding for teaching *any* procedural skill that we can encode as a computer program. This approach is less tedious than creating scaffolding by hand, and is much more general than writing algorithms to generate educational content for a specific topic or domain. I will now revisit these claims.

In Chapter 3, I demonstrated that we can use test input generation tools to generate practice problem progressions for any procedural thought process. These tools explore possible execution pathways through a program and generate inputs that cause the program to take these pathways. This allows us to draw a parallel between *code coverage* in software engineering and the *completeness of a practice problem set* in education. This technique can generate large numbers of practice problems in a way that ensures that we can find important practice problem types efficiently. This approach scales to the level of generating a full level progression for *Refraction* (a popular puzzle game played by one million people), or constructing a progression of words one should pronounce in order to learn the entire Thai alphabet. In the case of Refraction, we experimentally confirmed that the generated content was comparable to that created by experts.

In Chapter 4, I demonstrated that we can use ideas from debuggers and integrated development environments (IDEs) to generate step-by-step demonstrations of any procedural skill. By mapping programming language constructs to visual objects in an interface through *interface hooks*, we can call the user's attention to objects when they are important. We can also take advantage of linguistic information that is typically present in lines of code by converting code fragments to short text instructions that tell the learner what to

do. For some domains, such as two-dimensional spreadsheet mathematics, writing a small number of text generation templates and interface hooks enables this system to explain a large number of K-12 math procedures, and explain how to solve any particular practice problem for a given procedure.

In Chapter 5, I demonstrated that we can use programming-by-demonstration techniques to learn misconceptions by synthesizing a program (or a set of programs) that is consistent with the student's output. I described a system that can do this effectively for two-dimensional grid mathematics. This system advanced the state-of-the-art of programming-by-demonstration by enabling synthesis of programs with nested loops and conditionals inside loops. I showed that this system could diagnose 28 misconceptions across nine different procedures that were drawn from a well-known book of misconceptions. Diagnosing these errors by hand is much more challenging. In addition to learning incorrect programs, this system also enables educators to demonstrate procedural skills, making the other applications of my thesis (generation of practice problems and demonstrations) more accessible.

6.1 Future Horizons

The long-term goal of this work is to develop tools that allow every topic to be taught through interactive learning experiences that lead to mastery for every student. I envision the next generation of learning experiences as automatically generated instructional content that adapts to each user on the fly to suit knowledge and preferences, constantly improving its performance as more data is gathered. I believe that algorithmic advances in two areas are required. First, we need more powerful tools for exploring learning pathways and creating content that follows them. Second, we need automatic ways to map educational theory onto generated content in order to focus data-driven optimization on the most critical decisions. More specifically, we should investigate:

Automatic generation of instructional materials for any topic. In this thesis, I examined how we can generate instructional scaffolding for procedural knowledge. However, there are many other types of knowledge, and we need to explore how to generate scaffolding for other models and representations. Many topics in education require conceptual reason-

ing, requiring students to understand *why* and not just *how* or *what*. For example, many researchers have explored conceptual models for fractions [39, 63, 18]. Modeling logical inference (e.g. backwards chaining) can produce materials for teaching critical thinking, such as logic puzzles and geometry proof problems. I believe that automatic exploration of algorithms for natural language understanding, ranging from low-level operations like identifying parts-of-speech to higher level processes like extracting information from text, can help us generate passages, stories, and questions for teaching reading comprehension, foreign language, and word problems in mathematics.

Tailoring the experience for each student. Computer science can improve education through the automatic generation of learning materials that are matched to the learner's skills and preferences. I believe that a combination of techniques, such as programming-by-demonstration, knowledge tracing, and probabilistic methods, will help us build models of the learners knowledge, beliefs, skills, and learning strategies. We can then use these models to generate materials that ensure a smooth progression of difficulty by utilizing the specific combination of concepts the learner has already mastered to facilitate the acquisition of unlearned concepts. We can resolve misconceptions by modeling a student's faulty reasoning and generating demonstrations and tasks that show why that reasoning is incorrect or insufficient. For example, if we detect that a student is using a particular strategy, we can automatically create tasks that are easy or difficult for that particular strategy to keep the learner as engaged as possible.

Data-driven discovery of optimal learning pathways. Large-scale education provides a unique opportunity to answer some of the deepest questions in learning science and HCI, through data collection and experimentation with large numbers of students. For example, if we can determine that many students are using a particular learning strategy, we can generate tasks that are matched to that strategy. Similarly, if we can detect that a certain misconception is likely, we can construct materials to preemptively resolve that particular confusion. Since the design space of interactive learning experiences is typically huge, a reasoned approach is needed to prune this space in order to focus exploration on

the most critically important decisions. Key questions include: Which base concept should we introduce first? How quickly should we introduce new concepts? How frequently should we reinforce previously mastered concepts before they are forgotten? What is the optimal learning pathway for each student?

BIBLIOGRAPHY

[1] Nikki Aduba. JUMP Mathematics in Lambeth: Impact on KS2 National Tests 2009. `http://jumpmath1.org/research_reports`, November 2009.

[2] Vincent Aleven, Bruce M. McLaren, Jonathan Sewall, and Kenneth R. Koedinger. The cognitive tutor authoring tools (CTAT): preliminary evaluation of efficiency gains. In *Proceedings of the 8th international conference on Intelligent Tutoring Systems*, ITS'06, pages 61–70, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Vincent Aleven, Bruce M. Mclaren, Jonathan Sewall, and Kenneth R. Koedinger. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Int. J. Artif. Intell. Ed.*, 19(2):105–154, April 2009.

[4] Erik Andersen, Sumit Gulwani, and Zoran Popović. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI '13: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2013. ACM.

[5] Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, Seth Cooper, and Zoran Popović. On the harmfulness of secondary game objectives. In *FDG '11: Proceedings of the Sixth International Conference on the Foundations of Digital Games*, New York, NY, USA, 2011. ACM.

[6] Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, and Zoran Popović. Placing a value on aesthetics in online casual games. In *CHI '11: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2011. ACM.

[7] Erik Andersen, Eleanor O'Rourke, Yun-En Liu, Richard Snider, Jeff Lowdermilk, David Truong, Seth Cooper, and Zoran Popović. The impact of tutorials on games of varying complexity. In *CHI '12: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2012. ACM.

[8] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995.

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

[10] Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. Docwizards: A system for authoring follow-me documentation wizards. In *UIST '05 Proceedings of the 18th annual ACM symposium on User interface software and technology*, New York, NY, USA, 2005. ACM.

[11] Stephen B. Blessing. A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. *International Journal of Artificial Intelligence in Education (IJAIED)*, 8:233–261, 1997. Part I of the Special Issue on Authoring Systems for Intelligent Tutoring Systems (editors: Tom Murray and Stephen Blessing).

[12] Eric Butler, Adam M. Smith, Yun-En Liu, and Zoran Popovic. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 377–386, New York, NY, USA, 2013. ACM.

[13] Pavol Cerny, Sumit Gulwani, Tom Henzinger, Arjun Radhakrishna, and Damien Zufferey. Specification, verification and synthesis for automata problems. Technical report, 2012.

[14] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. MixT: automatic generation of step-by-step mixed media tutorials. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST '12, pages 93–102, New York, NY, USA, 2012. ACM.

[15] Pei-Yu Chi, Joyce Liu, Jason Linder, Mira Dontcheva, Wilmot Li, and Bjoern Hartmann. Democut: Generating concise instructional videos for physical demonstrations. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 141–150, New York, NY, USA, 2013. ACM.

[16] Albert T. Corbett and John R. Anderson. Knowledge tracing: Modelling the acquisition of procedural knowledge. *User Model. User-Adapt. Interact.*, 4(4):253–278, 1995.

[17] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper & Row Publishers, Inc., New York, NY, USA, 1990.

[18] V. V. Davydov and Z. Tsvetkovich. On the objective origin of the concept of fractions. *Focus on Learning Problems in Mathematics*, 13:13–83, 1991.

[19] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[20] Tao Dong, Mira Dontcheva, Diana Joseph, Karrie Karahalios, Mark Newman, and Mark Ackerman. Discovery-based games for learning software. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 2083–2086, New York, NY, USA, 2012. ACM.

[21] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, New York, NY, USA, 2010. ACM.

[22] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

[23] Krista D. Glazewski and Peggy A. Ertmer. Scaffolding disciplined inquiry in problem-based learning environments. *International Journal of Learning*, 12(6):297–306, 2005.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[25] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH 2009*, New York, NY, USA, 2009. ACM.

[26] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. A survey of software learnability: Metrics, methodologies and guidelines. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, New York, NY, USA, 2009. ACM.

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

[28] Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 2012.

[29] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[30] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61, 2011.

[31] Ankit Gupta, Dieter Fox, Brian Curless, and Michael Cohen. DuploTrack: A Reatime System for Authoring and Guiding Duplo Model Assembly. In *Proceedings of the 25th annual ACM symposium adjunct on User interface software and technology*, New York, NY, USA, 2012. ACM.

[32] Erik Harpstead, Brad A. Myers, and Vincent Aleven. In search of learning: Facilitating data analysis in educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 79–88, New York, NY, USA, 2013. ACM.

[33] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[34] Katherine Isbister, Mary Flanagan, and Chelsea Hash. Designing games for learning: insights from conversations with designers. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2041–2044, New York, NY, USA, 2010. ACM.

[35] S. L. Jackson, S. J. Stratford, J. Krajcik, and E. Soloway. Making dynamic modeling accessible to pre-college science students. *Interactive Learning Environments*, 4:233–257, 1996.

[36] Neven Jurkovic. Diagnosing and correcting student's misconceptions in an educational computer algebra system. In *ISSAC*, pages 195–200, 2001.

[37] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[38] Caitlin Kelleher and Randy Pausch. Stencils-based tutorials: Design and evaluation. In *CHI '05 Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2005. ACM.

[39] T. E. Kieren. Personal knowledge of rational numbers: Its intuitive and formal development. In J. Hiebert and M. Behr, editors, *Number concepts and operations in the middle grades*, pages 162–181, Reston, 1988. National Council of Teachers of Mathematics.

[40] Kenneth R. Koedinger, John R. Anderson, William H. Hadley, and Mary A. Mark. Intelligent Tutoring Goes to School in the Big City. *Proceedings of the 7th World Conference on AIED*, 1995.

[41] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

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

[43] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI*, 2004.

[44] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.

[45] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43, 2003.

[46] Jean Lave and Etienne Wenger. *Situated learning: Legitimate peripheral participation.* Cambridge University Press, Cambridge, England, 1991.

[47] Nan Li, William W. Cohen, and Kenneth R. Koedinger. Problem order implications for learning transfer. In *ITS*, pages 185–194, 2012.

[48] Nan Li, William W. Cohen, Kenneth R. Koedinger, and Noboru Matsuda. A machine learning approach for automatic student model discovery. In *EDM*, pages 31–40, 2011.

[49] Conor Linehan, Ben Kirman, Shaun Lawson, and Gail Chan. Practical, appropriate, empirically-validated guidelines for designing educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1979–1988, New York, NY, USA, 2011. ACM.

[50] Jonathan Liu. Dragonbox: Algebra beats angry birds. *Wired*, June 2012.

[51] Derek Lomas, Kishan Patel, Jodi L. Forlizzi, and Kenneth R. Koedinger. Optimizing challenge in an educational game using large-scale design experiments. In *CHI*, 2013.

[52] Noboru Matsuda, William W. Cohen, and Kenneth R. Koedinger. Building cognitive tutors with programming by demonstration. In *In S. Kramer & B. Pfahringer (Eds.), Technical report: TUM-I0510 (Proceedings of the International Conference on Inductive Logic Programming) (pp. 41-46): Institut fur Informatik, Technische Universitat Munchen*, 2005.

[53] Noboru Matsuda, William W. Cohen, Jonathan Sewall, and Kenneth R. Koedinger. Applying machine learning to cognitive modeling for cognitive tutors. Technical report, School of Computer Science, Carnegie Mellon University, 2006.

[54] Noboru Matsuda, William W. Cohen, Jonathan Sewall, Gustavo Lacerda, and Kenneth R. Koedinger. Predicting students' performance with simstudent: Learning cognitive skills from observation. In *AIED*, pages 467–476, 2007.

[55] Noboru Matsuda, Andrew Lee, William W Cohen, and Kenneth R Koedinger. A computational model of how learner errors arise from weak prior knowledge. *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 1288–1293, 2009.

[56] David McArthur, Cathy Stasz, John Hotta, Orli Peter, and Christopher Burdorf. Skill-oriented task sequencing in an intelligent tutor for basic algebra. *Instructional Science*, 7(4):281–307, 1988.

[57] Microsoft. Math Worksheet Generator. `http://www.educationlabs.com/projects/MathWorksheetGenerator/Pages/default.aspx`.

[58] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2001.

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

[60] Antonija Mitrovic and et al. Evaluation of a constraint-based tutor for a database language, 1999.

[61] Tom Murray. Authoring intelligent tutoring systems: an analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, (10):98–129, 1999.

[62] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[63] D. Niemi. Assessing conceptual understanding in mathematics: Representations, problem solutions, justifications, and explanations. *Journal of Educational Research*, 89(6):351–363, July 1996.

[64] Michael Pardowitz, Bernhard Glaser, and Rüdiger Dillmann. Learning repetitive robot programs from demonstrations using version space algebra. In *International Conference on Robotics and Applications*, 2007.

[65] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 135–144, New York, NY, USA, 2011. ACM.

[66] N Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[67] Vidya Ramesh, Charlie Hsu, Maneesh Agrawala, and Björn Hartmann. Showmehow: Translating user interface instructions between applications. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 127–134, New York, NY, USA, 2011. ACM.

[68] C. M. Reigeluth and F. S. Stein. The elaboration theory of instruction. In *Instructional Design Theories and Models: An Overview of their Current States*, Hillsdale, NJ, 1983. Lawrence Erlbaum.

[69] John Rieman. A field study of exploratory learning strategies. *ACM Trans. Comput.-Hum. Interact.*, 3(3):189–218, September 1996.

[70] R. Keith Sawyer, editor. *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press, 2006.

[71] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[72] Robert S. Siegler and Geetha B. Ramani. Playing linear numerical board games promotes low-income children's numerical development. *Developmental Science*, 11(5):655–661, 2008.

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

[74] Rohit Singh, Sumit Gulwani, and Sriram Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[75] D. H. Sleeman. A rule-based task generation system. In *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2*, IJCAI'81, pages 882–887, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[76] Adam Smith, Eric Butler, and Zoran Popović. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, 2013.

[77] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *FDG '12: Proceedings of the Seventh International Conference on the Foundations of Digital Games*, New York, NY, USA, 2012. ACM.

[78] Adam M Smith and Michael Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 273–280. IEEE, 2010.

[79] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182, New York, NY, USA, 2009. ACM.

[80] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[81] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.

[82] Saurabh Srivastava, Sumit Gulwani, and Jeff Foster. From program verification to program synthesis. In *POPL*, 2010.

[83] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

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

[85] Kurt VanLehn. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16:227–265, 2006.

[86] Kurt VanLehn. The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. *Educational Psychologist*, 46(4):197–221, October 2011.

[87] Kurt Vanlehn, Collin Lynch, Kay Schulze, Joel A. Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. The andes physics tutoring system: five years of evaluations. In *In Proceedings of the 12th international conference on Artificial Intelligence in Education*, pages 678–685. IOS Press, 2005.

[88] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, November 1980 / 1930.

[89] Wikipedia. Code coverage. `http://en.wikipedia.org/wiki/Code_coverage`.

[90] Wikipedia. N-gram models. `http://en.wikipedia.org/wiki/N-gram`.

[91] D. Wood, J. Bruner, and G. Ross. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, 17:89–100, 1976.