©Copyright 2019
Eric David Butler

Automatic Generation of Procedural Knowledge Using Program Synthesis

Eric David Butler

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

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Emina Torlak, Chair

Zoran Popović, Chair

Steven Tanimoto

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Automatic Generation of Procedural Knowledge Using Program Synthesis

Eric David Butler

Co-Chairs of the Supervisory Committee:
Assistant Professor Emina Torlak
Paul G. Allen School of Computer Science & Engineering

Professor Zoran Popović
Paul G. Allen School of Computer Science & Engineering

Computer-aided tools have revolutionized the way people approach design problems, such as Computer-Aided Design for industrial design. Computers help designers by taking care of rote tasks and computation, freeing designers' time and brainpower to focus on the challenging aspects of problems that cannot be automated. This thesis focuses on using computers to aid in design tasks involving models of human expertise and learning in problem domains such as high school mathematics or puzzles.

Educational technology that models the problem-solving process of a learning domain—called the *domain model*—powers many promising and proven-effective applications such as generating problems, providing student feedback, or estimating student understanding. Creating domain models is an integral part of such applications and is a challenging and time-consuming process, requiring expertise in both the learning domain and artificial intelligence. Domain models also have applications in fields such as game design: for example, understanding how puzzles and problems must be solved enables automatic game generation or analysis. All of these applications require first completing the challenging design task of creating a domain model.

To aid in this process, we turn to computers, and in particular formal methods, to help automatically learn domain models. Unlike typical machine learning tasks, the primary goal of this endeavor is not to solve domain problems automatically, but to get an *interpretable* description of *how* to solve problems. To support this goal, we use program synthesis. For the domains on which we focus, the domain model takes the form of a set of procedural rules for problem solving, such as the rules used to solve algebra equations. We model procedural knowledge as programs in a domain-specific language (DSL) of rules. Then we use program synthesis to automatically generate these programs. Our approach of using program synthesis for automated rule learning turns the typical use of programming languages on its head; rather than using a programming language as a vehicle for a human to give formal instruction to a computer, we're using a programming language as a vehicle for a computer to give formal instruction to humans.

In this thesis, we present a framework, RuleSy, for automatically learning domain models, instantiated for two separate domains. We present results showing that the rules generated by RuleSy are comparable to or better than human-designed rules. We additionally present two novel applications built on top of domain models: first, a mixed-initiative interface for generating puzzle progressions in an educational game, and second, an algorithm for automatically generating entire game progressions. Lastly, we present results from a case study in which participants design the DSL for use in RuleSy. We describe the challenges and difficulties participants encounter in this process, suggesting future avenues of research to make creating program synthesis tools such as RuleSy easier and accessible to a broad range of programmers.

TABLE OF CONTENTS

Page

List of Figures iv

List of Tables vii

Chapter 1: Introduction 1
1.1 Using Formal Methods for Automatic Domain Model Learning 2
1.2 A Mixed-Initiative Approach to Domain Model Design 4
1.3 Applications of Formal Domain Models 6
1.4 Structure of This Thesis 7

Chapter 2: A Mixed-Initiative Tool for Designing Level Progressions in Games . . 8
2.1 Background 8
2.2 The Game *Refraction* 11
2.3 Current Practices 13
2.4 Progression Design Tools 17
2.5 Application to *Refraction* 19
2.6 System Description 23
2.7 Evaluation 27
2.8 Discussion 30
2.9 Related Work 35
2.10 Conclusion 36
2.11 Acknowledgments 37

Chapter 3: Automatic Game Progression Design through Analysis of Solution Features 39
3.1 Background 39
3.2 System Overview 41

3.3 Extracting Solution Features . . . 42
3.4 Automatically Generating Levels . . . 45
3.5 Creating a Progression . . . 46
3.6 Evaluation . . . 51
3.7 Related Work . . . 56
3.8 Conclusion . . . 60
3.9 Acknowledgements . . . 61

Chapter 4: A Framework for Automatically Learning Domain Models . . . 62
4.1 Domain Models by Example: Propositional Logic Proofs . . . 62
4.2 Mathematical Preliminaries . . . 66
4.3 Procedural Knowledge as a Set of Condition-Action Programs . . . 67
4.4 Synthesizing Procedural Knowledge as Programs . . . 73
4.5 Conclusion . . . 77

Chapter 5: Learning Domain Models for Algebra and Tree Rewrite Domains . . . 78
5.1 Background . . . 78
5.2 Overview . . . 82
5.3 A Domain-Specific Language for Tree Rewrite Rules . . . 87
5.4 Characterizing Sound and Useful Rules . . . 98
5.5 Rule Mining, Synthesis, and Optimization . . . 101
5.6 Generalizing Beyond Algebra . . . 116
5.7 Evaluation . . . 118
5.8 Related Work . . . 123
5.9 Conclusion . . . 124
5.10 Appendix: Proofs of Theorems . . . 124
5.11 Appendix: Axioms . . . 133

Chapter 6: Learning Domain Models for Puzzle Games . . . 140
6.1 Background . . . 140
6.2 Overview . . . 145
6.3 The Mechanics of Nonograms . . . 149
6.4 A Domain-Specific Language for Nonograms Rules . . . 152
6.5 Characterizing Sound and Useful Rules . . . 164

6.6 Rule Mining, Synthesis, and Optimization 167
6.7 Evaluation 179
6.8 Related Work 183
6.9 Conclusion 184
6.10 Appendix: Proofs of Theorems 185
6.11 Appendix: Control Rules 188

Chapter 7: A Case Study on Designing Programming Languages for Synthesis Applications 192
7.1 Background 193
7.2 System and Design Task 195
7.3 Methodology 203
7.4 Results 205
7.5 Discussion 208
7.6 Limitations 210
7.7 Related Work 210
7.8 Conclusion 212

Chapter 8: Conclusion 213

Bibliography 217

LIST OF FIGURES

Figure Number Page

2.1 A level of *Refraction*, a free educational math game for which our tool was targeted. 12
2.2 The ideal workflow for our system (a waterfall-like pattern). 18
2.3 Screenshots of two of the three browser-based editing environments of our tool, each for editing the game at a different scale. 20
2.4 The pieces of our system. 24
2.5 View of the two types of constraint editors in our implementation, which allow manipulation of the various constraint parameters. 25
3.1 A procedural algorithm for subtraction, with labels at the key points. 43
3.2 Three example levels for *Refraction*, to illustrate solution features and graphlets. 44
3.3 Comparison of our automatically-generated progression with the expert-generated progression from *Refraction*. 52
3.4 Visualization of the first 61 levels in the two progressions in our experiment, the human-authored (blue) and automatically genreated (orange). 54
4.1 An example solution of a semantic proof problem. The $\perp$ symbol is used to notate that we have arrived at a contradiction (literally, the interpretation satisfies falsehood). 63
4.2 A semantic proof problem that requires branching multiple times. 65
4.3 Solution to the same problem as Figure 4.2, but using modus ponens. 66
4.4 An illustration of the three axioms that comprise the modus ponens macro. . 66
4.5 An example state where the modus ponens rules applies in multiple ways, each on different parts of the state. 69
4.6 The modus ponens rule, represented in our DSL for rules. 70
4.7 The schema for the semantics of the RuleSy meta language. 71
4.8 The shared semantics of the RuleSy meta language. 72
4.9 A general solving algorithm for problems. 72

4.10 The system architecture of the RuleSy framework, consisting of three phases: specification mining, program synthesis, and domain model optimization. . . 74

5.1 The inputs to RuleSy for the toy algebra domain. 84

5.2 A sample plan (b) mined from the shortest solutions (a) to the toy algebra problems. The plan specifies a tactic for canceling opposite constants (c). . . 85

5.3 Sample tactics synthesized for the toy plans (e.g., Figure 5.2b). 86

5.4 Optimal domain models for toy algebra (Figures 5.1 and 5.3). 86

5.5 Syntax for states in the algebra DSL, which we call *terms*. 87

5.6 Syntax for the algebra DSL. 92

5.7 Types for the algebra DSL's inputs/outputs and semantic functions. 93

5.8 Semantics for the tree-rewrite DSL. 94

5.9 Algebra-specific semantics for the DSL. 95

5.10 The tactic **IB** for combining like terms combines factoring (**I**) and constant folding (**B** in Figure 5.1a), but no interpretation of $\mathbf{I} \circ \mathbf{B}$ captures its behavior. 102

5.11 FindSpecs takes as input a set of example problems T and axioms $\mathcal{A}$, and produces a set of plans $\mathcal{S}$ for composing the axioms into tactics. 106

5.12 FindRules takes as input a bound $\bar{k}$ on program size and an execution plan S that replays a path from *src* to *tgt*. 110

5.13 Optimize takes as input a set of terms T, axioms $\mathcal{A}$ for reducing T, tactics $\mathcal{T}$ synthesized from $\mathcal{A}$ and T using FindRules and FindSpecs, and an objective function f. 114

5.14 Syntax for terms in the semantic proof domain. 117

5.15 The semantic functions that change for the semantic proof domain. 118

5.16 A custom algebra tactic discovered by RuleSy (variable names were chosen by the author for presentation). 121

6.1 An example of a Nonograms puzzle, with the start state on the left and completed puzzle on the right. 142

6.2 The *big hint* rule for one (a) and many (b) hints. 146

6.3 Example of the inputs and outputs for specification mining in our toy problem.147

6.4 Basic version of the big hint rule. 148

6.5 Examples of line segments. 151

6.6 General version of the big hint rule, which uses an `arbitrary` pattern instead of a `singleton` pattern. 155

6.7 Syntax for the Nonograms DSL. 156

6.8 Types for the Nonograms DSL's inputs/outputs and semantic functions. . . . 157
6.9 Semantics for patterns in the Nonograms DSL, ... 158
6.10 Semantics for the Nonograms DSL. 159
6.11 Variant of the basic big hint rule but using arithmetic constructs instead of geometric ones. 160
6.12 The output for Example 6.12. 163
6.13 FindSpecs takes as input a training set T and a verification set E. 169
6.14 A more general yet more complex version of the big hint rule than the one from Figure 6.6. 172
6.15 An example transition to which the basic big hint rule (Figure 6.4) does not apply but the more complex one (Figure 6.14) does. 173
6.16 An example line state whose most refined pattern is described in Example 6.24.175
6.17 FindRules takes as input a specification $\langle S, src, tgt \rangle$ and a bound $\bar{k}$ on program size. 176
6.18 An example of state decomposition. Because hints are ordered, if we know where one hint lies (in this case, hint 3), then we can consider sub-lines in isolation. 178
6.19 An example of a control rule that our system recovers exactly, annotated with comments. 181
6.20 An example top-10 rule learned by our system that is *not* in the control set, annotated with comments. 182
7.1 A screenshot of the DSL editing tool. 201
7.2 Semantics for patterns added by participants, grouped by pattern type. . . . 207

LIST OF TABLES

Table Number Page

2.1 A sample of the properties we model for levels in *Refraction*. 19

3.1 Example subtraction problems used in elementary-school mathematics and their corresponding traces. 43

4.1 The axioms that define the domain of semantic proofs for propositional logic. 64

4.2 The modus ponens proof rule, which, while unnecessary, can greatly simplify proofs. 64

5.1 Example problems (a) and axioms (b) for the algebra case study. 120

5.2 A textbook progression and the corresponding optimal domain models found by RuleSy, using 3 settings of α 122

7.1 Table listing the various features that each participant designed, not including the two (`hint/basic` and `cell`) given at the start of the task. 206

ACKNOWLEDGMENTS

This dissertation is the result of many years of work made possible only due to the support and guidance of many people. I am immensely grateful to those that have helped me.

First, I am deeply grateful to my advisors, Emina Torlak and Zoran Popović, for their patience and guidance over many years. Emina taught me how to be an good researcher, from asking the right questions to effective problem solving and quality writing. Zoran taught me how to focus on important research problems and communicate my work effectively. I would like to thank my committee members: Steven Tanimoto for his detailed and valuable feedback on this dissertation and Andrew Ko for his help in study design.

I would particularly like thank to Adam M. Smith for his mentorship and guidance, especially as I was finding my footing as a researcher. It is no exaggeration to say that making it through graduate school would have been impossible without his help and support.

I am hugely grateful to my colleagues and coauthors—Kristin Siu, Erik Andersen, Rahul Banerjee, Aaron Bauer, Gregory Nelson, Yun-En Liu, Eleanor O'Rourke, and Alexander Zook—for working with me. It was a privilege to be able to spend so much of my research time collaborating with others; that collaboration made both me and my research better. I would also like to thank my friends and labmates Yvonne Chen, Dun-Yu Hsiao, Alexander Jaffe, Seongjae Lee, Travis Mandel, Oleksandr Polozov, Zuoming Shi, Armando Díaz Tolentino, and Kathleen Tuite for creating a vibrant research environment. Much of my work was directly inspired by theirs, and their friendship and support have helped me immensely.

I very much want to thank the staff and students of the Center for Game Science: Ourania Abell, Nova Barlow, Colin Bayer, Brian Britigan, Matthew Burns, Craig Conner, Seth Cooper, Mai Dang, Happy Dong, Dmitri Danilov, Robert Duisberg, Kate Fisher, Jeff Flat-

ten, Ric Gray, Mostafa Hedayat, Barbara Krug, Saira Mortier, Christian Lee, Marianne Lee, Kathleen Moore, Tim Pavlik, Erin Peach, Rich Snider, Cullen Su, Roy Szetzo, Jennifer Vogel, Aaron Whiting, and Yanko Yankov. By building the games on which this research was developed and through ongoing support of my research, they made this dissertation possible.

I would like to thank my friends and coworkers Duncan Boehle, Katie Chironis Grossman, Connor Fallon, and Wesley Martin, Valeria Reznitskaya for keeping my ideas grounded in reality. Working with them in parallel to my dissertation has helped me ask better research questions and directly inspired parts of this work.

Finally, I'd like to thank my family, particularly my mother Anna Butler and father Thomas Butler, for supporting me throughout my life and education.

DEDICATION

This dissertation is dedicated to Kristin,

who convinced me to embark on the graduate school journey with her.

We made it.

Chapter 1

INTRODUCTION

One of the driving applications of technology has been augmenting what humans can do and how they approach solving problems. Specifically within design fields, computer-powered tools (e.g., Computer-Aided Design for architecture/industrial design, Photoshop/Illustrator for visual art) have revolutionized the way people approach design problems, changed what is possible, and expanded how many people can fruitfully work on such problems. Computers can help designers by taking care of rote tasks and computation, freeing designers' time and brainpower to focus on the challenging aspects of problems that cannot be automated. Beyond serving simply as a tool, computers can actually contribute the design, working with the designer to solve problems. This thesis focuses on using computers to aid in the particular class of design tasks involving models of human expertise and learning in problem domains such as high school mathematics or puzzles.

A motivating example of this class of tasks is that of building educational software for a domain such as solving linear algebraic equations. A key challenge in the design of educational applications is modeling the operational knowledge that represents the expertise and skill of human problem solving. Designing and building this model—called the *domain model*—is necessary and challenging for any educational application. For example, one needs to decide what should be taught in a class curricula. However, a computer-based tutor needs a level of formality and detail in the domain model that makes this design task even more difficult. Solving the domain-model design task lies at the heart of many educational technology applications and in broader applications such as game design.

Creating domain models by hand takes a tremendous amount of effort and resources [94]. The ability to effectively and efficiently create domain models opens up a path toward

revolutionary learning technologies such as generating targeted problems that adapt to a learner's skill level [2, 25, 3], automatically detecting misconceptions [43], estimating student skill [91, 108, 46], or providing adaptive scaffolding for learners. In this thesis, we focus on the underlying design challenge of automatically learning the domain models on which these applications are built. The knowledge and expertise we are interested in modeling has many components, but we focus on *procedural knowledge*: how to complete tasks and solve problems.

To illustrate the difficulty of model authoring, consider creating a domain model for K-12 algebra. Suppose that our model includes the basic rules, or *axioms*, for solving algebra problems: e.g., factoring, $c_0x + c_1x \to (c_0 + c_1)x$, where c_0 and c_1 are constants and x is any term; and constant folding, $c_0 + c_1 \to c_2$, where c_0 and c_1 are constants and c_2 is their sum. Should this model also include the rule for combining like terms, $c_0x + c_1x \to c_2x$, which composes factoring and constant folding? While such compound rules are redundant with respect to the axioms, standard educational curricula (e.g., [31]) include them to enable efficient problem solving with fewer steps, leading to less cognitive load [129] during learning and problem solving. But there is a limit to how many rules students can be taught, so the optimal set of axioms and tactics depends on the desired trade-off between maximizing solving efficiency and minimizing the memorization burden. Aside from making these kinds of design decisions, the designer has to also create formal representations of all of these rules, ensuring those representations are logically sound and suitably useful at solving desired problems. This thesis explores using formal methods—program synthesis in particular—to overcome these challenges. We present a framework, RuleSy, for automatically learning domain models by generating formal, sound, and useful rules for problem solving.

1.1 Using Formal Methods for Automatic Domain Model Learning

Automatically learning domain models presents an atypical challenge for standard machine learning approaches: the goal of this endeavor is not to build a computer system capable of solving a given problem per se. It is often straightforward to create a computer algorithm

to solve problems in such learning domains. Automated rule learning is a broad and well-studied problem [74, 114], but the research traditionally focused on computers learning for the sake of performing the tasks themselves. Instead, we want to learn an *interpretable* description of *how* to solve a given problem. The results are not meant to be executed by a computer but rather interpreted by educators, taught to students, and put into tools. The procedural rules that are best for students are likely very dissimilar to the best computer algorithm. We strongly favor simple explanations, because simpler rules are more likely to have lower cognitive load and be more effective for learning. Furthermore, strong guarantees of soundness are crucial for an educational setting. We do not want rules that are valid only with high probability; we want to find rules that are guaranteed to be logically sound.

As the technological foundation for our domain model-learning framework, we turn to formal methods based on automated SAT/SMT solvers [93]. These tools can provide strong formal guarantees for discrete problems of modest size and can extrapolate from limited data. This fits precisely with our design task: when creating a domain model for, say, algebra, we want to be fully certain that what we are creating is logically sound and follows the laws of algebra. At the same time, because the domain is eventually intended for human learning, the scope of the problems and the techniques that constitute the domain model are bounded by the (very small, compared to a computer) constraints of human cognition. This harmonious alignment is why we will approach this as a formal methods problem rather than, say, statistical machine learning. Statistical ML is designed for a wholly different class of problems. Approaches such as statistical ML do not provide the required soundness guarantees and require different kinds of data as input.

We frame the challenge of learning domain models as a *program synthesis* problem, where rules and strategies are represented in a domain-specific programming language (DSL). Program synthesis is the task of discovering an executable program (in some specified programming language) that realizes user intent expressed in the form of some specification [48]. In addition to myriad research and industrial applications, program synthesis has been successfully applied to educational applications such as feedback generation [113], solution gener-

ation [51], and problem generation [3]. Our approach will be to transform the problem of generating solving strategies into one of synthesizing programs. The DSL used to represent programs will come as input to the system. That is, the humum using our system designs the DSL for representing rules and strategies, and the system writes programs in that DSL.

Our approach of using program synthesis for automated rule learning turns the typical use of programming languages on its head; rather than using a programming language as a vehicle for a human to give formal instruction to a computer, we're using a programming language as a vehicle for a computer to give formal instruction to humans. The limits and strengths of program synthesis align well with the requirements for our design task. We require logically sound rules for problem solving, which program synthesis can guarantee. A primary limitation of program synthesis is scalability, but fortuitously, the limits of human cognition incentivize finding very small programs anyway. The primary technical contributions of this thesis are the design and implementation of a domain-specific language and a framework (built on program synthesis) for synthesizing rules. We adapted this framework, RuleSy, to two educational domains: solving linear algebraic equations and solving Nonograms puzzles. For the purposes of RuleSy, a *domain model* means a set of programs in some domain-specific language of solving strategies. Though this is just one possible way to represent domain models, it is compatible with the way tools such as cognitive tutors represent their domain models and allows us to model multiple distinct problem domains.

1.2 A Mixed-Initiative Approach to Domain Model Design

Research has investigated using machine learning to ease the domain model authoring process for computer tutors in particular [67, 90, 77, 61, 111], but these efforts focus primarily on programming-by-example or by demonstration, in which a tutor designer demonstrates known rules for a system to learn the formal representation. In contrast, this thesis targets a broader problem in which the designer *doesn't* know the particular rules they want to use in the domain model. The goal of RuleSy is supporting domain-model design in a mixed-initiative manner; a hypothetical user works collaboratively with the system to design rules

that they may not know a priori.

The potential benefits of a system like RuleSy are multiple. First, the computer can suggest rules and strategies for the model that the designer was not considering, ensuring a wide variety of problems can be solved. But even in scenarios where the designer has a good idea about the particular rules they want to use, the computer can ensure such rules are logically sound.[1]

We present results from implementing RuleSy for two different educational domains, finding that RuleSy is able to (a) produce logically sound rules, (b) find known rules actively used in existing curricula, and (c) discover novel rules that can better solve problems than currently documented rules.

RuleSy does this with a mixed-initiative interface, where both the (domain model-designing) user and RuleSy contribute to the design. The user of the framework provides a description of the learning domain and a domain-specific programming language for representing procedural rules in that domain, along with training and testing problems. RuleSy, using these inputs, finds rules in the given language that can solve the training problems and generalize to others while remaining sound with respect to the domain description. Looking at how these synthesized rules perform on the testing examples, the user can adjust the inputs, particularly the DSL, in order to find more desirable rules.

The technical contributions of the thesis focus on realizing a framework capable of finding rules. However, there remains an important question about how users could fruitfully interact with such a system. How challenging is the language design process and what could we do to make it easier? This question is broader than just for the RuleSy application; many program synthesis applications—such as programming-by-example systems—rely heavily on the design of the domain-specific language for success. However, very little has been written about how to effectively design such languages. In this thesis, we explore a preliminary

[1]Ensuring that hand-authored rules are actually correct is not just a hypothetical problem: during development of RuleSy for Nonograms (Chapter 6), two rules in the evaluation were discovered to be logically faulty. These turned out to be two of the hand-authored rules used for comparison. The authors could not even write a dozen rules without making multiple logical mistakes, which RuleSy caught!

user study of RuleSy in which participants attempt to design (part of) the language used in the Nonograms application of RuleSy. Our focus of this study is evaluating whether potential designers could successfully design DSLs for use with RuleSy and understanding what challenges they encountered in doing so. More generally, DSL design for program synthesis is a crucial yet understudied part of building successful program synthesis systems. Based on our study, we suggest avenues of future work on this problem.

1.3 Applications of Formal Domain Models

While designing effective domain models is an important component is existing applications ranging from traditional class curricula and educational technology to game design, access to detailed formal models of human skill enables novel applications not otherwise feasible. For the first part of this thesis, we describe two such potential applications, illustrating tools for designing educational games that rely on domain models to function. Both of these tools are focused on designing a game's *progression*—the sequence of problems/puzzles presented to the player. Choosing an appropriate progression is a critical challenge for games as well as education. Which skills are taught in which order and how they are practiced has dramatic effects on learning.

Designing a progression requires balancing design considerations at multiple scales: individual problems or puzzles require iteration and effort to produce, yet broad-scale constraints such as the order concepts are introduced and how they are combined must be respected. Traditionally for game design, these considerations must be tracked by hand, typically informally. The first tool we present is a mixed-initiative interface for designing game progressions that explicitly models these design constraints and various scales. This allows for both analysis and generation of puzzles and progressions, enabling a designer to work with the computer to create a complete game. We present a proof-of-concept implementation using the math education game *Refraction* and a set of case studies showing novel design interactions such a tool enables. Modeling the progression necessarily involves modeling the skills and strategies used to solve puzzles, which is precisely the domain model RuleSy aims to generate.

Beyond empowering designers creating progressions manually (or in collaboration with a computer), a long-term goal of game design research is to achieve end-to-end automation of much of the design process. As our second application, we present a fully generated version of *Refraction*. This version, dubbed *Infinite Refraction*, has an adaptive progression based on player performance in prior puzzles, choosing which skills should be introduced next and then choosing the most appropriate puzzle from a large bank of generated puzzles. We present results from an in-the-wild study comparing *Infinite Refraction* to the original, hand-designed game, showing the games comparable on a key engagement metric. As before, such an application relies heavily on a useful domain model, this time substantially more complex than the progression design tool. In both cases, the domain model was developed by hand and much more limited than would be necessary in real-world applications.

In both of these applications, the way a domain model is represented differs from that of RuleSy; they are both substantially simpler than RuleSy's representation. We present these applications specifically to motivate the need for effective tools for domain model design; the reason the domain models in these applications are quite simple is that it was too challenging for us to design more complicated ones by hand.

1.4 Structure of This Thesis

The structure of the remainder of the thesis is as follows. Chapters 2 and 3 describe two novel applications that require domain models to function, motivating the need for a tool to automatically generate them. Chapter 4 introduces the RuleSy framework to do this automatic generation of domain models. Chapters 5 and 6 describe our two instantiations of RuleSy, for the domains of tree-rewrite rules (specifically algebra) and Nonograms, respectively. Then, Chapter 7 describes a user study investigating the challenges of designing languages for use with RuleSy, before the concluding chapter describing future work.

Chapter 2

A MIXED-INITIATIVE TOOL FOR DESIGNING LEVEL PROGRESSIONS IN GAMES

In this chapter, we present work on a novel user-interface and interaction paradigm for designing progressions for games, with an example application of the educational math game *Refraction*. The tool is an editing environment for a sequence of puzzles in which the system can work with the designer (hence, mixed-initiative) to generate content. Such an application is built on top of a domain model of skills used in the game. In our proof-of-concept implementation, we used a very simplistic model, but any practical application of this technology would require a much more sophisticted understanding of potential player skills and strategies, motivating the RuleSy framework presented in later chapters. The following Chapter 3 presents another example application for fully automatic game generation that is also built on domain models. Portions of the chapter are reused from or based on prior technical publications [27].

2.1 Background

The design of interactive experiences that consist of a sequence of episodes building in complexity is an intricate, multi-dimensional design problem that often takes experts a long time to complete. This is particularly apparent in game design. Designing just a single gameplay element (e.g., puzzles, challenges, encounters, levels) requires many iterations and interactive playtests to uncover a satisfying result. Crafting a coherent and effective sequence of these elements into an entire game, called a *progression*, is even more involved because the experience of playing the game is highly dependent on the way individual components are connected. As game designers adjust their game at these different scales—altering details of

a single gameplay element and scheduling appropriate introduction of those details across the game's entire progression—they wrestle with a mixture of formal and informal requirements. Keeping all of these concerns in mind while designing each level is difficult.

Game designers have adopted semi-structured practices for addressing this complexity. They often create informal notes and plans that capture a sketch of the progression's overall properties and then consult this document while designing individual levels. Although developers often create complex in-house tools for authoring game content (e.g., the Dragon Age Toolset[1]), these tools focus on individual levels and rarely explicitly model progressions. As designers improve their level designs in response to testing, the progression plan is rarely updated and the high-level structure of the final level progression is not articulated in general terms again.[2] As a result, the coherence and intent of the original plan can be lost.

We propose the creation of progression design tools that aid game designers through all stages of design: sketching, rapid exploration, iteration, and final authoring of a complete game progression while keeping the progression plan in-context and up-to-date. Such tools should allow editing of both individual elements and progressions over those elements. Building on the ideas of mixed-initiative planning and constraint-based game content generation and verification, we demonstrate a prototype tool using the educational game *Refraction* that combines existing single-level editors into a mixed-initiative progression design tool.

This tool represents the game at multiple layers: individual puzzles, a progression of which skills should be used in each puzzle, and high-level contraints over the progression. In this context, the domain model takes the form of the set of skills that each puzzle may or may not require to be solved. Following on from the previous chapter, this representation of domain models differs from the one we will be generating from Chapter 4 onwards, but serves the same function: serving as a formal represention of expert solving strategies for the problem domain. Thus, a progression essentially defines which skills from the domain model

[1] `http://social.bioware.com/page/da-toolset`

[2] In a very interesting exception to this trend, fans sometimes recreate visual progression plans for popular games, such as Piotr Bugno's detailed outline of the story and level progressions for *Portal 2*: `http://www.piotrbugno.com/2012/06/portal-2-timelines/`

are required by each puzzle.

Our focus on tools for designers contrasts with current trends in game design automation research. Many projects aiming to reduce designer burden offer fully automatic generators for individual levels, often focusing on optimizing a fixed metric for the quality of the level [132]. These systems are not designed to optimize the relative placement of levels within a progression. Other fully automated generators explicitly take user-configurable constraints as input [109, 117]. These generators are directable down to a fine scale but still require an external progression plan. Zook et al. [140] describes a system for fitting a progression of generated challenges in a training game to an ideal player performance curve; however, the system requires all design input to take the form of formally modeled properties such as evaluation functions and causal coherence constraints. We are interested in allowing the designer input at all scales, so fully automated methods will not suffice. Mixed-initiative design tools, such as Tanagra [121] and SketchaWorld [115], pair generative techniques with an interactive editing interface that allows the human and machine to take turns editing a shared level design. Such tools help the designer prototype new ideas and check constraints for quality assurance while still allowing designers to craft levels with subtle properties that are difficult to formalize. These tools focus on creating single levels, whereas we want to create a tool that allows designers to work at multiple scales. However, we follow similar design process pattern for the problem of designing coherent progressions of detailed levels.

Effective progression design tools could provide valuable assistance to the same expert designers who would normally have designed levels without such support. We expect that they would be almost required for very large, complex progressions where manual exploration becomes intractable. More importantly, however, they would open up possibilities for end-user progression design. Consider an experienced teacher who wants to directly manipulate which concepts will be used in a progression tailored to their class. They might prefer that the tool automatically generate the relevant challenges at the scale of individual puzzles everywhere except where one or two key levels should visually resemble examples previously shown in class. This property is unlikely to be supported by any fully automated system.

In this chapter, we discuss the requirements of a progression design tool, and present our prototype implementation for the educational game *Refraction*, illustrating its utility through several case studies. We expect that these ideas might be explored in domains outside of games such as interactive programming tutorials or the education domains targeted in later chapters of this thesis. The work presented in this chapter makes the following contributions:

- We identify the need for progression design tools and describe their impact with respect to current practices.
- We sketch a general architecture for progression design tools, including the use of generative techniques on a per-level basis where available.
- We present a prototype implementation of our design tool, attached to our own active design project, *Refraction*.

2.2 *The Game* Refraction

Before describing further background or the technical details of our system, we describe *Refraction*, the application used in both this and the following chapter. In *Refraction*, players solve spatial puzzles by splitting virtual lasers into fractional amounts. Each puzzle is played on a grid that contains laser sources, target spaceships, and asteroids which obstruct lasers, as shown in Figure 2.1. 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 to schoolchildren, it has found popularity with players of all ages and has been played over one million times. *Refraction* is (at the time of writing) freely available and can be played by anyone with a web browser.

The game consists of a sequence of puzzles in which the player must direct lasers into targets, a task that requires both spatial and mathematical problem-solving skills. The game relies primarily on the quality of puzzles to engage players. The original version of *Refraction* contains 61 puzzles that game designers (including the author) created by hand over many

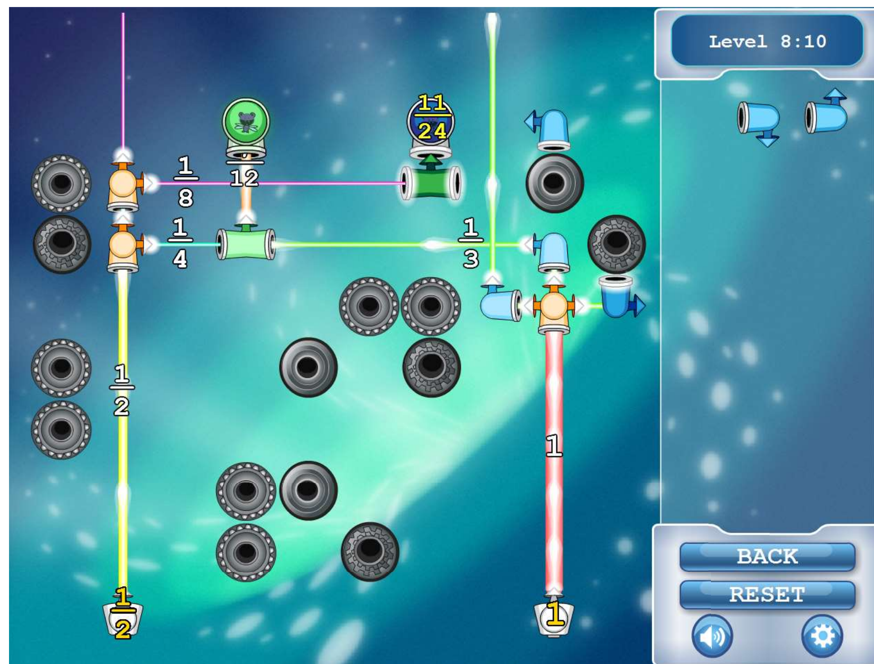


Figure 2.1: A level of *Refraction*, a free educational math game for which our tool was targeted. Over multiple years, the creators (including the author), continued to release new versions of the game with new content or even new mechanics. This motivated the need for a progression design tool.

dozens of hours. As the player progresses, the puzzles gradually increase in difficulty through the introduction of new concepts, such as benders, splitters, combiners, multiple sources, and multiple targets. The puzzles also increase in difficulty by combining base concepts together in way that necessitates more complex search processes, such as puzzles that require bending a laser clockwise and then counterclockwise to circumvent an obstacle, or puzzles that require splitting a laser into halves and then into thirds in order to produce one sixth. Describing a player's exact solution process for *Refraction* is very 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 puzzle correctly. Although most of these solutions are slight positional perturbations of other solutions, solutions can also differ in qualitative ways. When the network of laser beams is examined as a graph, different solutions may

involve different connectivity patterns or even omit different sets of pieces from usage. For this reason, we cannot directly model solutions as unique sequences as in previously existing frameworks [3].

Though at the time this research was done *Refraction* had already been released, its creators continued, for years, to develop new content (puzzles and progressions) for the purpose of various research experiments or adapting the game to different age groups or audiences. This often included, for instance, testing brand new mechanics. This is one of the factors creating a strong need for tools that allow designers to effectively and rapidly explore and iterate level progression designs; designers are not always simply creating a brand new game, but often are adapting or iterating on an existing game by adding new material.

2.3 Current Practices

Game developers have created a wealth of guidelines and best-practice suggestions for creating effective level progressions in games. In this section, we review current practices and discuss which aspects our system aims to improve. These practices are not well-documented, and there is not a clear accepted plan for tackling these challenges. Industry magazines such as Gamasutra[3] archive articles and discussions on game development written by industry members, from which we draw our information.

2.3.1 Level Progressions

Many games consist of a sequence of distinct elements (e.g., levels, stages, challenges, scenarios or puzzles) that the player encounters. We call this sequence a *progression.* Some games (including *Refraction*) impose a total ordering on this sequence; players must go through levels one after the other. However, many games' progression elements can be explored in a less restricted order; game designers call these *non-linear* games, though they still use the term *progression* to describe the player's progress through partially-ordered content. Our

[3] `http://gamasutra.com`

tool focuses on games with a total ordering of levels (i.e., *linear* games), though many of the ideas presented here also apply to non-linear games. In *Refraction*, the elements are the individual puzzles (involving 2–5 minutes of play), and the progression is the entire sequence of puzzles (2–5 hours of play). There are many ways to discretize a game progression; puzzles in *Refraction* are very short, discrete units so make a reasonable choice. Games with larger discrete units or no obvious discrete boundaries may make other choices about what counts as a unit of progression.[4]

In this thesis, we will generally refer to the individual elements as *levels*. Levels cannot be designed entirely in isolation; how they relate to each other is critical for effective game design. Much of the effort spent during the iteration process focuses on altering individual levels to improve the overall progression.

As a result, game design often happens at multiple scales. At a broad scale, designers create a *progression plan*, which documents the features they want each of their levels to have. These features might include, for example, which mechanics appear in a level or which key dramatic events occur in the game's narrative for a story-oriented game. In *Refraction*, there are several different types of pieces the player may use, such as bending pieces, splitting pieces, and adding pieces. The pieces required to solve any particular level are a feature we consider during the design process. At a narrow scale, designers are editing the levels themselves, trying to craft levels that have the complex properties they desire.

There are many design considerations when creating progression plans. For example, many games introduce different mechanics slowly and deliberately to allow the player to learn each of them without becoming overwhelmed. Game designer Dan Cook associates these mechanics with *skill atoms* [33] and suggests diagramming the dependence between atoms to better understand how a game works.[5] In many cases, the introduction of a new skill atom

---

[4]Games may not even have a good single definition of a level. While a two-tiered view of levels as atomic units within games is reasonable for *Refraction* and a useful simplification for this work, other, particularly larger games might be better thought of as several layers of progression over finer and finer elements. We do not handle such complexity here but many of the ideas can be applied to such a setting.

[5]The theory of skill atoms (and others used in the industry) are created by practicing game developers.

may depend on the player having previously mastered another. For example, in Nintendo's *Super Mario Bros.*, the player should learn to jump before they can learn to jump on an enemy. Similarly, overusing a certain atom can lead to burnout, and avoiding this requires additional constraints between levels to be checked. In *Refraction*, we wish to introduce pieces in a particular order and pace. Skill atoms may also have intra-level considerations: in *Refraction*, for example, some types of pieces cannot be used unless another type of piece is present in the level. For the purposes of this dissertation, we can view skill atoms as the elements that make up the domain model of a game: the set of all possible skill atoms a player might learn represents the expert solving strategies for the game.

Another important design consideration of progressions is pacing. *Pacing* describes how elements such as complexity, difficulty, and intensity vary over the course of the game. Game designer Jenova Chen writes [32] about how Csikszentmihalyi's concept of *Flow* [35] applies to game progressions. If the difficulty increases too quickly, players can become frustrated; too slowly, they become bored and disengaged. Many articles have been written on Gamasutra analyzing and discussing techniques for effective game pacing [6].

2.3.2 Design Practices

Designers understand the value in explicitly planning their games' progressions. Progression design considerations and progression plans are often sketched out in a design document or whiteboard before production of levels. Designers have written articles endorsing planning game progressions before production, writing that teams that do not explicitly create progression plans often end up redoing work after user testing reveals problems with their

While sometimes (as in this case) inspired by research results, these theories are typically not rigorously supported. Nevertheless, both because they are used in practice and because there is not a lot of rigorous work on player skill development, theories like skill atoms are suitable representations of domain models for this system. The work in later chapters on RuleSy draws its inspiration directly from research in educational psychology rather than mimicking existing praxis.

[6] e.g., `http://www.gamasutra.com/view/feature/134815/` or `http://www.gamasutra.com/view/feature/132415/`

progressions, resulting in a worse final product[7]. However, the progression may change greatly during production, user testing, and iteration of levels; levels and mechanics may be reordered, or mechanics may be dropped entirely or new ones added. So while planning progressions beforehand is very useful, if left unaltered, the progression plans become increasingly inaccurate. Although game developers frequently spend great effort creating in-house editors and other authoring tools for their games[8], to the best of our knowledge, they do not create editors for their progression plans that automatically sync with their levels. For designers to continue to work at this broad scale during development, they must first evaluate the current progression manually, a significant task[9]. After production, sometimes developers (or even players) recreate visualizations of the progression realized in the final game. While these may be used in post-mortem analysis, they are still created mainly by hand, and are of little utility for design of that game, since it has already been published and is unlikely to undergo large changes. Integrating an up-to-date progression plan as part of the standard editing environment could help the designer more effectively work at multiple scales.

Because the information about the progression plan is cumbersome to keep in sync with the actual level progression, ensuring that the current levels meet all progression design considerations is an expensive, error-prone task. Edits to levels frequently change their properties in ways which may, for example, disrupt desired pacing, or introduce game mechanics too quickly or in the incorrect order. If not noticed by the designer, these problems will eventually be discovered in user testing, a relatively expensive resource. Automated detection of these problems may save time and effort by finding issues quickly, without designer effort.

We expect having the ability to rapidly explore different progression plans would enhance

---

[7]e.g., `http://www.gamasutra.com/view/feature/3848/`

[8]For example, players can create complex mods and scenarios using *Starcraft 2*'s in-house level editor: `http://us.battle.net/sc2/en/game/maps-and-mods/`

[9]Recommendations to do just that come from `http://www.gamasutra.com/view/feature/132256/`

the design process. However, if the designer wishes to make a minor adjustment to the progression, say moving the introduction of a mechanic to a later point in the game, this might require significant modifications to existing levels. While a large portion of game developers create level editors to help author levels, these adjustments are still expensive to make. Because of this, designers can be limited in how easily they can explore the design space.

Current design practices result in several problems we wish to overcome in the design of our system. Designers are not able to easily explore progressions because level authoring is time-consuming. Designs for progressions plans are often abandoned after level creation begins because they are too cumbersome to keep up-to-date. As a result, level edits often introduce problems that are not discovered without significant inspection and user testing.

2.4 Progression Design Tools

In this section, we formalize our design problem and discuss our goals for an effective game progression design system.

2.4.1 Definitions

We will use *levels*, *progressions*, and *progression plans* as defined previously. Each level has several *properties*, features that the designer cares about. Examples include which game skills (i.e., elements of the domain model) are used in a particular level, or whether a particular narrative event occurs in a level. There are often a huge number of possible levels that share a particular set of properties. Likewise, there are many progressions that conform to a particular progression plan. The designer may have particular considerations for their game's progression, such as pacing or the ordering in which mechanics are introduced. We refer to all of these considerations definable at the level of the progression plan as *progression constraints* (or just *constraints*). We use the term (loosely) in the sense of optimization problems: the designer has a set of constraints in mind, and the design problem is to generate a progression best satisfying those constraints. Constraints may be hard (never to be violated),

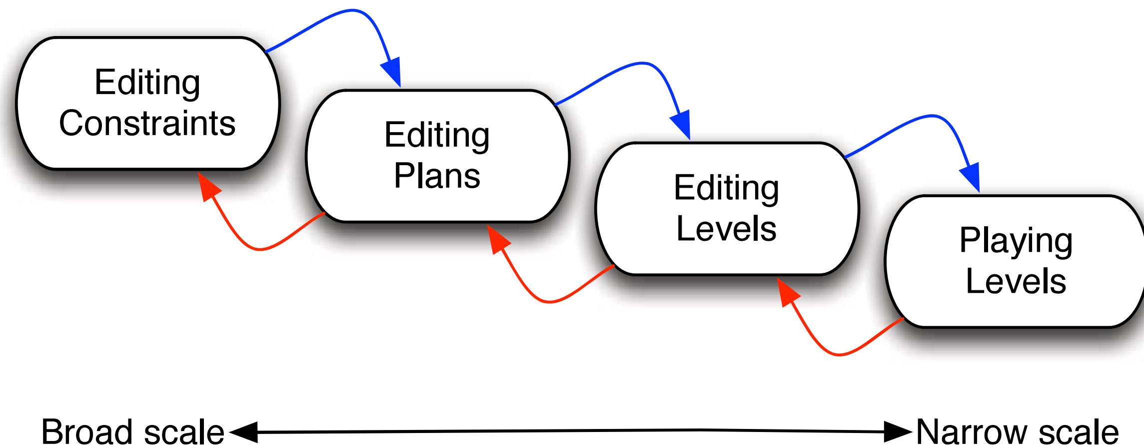


Figure 2.2: The ideal workflow for our system (a waterfall-like pattern). Designers must iterate at multiple scales, so our tool aims to allow the designer to move freely between broad and narrow-scale editing.

soft (flexible in exchange for satisfying other constraints at some cost), or unmodeled (not tracked by the tool). Again, there will generally be many progression plans that equally satisfy a particular set of constraints.

2.4.2 Goals of the Design Tool

The tool is intended to be a full-fidelity game editor and for authoring the final levels seen by players. The system should support rapid iteration and exploration at multiple scales of the design: progression constraints, progression plans, and individual level designs. Designers should be able to playtest the current progression. Figure 2.2 demonstrates the desired workflow for the system. Many constraints and properties can be formally modeled, so we would like to draw on advances in procedural content generation to automate generation and analysis of content where possible. Level generation is not an option for all types of games (and can be too costly to implement even when it is), so it is an optional component of our tool. It is important to note that many critical design considerations, such as the moment-to-moment difficulty or affective impact on players, are subjective or cannot be easily modeled.

Concept	Description
Bending	Must use bending pieces
Splitting	Must use splitting pieces
Adding	Must use adding pieces
Blockers	Must contain obstructive pieces
Wasted Laser	Leaves some laser beams unused
Crossed Laser	Laser beams are unavoidably crossed

Table 2.1: A sample of the properties we model for levels in *Refraction*. These properties are binary, describing whether or not a particular gameplay concept is required to solve a level. These concepts, taken together, constitute the domain model for this application.

Thus human editing *must* be available at all scales, so the system should integrate existing level editors that game designers already create. As the designer changes from broad-scale progression planning to narrow-scale level editing and back, the system should automatically keep other parts updated: edits to levels should be reflected in the progression plan, and edits to the plan should notify the designer if plan-level constraints are violated.

2.5 *Application to* Refraction

To ground this discussion about progressions, levels, and constraints, we discuss how these ideas apply in the design of our game, *Refraction*. In this section, we discuss progression constraints and level properties specific to *Refraction* used in the proof-of-concept implementation of the tool as well as which game-specific components we needed to provide for the implementation. Figure 2.3 shows screenshots from our prototype intended to demonstrate how a designer might control and view plans and levels. We delay discussing the system components until the next section.

2.5.1 Constraints and Properties

All level properties we model in this prototype describe whether a particular gameplay *concept* (or *skill atom*, to use Cook's [33] terminology) is required to solve a level. The set of all

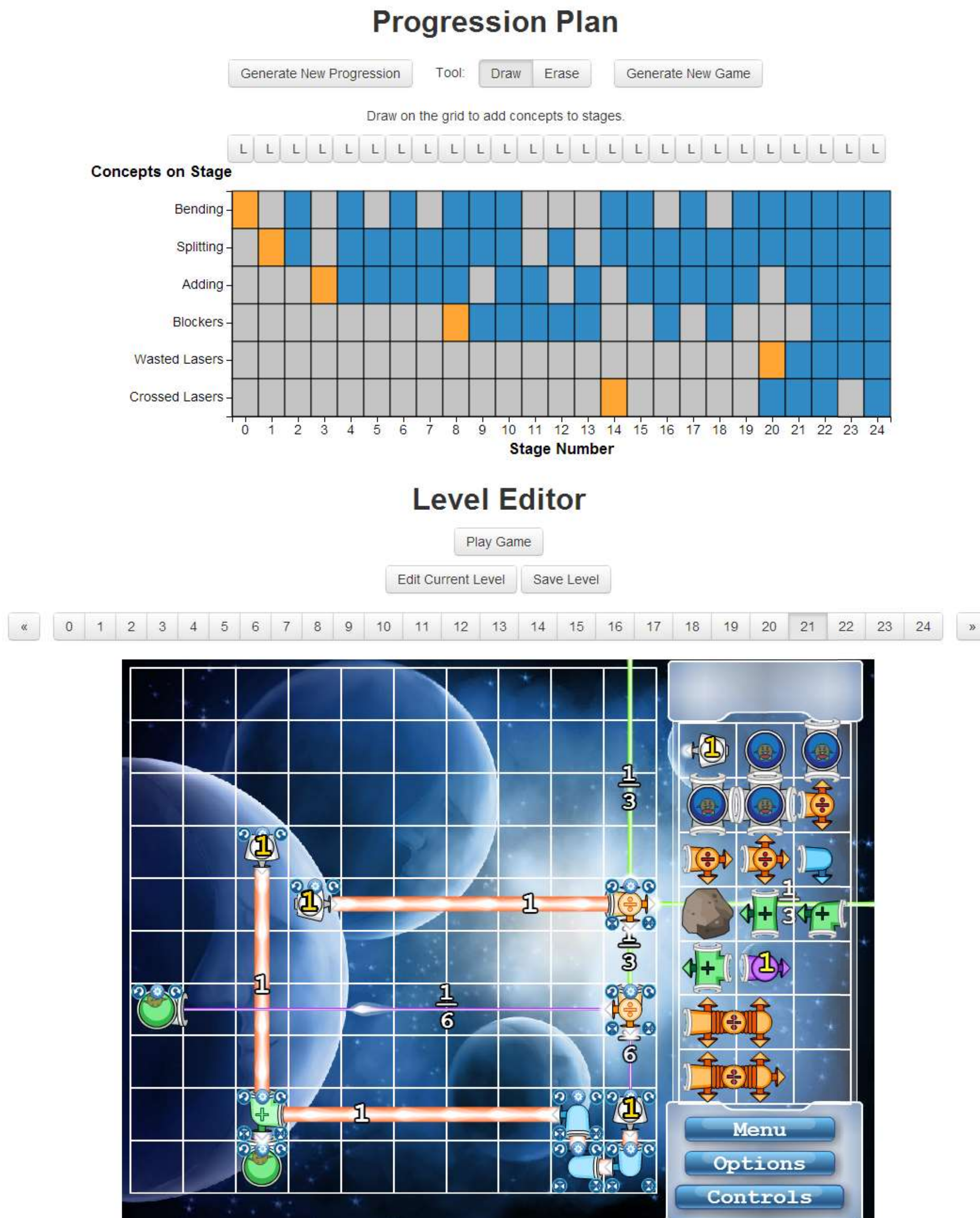


Figure 2.3: Screenshots of two of the three browser-based editing environments of our tool, each for editing the game at a different scale. On top is the *progression plan editor*, used for directly manipulating the plan. On the bottom is the *level editor*, which embeds *Refraction*'s custom editor. There are additional views for editing constraints and playtesting.

possible concepts/skill atoms constitutes the domain model for this game. As mentioned in the previous chapter, this is a much simpler representation than the domain models we will be automatically generating in later chapters. Thus, we will use the term *concept* to refer to the elements of the domain model rather than a term such as *strategy* to highlight this simplicity. Table 2.1 lists a sample of these concepts. Because we currently only model concepts, all of our level properties are binary, describing whether a concept is required. Many additional properties have not yet been modeled, such as how closely the pieces must be placed together in all possible solutions. Some cannot be directly modeled, such as whether a level is fun or aesthetically pleasing. Though not tracked by our tool, these can still be addressed through manual edits to level designs. We have implemented four types of progression constraints, described below.

Prerequisites

Prerequisites are *inter*-level constraints that define a partial ordering over the introduction points of the concepts (which must be realized by some total ordering in the progression). For concepts A and B, if A is a *prerequisite* of B, then the level of A's first appearance must precede the level in which B first appears. For example, we want to introduce pieces that bend the laser before introducing pieces that split the laser.

Corequisites

Corequisites are *intra*-level constraints on which concepts can appear together. A is a corequisite of B if A must show up in any level that has B. For example, one concept is to require that the laser be looped around to cross itself. This is only generally possible if bending pieces are available. Thus, "Bending" is a corequisite for "Crossed Laser."

Concept Count

For each level in the progression plan, we compute an illustrative proxy measure for "level intensity" (a term often occurring in other designers' progression plans) as a count of the concepts that are in the level. Although this metric does not correspond directly with real-world difficulty, it is nevertheless useful for controlling pacing of the progression at a broad scale with a few quick adjustments. The constraint assigns target number of concepts for each level. Our interface allows control via a spline editor, as seen in Figure 2.5.

Concept Introduction Rate

The *concept introduction rate constraint* controls the rate at which new concepts are introduced, describing, for each point in the progression, the number of concepts that should have been introduced by that point. This is an equality constraint: the progression must introduce at precisely the rate given by this constraint. This constraint, expressed via spline editor, can be used to ensure concepts are introduced at a pace allowing the player to master one before proceeding, as well as roughly showing how the complexity of the game changes at a glance.

Clearly, these concepts are specific to *Refraction*. However, the progression constraints are quite general. Given another game's set of gameplay concepts, we imagine that prerequisites and corequisites will still be meaningful for shaping progression plans. The designer could specify constraints that apply to properties related to categories other than gameplay, such as aesthetics. For example, a designer might want to specify that aesthetic properties like background music and graphical tileset should change in groups, corresponding to movements to different places in the game's fictional world.

2.5.2 Game-Specific System Components

Using these particular properties for level generation and analysis in *Refraction* requires a subtle technical approach. Ensuring that a concept is *required* to solve the level entails

checking that *all* possible solutions of the puzzle (there are generally many) use the concept, a computationally difficult problem. For example, in a level in which the player is expected to split a laser in two then later add it back together, careless design might enable the player to use only bending pieces to bypass the splitting and adding altogether. Smith et al. [116] developed a level generator (with *Refraction* as the example application) that can enforce this type of constraint on its outputs. The generator uses *answer set programming* (ASP), a declarative constraint-programming technology, to ensure that specified concepts must be used in all possible solutions [44]. In addition to reusing this generator in our system, we extended it to use the same formal game model to determine if a designer-altered level requires each gameplay concept. This level analyzer is used to update the progression plan after levels are manually edited.

We created a progression plan generator using ASP's ability to solve constrained optimization problems. Progression constraints (e.g., prerequisites) are expressed as soft constraints and manual edits locked by the designer as hard constraints. Soft constraints are implemented with an integer-valued penalty function. The generator searches for plans satisfying all hard constraints with the minimal penalty for soft constraints; this problem is an instance of the weighted MaxSAT problem [17], in which we must satisfy a set of hard constraints while maximizing the weighted-sum of a set of soft constraints. This problem can be solved by our answer set programming tool. We include additional soft constraints that add variation to the plan by penalizing repetition. We implemented (in Javascript) a corresponding analyzer that, given a progression plan, checks for violated constraints.

2.6 System Description

In this section we describe in detail each of the system components we built for our *Refraction*-based prototype. For each component, we discuss its role and how our implementation works. Some of our implementations of these components are game-specific, while others could be directly reused for other games. Of course, all parts of the system can be extended or replaced in the interest of better supporting the designer. Figure 2.4 illustrates how the

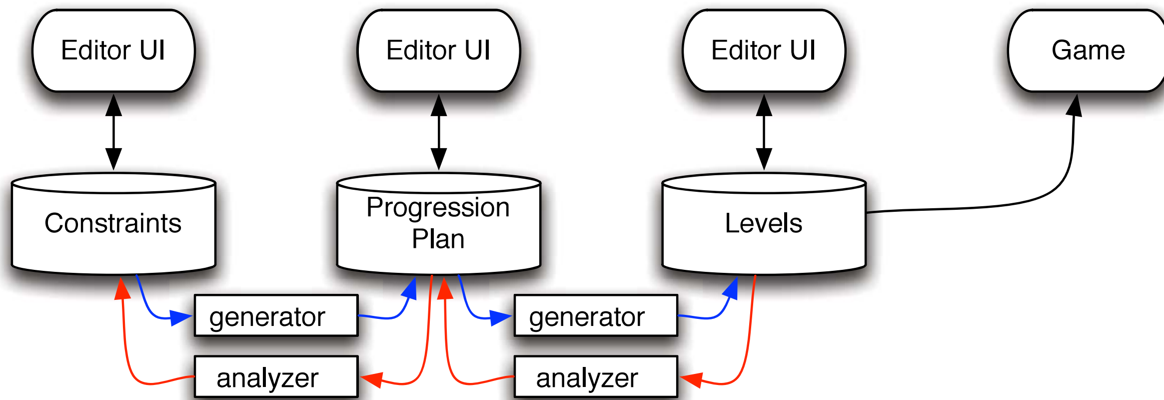


Figure 2.4: The pieces of our system. Cylinders are submodels, which together are the full model of the game, its progression, and its constraints. Ovals represent editing/viewing interfaces that correspond to the four elements of the workflow outlined in Figure 2.2. The square components are the automated parts that ensure consistency across scales.

system components fit together.

2.6.1 Model

The model that the tool manipulates consists of three parts, which directly correspond to the scales of the workflow: the progression constraints, the progression plan, the progression itself (the sequence of concrete level designs). The designer can also output the playable game from the tool.

2.6.2 Working with Progression Plans

The broad-scale iteration loop takes place between manipulating progression constraints and the progression plan. The system has a set of user interface components used to display and manipulate the progression constraints. Figure 2.5 shows the set used in our implementation. These components are responsible both for letting the designer adjust the constraint parameters and showing whether the current progression plan violates these constraints. As the

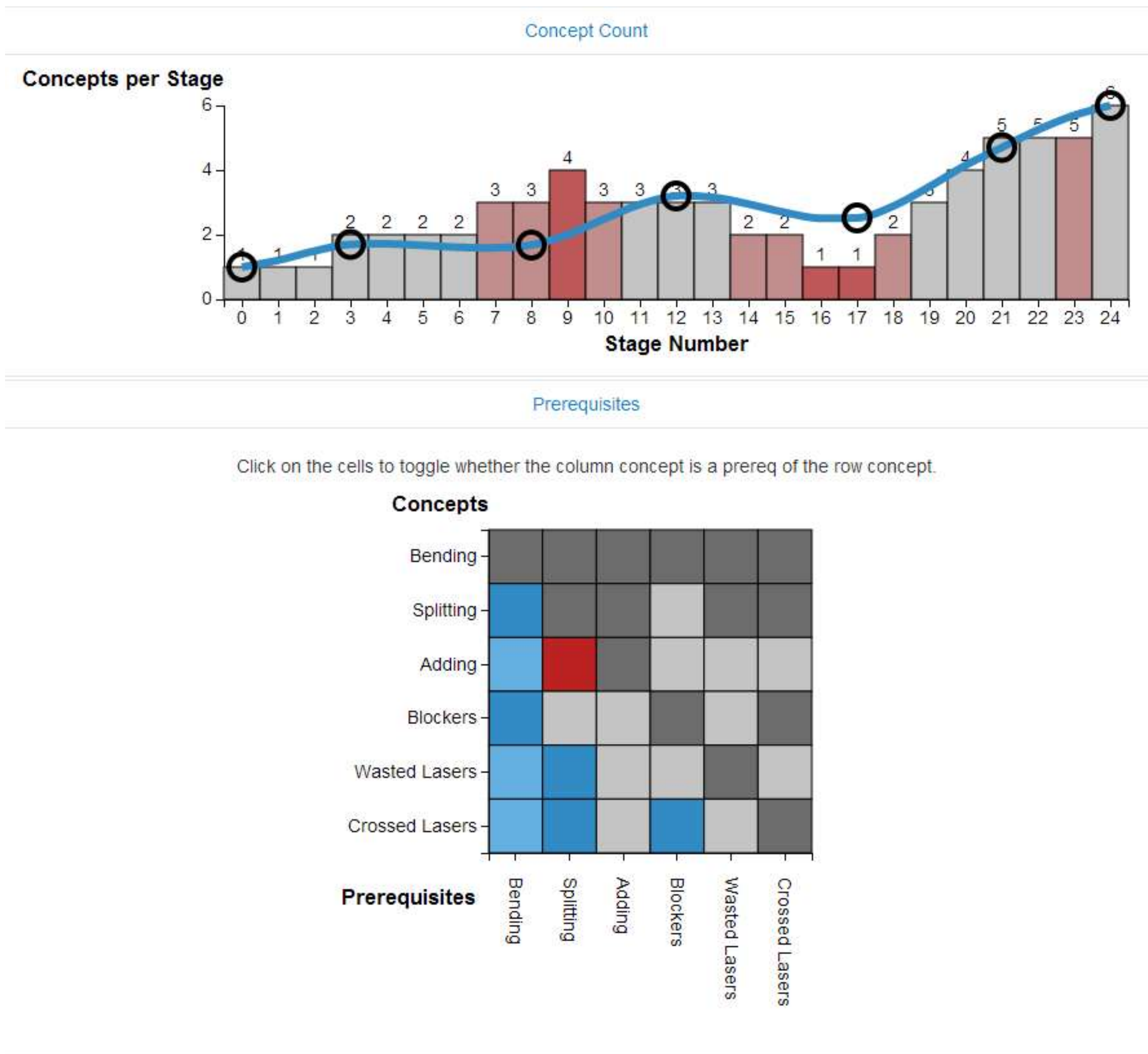


Figure 2.5: View of the two types of constraint editors in our implementation, which allow manipulation of the various constraint parameters. Each interface type is used for multiple constraint types, e.g., the grid is used for both prerequisites and corequisites. If the current progression plan violates these constraints, the violations are displayed in red. On top, the *concept count* graph shows us that several levels have too many or too few concepts. The blue curve is the target, and the bars are the values of the current plan. On bottom, the red cell of the *prerequisite* chart shows us that the "adding" concept is erroneously introduced before the "splitting" concept. Blue indicates an active constraint (dark is manual, light is inferred via transitivity), while dark gray indicates an unselectable constraint (e.g., two concepts may not be mutual prerequisites).

designer changes the constraint values, the component interfaces update to display whether the current progression plan satisfies the new constraints.

The primary component for manipulating the progression plan is the *progression plan editor*, shown at the top of Figure 2.3. For our implementation, this consists of a grid showing level properties of each level, where the cells are the current value of the property for that level. The user can directly manipulate this grid to change level properties. As the designer edits the plan, the *progression analysis* component checks whether the constraints are still satisfied. This is then reflected in the constraint editors in real time.

The designer manually manipulates the constraints and progression plan in order to produce a plan that matches their intent. However, the *progression generation* component can be used for rapidly populating the plan and sampling alternatives. When manually activated by the designer, the progression generator creates a new progression plan (overwriting any existing) that best satisfies the current constraints.[10] In order to prevent any manual edits from being overwritten by generation, our system allows the designer to "lock" a particular level. The generators will not modify these levels or their corresponding entries in the property matrix.

Thus, the full iteration loop for this scale is: set values of the constraints, optionally generate a new progression, manually edit the progression, check whether these edits violate any new constraints or unmodeled design criteria, repeat.

2.6.3 Working with Levels

The narrow-scale iteration loop involves manipulating the progression plan and the levels themselves. The structure of components at this scale corresponds with the structure above. First is the previously described progression plan editor. Like the constraint editors, manipulation of the progression plan does not immediately change the levels below.

---

[10]As there are often many progression plans that achieve an equivalent score with respect to the stated constraints, we have configured the progression generator to make liberal use of randomness in the interest of exploring diverse solutions.

The *level editor* is used for editing individual levels. Because levels are always game-specific, a single-level editor must be supplied by the game's developers. Developers must also provide a *level analyzer* that computes the relevant design properties for each level. The difficulty of producing such an analyzer scales with the subtlety of the properties checked: determining if a level uses a particular piece or is set to use a particular graphical tileset is easy, but determining the level of strategy needed to complete the level is harder. When the designer edits a level, the level analyzer is used to immediately reflect the changes in the progression plan. These changes are propagated further up to the progression constraints.

For the purposes of automation, the developers may optionally provide a *level generator*. Though the designer could manually edit levels, certain games, including *Refraction*, have generators that divert significant burden from the designer. The level generator, if it exists, can be triggered by the designer to create levels that match the properties described in the current plan. Any levels that are "locked" are not overwritten, so manually-tuned levels can be preserved.

This iteration loop corresponds to the one above: edit the progression plan, optionally generate levels, manually edit levels, check how the progression (and the progression constraints) changed based on these edits, repeat.

2.6.4 Testing the Game

The final component of the system allows for playtesting the levels directly. Thus the game must have an interface that accepts new level progressions to be played immediately. In the case of *Refraction*, since it is an online game, it can be easily launched from within the tool.

2.7 Evaluation

The implementation of the prototype system was intended for internal use by the authors of the original publication [27]. We evaluate the general architecture and our game-specific implementation through a set of case studies, exploring new design techniques that our system enables. The first case study requires the game-specific level generator, but the

others do not require such technology.

2.7.1 Rapid Exploration at a Broad Scale

Assuming the system has a level generator, this tool enables game designers to rapidly explore different progression plans and test the plans in playable games. For example, consider a scenario for *Refraction* early in the design phase of a new progression for a new, younger audience. We do not know how this set of players will respond, so we wish to rapidly explore and playtest several possible progressions. We might want to remove “addition pieces” from the game entirely and avoid introducing the “crossed laser” concept (see Table 2.1) until late in the progression. Our tool allows the designer to quickly sketch these modifications into the progression plan, generate a new set of levels satisfying the plan, and test these levels immediately. While these generated levels may lack some of the unmodeled properties the designer desires, this is acceptable at an early stage of rapid exploration. The designer also has the ability to edit the generated levels to suit their needs, which can be easier than creating new levels from scratch. Previously, creating even an initial progression with the key gameplay concepts removed would have entailed massive effort before the first round of tests. An even stronger motivation for this use case is the situation where the designer wants to modify an existing game by adding in one or more new mechanics. The cost of creating an entire new set of levels that integrate the new mechanic is high.

This rapid exploration applies at the broader constraint scale as well. Suppose in *Refraction* we wish to add an additional prerequisite constraint that “laser crossing” must come after introducing “splitting pieces.” The tool can automatically create a progression plan in conformance with this constraint. It then populates a sequence of levels compatible with that plan, allowing a game satisfying the new constraint to be played with minimal designer effort. Without our system, the designer would have to first manually evaluate where the existing plan failed, devise a new plan for an adjusted progression (in notes outside of the editor), then create several new levels, either by hand or by manually configuring the level generator. We expect this reduction in friction to enhance rapid prototyping of progressions.

2.7.2 Automatic Detection of Problems during Iteration

As discussed, levels and progressions undergo a great deal of iteration during the development process. One of the dangers of changes during iteration is breaking previously expressed planning intents, both global progression constraints and level property constraints.

Consider an example in *Refraction* where, late in the development process, the designer wishes to make a minor adjustment to the mathematical portions of the fifth level of the progression. They exchange a "splitter piece" with two outputs for one with three outputs. However, this extra output introduces what we call a "wasted laser," or a laser that is not used in the solution. 'Wasted lasers" is one of the concepts controlled in the system and is constrained via the prerequisite constraint to first appear after another concept, "crossed lasers," which has not been introduced by this level. This could be corrected by adding additional pieces (a new target to absorb the extra laser), but may go unnoticed by the designer. However, upon saving the change in our tool, the system will automatically update the progression plan to reflect that wasted lasers appear in this early level. Since a prerequisite constraint is now violated, the constraint editor highlights this problem to notify the designer. The designer may then take steps to fix it: one option is moderately editing the offending level. If the designer prefers to keep the level as-is, they can generate a new plan for surrounding levels or remove this prerequisite constraint. We believe that this feature can make managing large design projects easier by notifying the designer of automatically detected problems.

2.7.3 Global Optimization of Progression Properties

A designer often needs the progression to satisfy many complex time-varying properties simultaneously. For large progressions this task is difficult to do manually. For example, suppose a designer for *Refraction* is trying to control the pacing of the progression. They are interested in controlling aggregate properties: the number of concepts per level, and the rate at which new concepts are introduced during the progression. The designer wants

new concepts to be introduced at an even rate, but they want the number of concepts per level to vary significantly between levels. These constraints are modeled in our system, so the designer may directly edit curves (see Figure 2.5) to control the constraints. Our system's built-in progression generator can find the closest feasible solution through global optimization. As the designer refines these curves, they can quickly see possible solutions and adjust parameters appropriately. The designer may also rely on the fact that all design changes that were manually introduced can be preserved through the optimization process. We expect this to ease the process of creating progression plans that satisfy several complex constraints.

2.8 Discussion

We believe that ideas from this system can be applied to a wide range of games. Some components are game-specific, such as the level editor and generator. The others, particularly those at the broader scale of progression constraints and plans, can be reused between games; the pairing of our constraint editing interfaces with constrained optimization tools on the back end are generic. The modularity of our framework allows it to be extended for new games and for additional functionality. In this section we discuss some of the limitations and some of the possible ways in which the system could be extended. We then discuss how these ideas might be explored in other, non-game domains.

2.8.1 Limitations and Extensions

We have defined progression plans as a sequence of level properties. In our implementation, these properties are binary values indicating whether a level contains a particular concept, but more nuance is certainly possible and very likely necessary for practical applications. For example, we may be concerned with *how many* bending pieces appear or the overall size of the player's solution. Having numeric knobs instead of binary values would enable the designer to specify even more precise progression plans. For many applications, a designer may be able to specify these concepts manually. For others, particularly skill-based games

or educational domains, it is a daunting task to figure out a set of concepts that reflects how players actually solve the challenges presented in the game. The following chapters in this thesis, particular the RuleSy framework presented from Chapter 4 onward, was created to tackle precisely this task.

The system is also extendable to address some aesthetic concerns; for example, we might give the designer control of the average distance between pieces to allow them to create dense or open-feeling levels. We have already explored constraints on symmetry, balancing, and packing for *Refraction* levels, but elided those properties from the prototype presented here. These new constraints are significantly more detailed than the current set and will require new display mechanisms to make them easy to see and manipulate.

Our prototype supports only a linear progression, limiting its application to some games. For example, in Nintendo's *Super Metroid*, the player explores a large, nonlinear open world. The player finds "power ups" in this world that unlock new abilities, allowing the player to explore previously inaccessible locations. Therefore, the "progression" is defined over which abilities the player has unlocked and which mechanics they understand how to use, rather than a single number describing how many levels they have completed. While extensions to the system to support other progression structures are plausible, most require finding a natural visual depiction for such structures. Currently, our ability to generate progressions and levels under hard and soft constraints exceeds our ability to present useful interfaces that would allow a designer to guide these generators in directions that would satisfy some external intent, so more research must be done to explore how to depict and interactively manipulate these more abstract progression mechanisms.

Refraction is a game of relatively small scope, but we believe this system could be used for much larger games. To remain tractable, one strategy would be to only model the most salient properties of levels and solutions. If complex levels can be created by connecting detailed tiles (e.g., pre-authored chunks of terrain in a 3D adventure game), then the representation used in progression planning can remain discrete and visually manageable. Another strategy would be modeling at additional scales. For *Refraction* it was feasible to work at two scales

because puzzles are short enough to be treated as atomic objects. However, many games have very large "levels," such as the single-player campaign missions of real time strategy games like Blizzard Entertainment's *Starcraft*. While designers may wish to model progressions over the entire game's missions, the levels themselves are quite long, and the designer may wish to model an "intra-level" progression of a single level, or perhaps even modeling the progression over individual encounters within a level. Such progressions could be modeled by hierarchically nested progression diagrams: scales for each encounter, mission, and overall game.

We have supplied a small example of possible progression constraints, but they can be extended or new ones introduced. Luckily, many progression constraints apply generally to a broad class of games and so can be reused. This is in contrast with level constraints, which are almost always game-specific. For example, many games are concerned with the order mechanics are introduced, so our *prerequisite* constraint can be used in those games. Likewise, many games require care with pacing, so a constraint dealing with how long concepts should be practiced after introduction before moving on to new concepts would be generally useful.

The existing user interfaces for constraint editing can be repurposed for different kinds of constraints. For example, in addition to prerequisites and corequisites, we might use the same interface for constraints such as: *mutual exclusion*, where concepts A and B shall not show up in the same level; *sequential exclusion*, where if A was used in a previous level, B cannot be used in the current level; or *sequential introduction*, where A must be introduced immediately preceding B.

Similarly, the spline editing interface can be reused for many designer-specified functions. For example, we can model a sense of "scale" for *Refraction* levels by measuring the minimal number of pieces required to complete the level. Our *concept count* constraint gives a very primitive idea of level intensity, and as long as a designer can usefully employ edits of this curve in place of lower-level tweaks, it is valuable. However, it would be desirable to use a function more representative of real-world difficulty and player experience. Backing the spline editor with more sophisticated functions, such as a learning rate derived from an

externally-validated learner model, is an obvious and enticing avenue of future work.

Our system has basic support for preserving manual edits by "locking" entire levels as fixed with respect to the generators, but better methods could be applied to make the tool more useful. An obvious first step is allowing more fine-grained locking by locking only particular concepts: for example, the designer may want to ensure that splitting shows up in levels 2–5 but wants to allow the generator to make other decisions for those levels. Allowing the generators to reorder existing levels or minimally modify levels while preserving designer-specified features may also make the locking tool significantly more useful.

Finally, the generators and analyzers can be replaced as technology improves. Depending on the game, many existing level generation technologies would fit in the system. While answer set programming fit our game well for both level and progression plan generation, we could explore many other optimization or planning techniques. For example, for level generation, a game with continuous physics like Rovio Entertainment's *Angry Birds* might use a generate-and-test system that uses the game's internal physics engine. Many relatively simple techniques, such as making an approximate query against a database of previously authored and annotated plans and levels, are likely to be fruitful as well.

2.8.2 Application to Other Domains

We believe these ideas can be explored in other domains beyond games. Here we propose some potential applications of this system, looking at two other domains in closer detail.

Example Domain: Teaching Programming

As one example, we consider an interactive application that teaches programming, similar to *Codecademy*.[11] "Levels" in this domain are individual programming exercises. We speculate a designer may be concerned with properties such as which programming concepts (e.g., recursion, conditionals) are explained or introduced in the exercise, or which concepts the

[11] http://www.codecademy.com

user is assumed to already know. Another may be the complexity of the exercise, either by number of lines of code required or whether certain library function calls are required. Other properties are subjective in nature, such as the directness of suggestions given to find the solution, the narrative used to give context to the user, or the visual layout of the exercise in the application and access to related examples.

A designer may wish to ensure several global progression constraints over the exercises. Concepts should not be used until an exercise where they are intentionally introduced. Complexity should vary between exercises to prevent frustration or boredom. Concepts are revisited to give users sufficient practice, and should be recombined with other concepts; for example, designers may wish for each concept to be used with at least three other concepts.

This problem domain shares many of the characteristics of our game design problem (indeed, many of these concerns map exactly into Cook's discussion of skill atoms) so it should likely benefit from using a progression design tool. As most of our tool components can be directly reused, the designers need only supply an editor for exercises and tools to compute properties of crafted puzzles. For the computationally complex task of determining if there are alternate solutions to a programming task that avoid the use of certain concepts, program synthesis techniques could be used to search the constrained space of small programs in a language that fit a specification [12].

Example Domain: High-School Algebra Problems

We consider creating a homework assignment containing a progression of algebra problems. There are several level properties a curriculum designer may wish to control, some modelable, others subjective: which algebraic concepts are used (e.g., factoring, cancellation, distribution), how many steps a particular problem takes to solve, or whether the problem resembles in-class examples. In Chapter 5, we describe our framework, RuleSy, for automatically building a very detailed model of concepts and strategies used for solving algebraic questions. A designer could use such a model with this progression tool.

We consider several progression constraints a designer might wish to enforce, such as

the partial ordering of concepts (e.g., properties of roots before properties of logarithms). Designers may want to introduce new concepts in isolation to allow students to master them before combining them with others. They may also wish to ensure the supporting text for each problem is thematically coherent with nearby problems.

More generally, we suspect that the progression designer could be useful for analysis of the entire math or science curricula through 13 years of K–12 education, or design of a problem-based learning course.

2.9 Related Work

There is a long history of creating design tools in HCI research. Many tools have been created to support rapid exploration and prototyping at an early stage. Sketch-based tools such as Silk [73] and Denim [81] allow the designer to sketch interfaces with a stylus. Suede [66] explored the rapid creation of prototypes of speech-based user interfaces using wizard-of-oz techniques. Similar techniques could be used to prototype progression design tools in domains where automation similar to that which we built on is not yet available. In this case, the ability to sketch in broad-scale properties of a progression (e.g. a target pacing curve) and sample alternative plans is still useful even if per-level design is done by hand.

Systems like d.tools [55] support an iteration loop of creating, testing and analyzing interface prototypes. Because user testing and iteration is critical when creating game levels and progressions, our system also aims to support these kinds of iteration loops, but in a manner specific to our domain.

Juxtapose [56] and Side Views [130] are examples that allow the user to quickly explore alternatives to the current design. Our tools tries to support this through sampling randomly generated alternatives under constraints, though adopting a literal side-by-side view could be more effective.

Much work has been done using constraint solvers in user interfaces, such as Cassowary [9] and the engine in Amulet [95]. In contrast to familiar applications of geometric layout constraints on interface elements, many of our constraints are much more abstract in nature,

such as the nonexistence of shortcut solutions for a level that is supposed to introduce a new gameplay concept. Even so, we strongly separate the definition of the constraint (as driven by the interface of the design tool) from the back-end search technology used to find satisfying solutions. Creating generators and analyzers for *Refraction* involved setting up constraints and passing these to a domain-independent solver from the Potassco project [44].

There is also a rich history of work in mixed-initiative planning and collaboration tools. COLLAGEN increased the ease with which users could create mixed-initiative planning tools [107]. The framework specifies an interface with goals, recipes, and steps, to be implemented by domain-specific planners. COLLAGEN would then provide a dialogue system capable of conversing with the end-user, using the underlying planner. More generally, the SHARED-PLAN architecture underlying COLLAGEN allows computer and human agents to collaborate together in groups to satisfy goals, such as planning or interface design [47]. Our tool is more visually oriented, rather than relying on dialogue as a method of interaction; in addition, we optimize directly while planning subject to constraints instead of requiring recipes to search for solutions.

Mixed-initiative planners have also been used in other domains. NASA's MAPGEN is actively used to create daily activity plans for Mars rovers [22]. OZONE was designed to be a mixed-initiative constraint-based planning framework with pluggable components and was used to plan military resource allocation and transport tasks [123]. Our domain is different in that designers must both create plans and levels.

2.10 Conclusion

In this chapter, we have identified the potential utility of mixed-initiative progression design tools for games, described how such systems could work, and created an implementation to be used with our own deployed game. Progressions and levels are both very difficult to design, and it is arduous to juggle *all* design considerations while working at multiple scales. For several types of games, many of these considerations can be formally modeled, so we can use computation to automate portions of the process. At the same time, many design

concerns are completely subjective, so there is a strong need to retain human involvement in the creation of final progression plans and levels for deployment. We expect that game progression design tools can significantly enhance the game creation process. There are many other domains where interactive experiences are composed of scaffolded episodes. Practically all educational and general training environments fit this paradigm to some extent. We describe how given a breakdown of key concepts and the ability to automatically generate levels from parametric specification other domains directly map to our progression design process.

This tool is built upon a domain model, in this application represented as a set of the concepts and skills used by players to solve challenges in a game. In our example, we hand-picked a simple model to illustrate the proof-of-concept tool. In practice, this set of concepts should be one that reflects the skills actually used in the game. Figuring this is out is a substantial challenge, and later chapters of this thesis explore solving this design task automatically.

There are several areas of future work, in addition to potential extensions mentioned earlier. Experience with this prototype has already prompted a number of game-specific and general directions to explore next. For example, beyond adding more details to the progression plan (properties and curves), we want to investigate how concrete patterns discovered in the level editors (such as a commonly used cluster of pieces) can be upgraded into plan-scale properties without additional programming. Meanwhile, broader user studies are required to determine the effectiveness of this model before we can make progress on deploying progression design tools for an audience beyond experienced level designers. More study is needed to discover how well this system applies to a wider class of games. Finally, we propose exploring these ideas in other, non-game domains.

2.11 Acknowledgments

This work was supported by the University of Washington Center for Game Science, DARPA grant FA8750-11-2-0102, the Bill and Melinda Gates Foundation, the William and Flora

Hewlett Foundation, and an NSF Graduate Research Fellowship under Grant No. DGE-0718124.

Chapter 3

AUTOMATIC GAME PROGRESSION DESIGN THROUGH ANALYSIS OF SOLUTION FEATURES

The previous chapter introduced a novel system and user interface for designing progressions built on top of a formal domain model of a game's skills, using the educational math game *Refraction* for its example implementation. In this chapter, we explore taking the designer out of the loop entirely, and generating a full game and its progression automatically. The motivation for such a tool is being able to create games that adapt to the player; for example, with an educational math game, we are interested in targeting content towards player skills to maximize learning. We present a system for an adaptive, automatically generated version of *Refraction* we call *Infinite Refraction*, along with a in-the-wild user study suggesting that the generated game has comparable engagement to the original, hand-crafted one (though we still refer to the game as *Refraction* throughout the chapter). A large portion of the effort in building this system was spent creating the domain model used; the following chatpers address this challenge with a framework, RuleSy, for automatically creating models that could be used in applications like the one presented here. Portions of the chapter are reused from or based on prior technical publications [25].

3.1 Background

For many types of games, engagement is closely linked to the quality of the *level progression*, or how the game introduces new concepts and grows in complexity as the player progresses. Several game designers have written about the link between level progressions and player engagement. Game designer Daniel Cook claims that many players derive fun from "the act of mastering knowledge, skills and tools," and designs games by considering the sequence of

skills that the player masters throughout the game [33]. Others have written about Flow Theory [35] and the importance of engaging players by providing content that appropriately matches players' skill as it grows over time [8]. As the previous chapter established, producing game content with an effective progression is difficult to do and is typically done by hand. As a result, designers may be reluctant to revise their progression designs as new information about what players find interesting or challenging becomes available.

One goal of game design research is end-to-end automation of the game design process. We take a step towards this goal through a framework that reduces the need for explicit expertise in learner modeling. We aim to enable a different method of game design that shifts away from manual design of levels and level progressions, towards modeling the solution space and tweaking high-level parameters that control pacing and ordering of concepts. A key component of many games is the combination of basic concepts. What is interesting and engaging is not just the individual game rules and interactions, but how the progression combines them in increasingly complex ways. There is a combinatorial explosion in the number of ways to mix basic concepts, which, while allowing for many deep and interesting game experiences, makes creating progressions a challenge. Theories such as Vygotsky's zone of proximal development [137] motivate introduction of concepts at a controlled rate to allow the player time to master them. We propose that tracking and seeking mastery of combinations of basic elements of solutions is an effective way to organize and control the design space. In this setting, the learner model is defined with respect to the domain model. That is, the domain model represents the skills required to solve puzzles, while the learner model represents how well a player has mastered those skills.

Andersen et al. [3] proposed a theory to automatically estimate the difficulty of *procedural* problems by analyzing features of how these problems are solved. *Procedural* problems are those that can be solved by following a well-known solution procedure, such as solving integer division problems by hand using long division. The system treats these procedures as computer code and the human solvers as computers. Given such a procedure and set of problems, the system considers the code paths that a solver would follow when executing the

procedure for a particular problem, for example, how many times a particular loop must be executed or which branch of a conditional statement is taken. We call these paths *procedural traces*. Using this information as an estimate of difficulty, the framework specifies a partial ordering of these problems. While the previous framework can analyze existing progressions and generate problems, it cannot synthesize full progressions. This is mainly because Andersen's framework does not account for pacing. In contrast, our system allows a designer to control the rate of increase in complexity, the length of time spent reinforcing concepts before introducing new ones, and the frequency and order in which unrelated concepts are combined together to construct composite problems. Furthermore, Andersen's system can only handle highly-constrained procedural tasks. Games often have open-ended problem structures with many valid solutions and solution methods, necessitating an alternative approach.

In this chapter, we extend the ideas from procedural traces to automatically synthesize an entire progression for a popular puzzle game, *Refraction*. The game *Refraction* has no unique or preferred procedure for solving each puzzle. Therefore, we extend the partial ordering theory of procedural traces to work in a non-procedural domain. We present a system that controls the pace at which concepts are introduced to synthesize a large space of puzzles and the full progression for the game.

We evaluated whether this system can produce a game able to engage players in a real-world setting. We ran a study with 2,377 players and found no significant difference in time played compared to the original, expert-crafted version of the game. This original version found success and popularity on free game websites (played over one million times), and thus these empirical results suggest our framework captures important aspects of game progression design. Our long term goal is to extend this work to use data to optimize the progressions for each player.

3.2 System Overview

We first summarize the components of our system, then go into detail about each component in subsequent sections. Our system is built for the education math game *Refraction*, detailed

previously in Section 2.2.

In this chapter, we use the game-terminology *levels* to mean any completable chunk of content, such as an single puzzle in Sudoku, a math problem in a workbook, or a one of the stages in Nintendo's *Super Mario Bros.* For our application, *Refraction*, a level is an individual puzzle. A *level progression* is a sequence of levels that create an entire game.

In order to arrange a given set of levels into a progression, we need to be able to extract features from the levels that can be used to create an intelligent ordering. In this chapter, we propose the use of *solution features*, which are properties of the solution to a level. In *Refraction*, a solution is a particular board configuration, and the features are deduced from the structure of the board pieces in the solution. The techniques we describe, while containing game-specific components, can be applied to a variety of games.

Of course, before we can automatically arrange levels into a progression, we first have be able to create individual levels automatically. Creating levels is entirely game-specific, and we discuss the techniques used to create *Refraction* levels for our study.

Finally, the system must have a method for using the solution features to arrange levels into a progression. The method we present treats features indistinguishably and therefore applies generally to any game for which solution features are computed.

3.3 Extracting Solution Features

Andersen et al. [3] proposed the use of n-grams (sub-sequence of length n from a larger sequence) to abstract procedural traces, allowing the creation of a natural partial ordering on traces. In this section we describe how we extend this to extract features from non-procedural games such as *Refraction*.

3.3.1 Procedural Traces and n-grams

We begin with a review of traces and n-grams for procedural domains. Consider the algorithm for subtraction detailed in Algorithm 3.1. We indicate traces through this algorithm as

sequences of letters; these letters are output when the program executes commented lines in the above procedure. A few problems and their traces are shown in Table 3.1.

```
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:        p[i + j] := p[i + j] − 1
11:      a[i] := q[i] − p[i]
12:    else                                    ▷ Copy down (C)
13:      a[i] := q[i]
```

Figure 3.1: A procedural algorithm for subtraction, with labels at the key points. We can characterize the skills required to solve a problem by looking at the trace of labels when executing the procedure on that problem.

Subtraction Problem	**Trace**
$1 - 1$	DS
$11 - 11$	DSDS
$11 - 1$	DSDC
$11 - 2$	DSBDC
$101 - 2$	DSBZDCDC

Table 3.1: Example subtraction problems used in elementary-school mathematics and their corresponding traces.

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 concepts, and higher-value n-grams represent the sequential

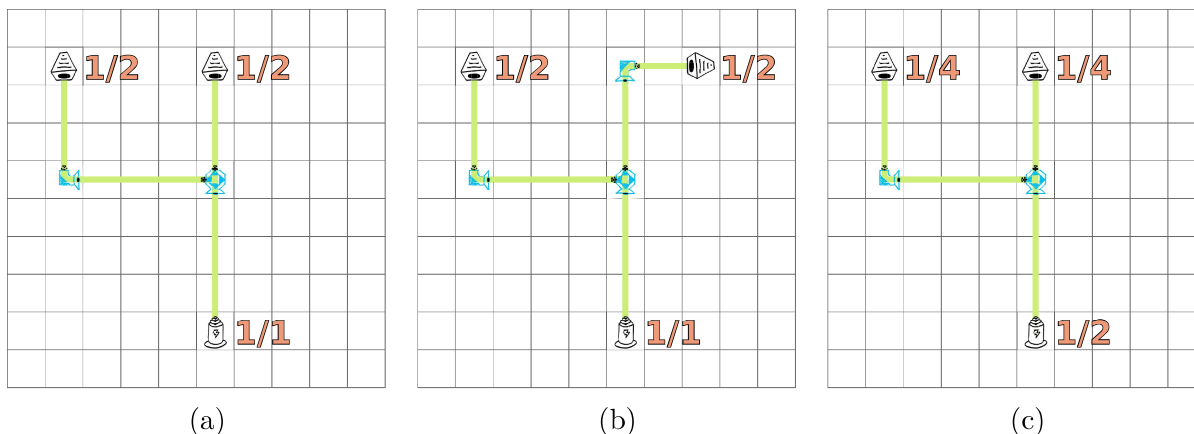


Figure 3.2: Three example levels for *Refraction*, to illustrate solution features and graphlets. When considering the *solution graph*, levels (a) and (b) have the same 1-graphlets. They differ in 2-graphlets: (a) has a splitter-target chain, while (b) does not. They differ on several 3-graphlets; notably, (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.

interaction, not captured by 1-grams, of using multiple concepts together. n-grams are intended as a method for computing comparable features for traces. We can represent a problem by the n-grams of its trace and use this to compare problems' relative complexity. We typically will only compute n-grams up to a small, fixed n, such as 3 or 4. The choice of n is a trade-off: a small maximum n will not capture salient interactions between concepts, but a large n will distinguish problems with only a slight difference in traces.

3.3.2 Solution Features in Refraction

Because *Refraction* does not have a preferred solution procedure, we cannot directly apply n-gram analysis. Rather than attempting to describe the procedure players use, we model features of solutions (henceforth, *solution features*), that players necessarily encounter. That is, for this application, the domain model is a set of solution features. As mentioned in the introduction of this dissertation, this is a simpler representation of domain models than

we will be generating in the following chapters. For now, we assume that every puzzle has a reference solution that is representative of all possible solutions. In the section on level generation, we describe how we can extract features even though *Refraction* puzzles often have many valid solutions.

As the players are forced to connect the pieces into a graph, one obvious source of solution features is the reference solution's laser graph, which we call the *solution graph*. As the analogue to n-grams for procedures, we use *graphlets*, which are small connected non-isomorphic induced subgraphs of a larger graph [105]. Figure 3.2 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.

While the solution graph captures many features of the solution, particularly the spatial components, our domain model includes additional graphs to capture other mechanics. For example, we can construct a graph whose nodes are the fraction values of the source and target pieces, where two nodes, source node s and target node t, are connected with an edge if, in the reference solution, s has a path to t. We select graphlets on both this auxiliary math graph and the solution graph. Although we do not claim to understand how the spatial and mathematical challenges interact when players solve the puzzle, by using multiple graphs we are able to capture those features we find relevant for progression design in sufficient depth.

3.4 Automatically Generating Levels

We applied the level generation process described by Smith et al. [116] (also used for generation in the tool presented in Chapter 2) to generate a large and diverse database of puzzles. In each puzzle, we enforced the constraint that all potential solutions to the level share the same solution graph. The generation process additionally produces a reference solution. Since all solutions are guaranteed to share the same graph, we can safely extract general solution features from the single reference solution produced by the generator. This analysis could

be extended to puzzles with multiple, significantly different solutions by considering which subset of features must show up and which optionally may be used to solve the problem, but we omitted this for sake of simplicity.

3.5 Creating a Progression

In this section, we describe how to take a set of levels (described by solution features) and create a full progression. We use the term *n-grams* in this section, but everything described applies equally to graphlets.

Given a diverse set of a problems and their traces, the solution features suggest a natural partial ordering of these problems, described by Andersen et al. [3].To summarize, given two levels L_1 and L_2, L_1 is considered conceptually simpler if, for some positive integer n, the set of n-grams of L_1 is a strict subset of the set of n-grams of L_2. Other approaches use data collected from players to determine the complexity relationships between problems (e.g., [40]). However, for spaces with a large set of possible traces, learning all these relationships from data or specifying them manually can be infeasible. Our technique uses the structure of the solutions to compute these relationships without data, which has been shown to correlate with users' perception of difficulty in the domain of an educational algebra game [3].

Our system must have a sequencing policy for creating a full progression, allowing the designer to control pacing and ordering. A simple approach would be to traverse the graph of partially-ordered problems in an order consistent with a topological sorting of the graph. This would ensure problems would be introduced before any of their (more complex) descendants. This approach has several drawbacks. It does not allow for control over the rate at which complexity increases or control over the balance between when new concepts (1-grams) are introduced versus combinations of concepts (2-grams and 3-grams). Moreover, this method would attempt to show every problem. Games cover a deliberately chosen subset of the entire (combinatorily large) space of possible problems, but this method lacks an intelligent way to decide which subset to use. As discussed, rather than trying to fully order a set of problems, we instead will use a large set of generated levels as a library from which to select

an appropriate progression.

Systematically Tracking and Introducing n-grams

To create progressions, we propose tracking players by estimating their mastery of concepts (i.e., knowledge components) identified by the n-grams found in level solutions and pairing this with a mastery-learning-based sequencing policy to teach content. That is, the system's goal is to systematically introduce each of the possible solution features. This *learner model* (or alternatively, *player model*) is defined with respect to the domain model. The learner model should assign, for each concept from the domain model, a measure of whether the student has had success over problems involving that concept. This, along with a method of quantifying the expected difficulty of future problems, gives us a basis for choosing the next problem to present. With our diverse set of labeled problems, the system will be able to carve nearly any possible path through this space, allowing the learner model to choose the most appropriate content at each stage. This ordering may be done offline to create a static progression, or online to create an adaptive game that responds to player performance. As it was not the focus of this work, we chose a simple method for modeling players in this system that does not reflect the state-of-the-art in student modeling. This framework allows for the insertion of more sophisticated learner models, such as Bayesian Knowledge Tracing [34].

In our system, the learner model tracks, for each component, whether the user successfully completed a problem containing that component. Note that we usually do not want to track *all* the concepts. As n increases, the number of n-grams tends to increase, sometimes very quickly. Therefore, it generally makes sense only to track all n-grams up to a small n.

The learner model is updated every time the player completes a problem. If the player is successful, then we mark as successful *all* of the n-grams contained within the problem that was just given. If the player fails, then it is less clear what to do. If we have no knowledge of how or why the player failed that might allow us to update the learner model, then the strictest option is to mark all components contained within the problem as unsuccessful. If we have more knowledge of where the player failed in the trace, we can update this

more accurately. If we know that the player failed on a specific 1-gram, then we mark as unsuccessful all n-grams containing that basic concept.

Choosing the Next Problem

Given a set of problems, the domain model, and a learner model, the framework must generate a sequence of problems for the player. We want this choice to respect the partial ordering determined from the n-grams, but allow the designer and learner model to control the pacing of content. The specific task is: given the current learner model state and a set of problems, we must choose the most appropriate next problem. Our progression generator builds progression one level at a time, choosing the next based on history of previous problems and data from player performance. We use a dynamic cost, which allows us to control the pacing and respect the ordering. The cost of a particular problem p is a weighted sum of the n-grams in the trace of p. These weights have 2 components: one based on what the learner model knows about the player, and one designer-specified weight.

First, at each point in the progression, for a given n-gram x, the learner model assigns a cost $k(x)$. This cost should be high for unencountered n-grams and low for ones already mastered, which will ensure more complex problems have a higher cost than simpler ones with respect to the player's history. This is used to respect the partial ordering of problems. The cost of a problem can intuitively be considered inversely proportional to the chance of the modeled player succesfully completing the problem using only their currently mastered knowledge.

The second component allows the designer control over pacing and relative order of basic concepts. The system treats concepts indistinguishably as opaque solution features. On the other hand, as expert designers, we know (possibly from data gathered during user tests) that particular concepts are more challenging than others or should otherwise appear later in the progression. For example, we may like to have some control over the ordering in which the system introduces 1-grams, or have the system give a higher cost to 1-grams than 2-grams, to prefer increasing complexity over adding new concepts. To this end, the second

component of our cost function is a designer specified weight $w(x)$, which can be used to influence the relative order of otherwise independent n-grams.

Thus, given a library of problems $\mathcal{P}$, choosing the next problem p_{next} consists of finding the problem with the closest cost to some target value T. This is shown in Equation 3.1. Intuitively, a minimum cost problem would be the easiest for the player to successfully complete, but it would not stretch their knowledge into new areas at all.

$$p_{\text{next}} = \arg\min_{p \in \mathcal{P}} \left| T - \sum_{x \in \text{ngrams}(p)} w(x)k(x) \right| \tag{3.1}$$

The target value and weighting functions can be based on the learner model. For example, an adaptive progression may change the target value based on how well it perceives the player to be performing. Note that (for any positive weight function) this cost function respects the partial ordering because for any levels $L_1 \prec L_2$, L_1 has a strict subset of the n-grams of L_2 and therefore less cost.

In order for this sequencing policy to be effective, it requires a large library of levels from which to choose. The levels should be diverse, covering the space of all interesting solution features. Furthermore, in order to enable effective control of pacing, this space should be dense enough that the progression can advance without large conceptual jumps. That is, for any level in the space, we want there to be a potential next level that adds few new n-grams. We accomplished this by ensuring that our level generation procedure attempted to create levels for every possible solution graph up to a particular size.

3.5.1 Level Selection in Refraction

During live gameplay, the library of levels embedded into the game is repeatedly consulted to determine (using the described cost function) which level to give to the player next. The graphlet-specific weights ($w(x)$ in the cost function) were largely left at a default value (1.0). It might seem strange to assign the feature of using a single bender the same weight as forming a chain of three benders in a row. However, graphlets have similar properties

of containment as n-grams: for example, every level that possesses the triple-bender-chain graphlet will also contain the double-bender-chain graphlet. Since our cost function respects the subset-based partial ordering (defined identically for graphlets as for n-grams), we can be certain that levels that combine independent concepts or explore embedded repetitions of those concepts have higher cost than the levels on which they build (causing them to occur later in generated progressions). We only adjusted the weight for four of the features, usually associated with graphlets of size 1. This very sparse advice allowed the progression system to know that, all else being equal, it is easier to introduce benders than splitters or that it is easier to introduce two-splitters than three-splitters.

Our player model tracked, for each graphlet, the number of levels the player successfully completed containing that graphlet, minus the number of unsuccessful attempts (but never less than 0). For each each graphlet x, call this value $c(x)$. We assigned cost $k(x) = 1/(1 + c(x))$, so that graphlets would move towards zero cost as they were successfully encountered.

One additional constraint we imposed on level selection in *Refraction* is that new levels were required, if possible, to include at least one graphlet that had $c(x) = 0$ (i.e., had either never been seen or had recent unsuccessful attempts). This guaranteed that the progression always chose something with an unmastered challenge: it would otherwise be possible (albeit unlikely) that the level with the closest cost to the target value is actually easier than the most-recently beaten level. Furthermore, in the interest of presenting the player with a series of interesting challenges, we ensure that no level is ever selected twice.

As the construction of auxiliary graphs, selection of graphlet features from them, and assignment of weights is an open-ended design task, we iterated on our choices for part of the system several times. The time to re-label a library of levels and re-build a version of the game using the new labels was under 30 seconds, allowing us to rapidly explore many different trajectories through our diverse library without re-running the level generator.

3.6 Evaluation

With our evaluation, we hope to demonstrate that this framework is capable of producing a game of comparable engagement to an expert-designed version. Specifically, we hope players would play for comparable amounts of time. In order to accomplish this, we generated an entire progression for an existing successful game, *Refraction*, and performed a between-participants experiment against the original version of the game. The original's puzzles were hand-crafted by game designers. We believe that the original version of the game serves as a fair comparison against which to test the generated version. The game found a large amount of success (being played over a million times since its release) and relies heavily on the quality of the puzzles. Therefore, even though it is prone to bias, we argue the puzzle design in the original version can be considered to be of high enough quality that it serves as a reasonable target for our automated system.

We deployed both versions of *Refraction* and performed a between-participants experiment to measure engagement. We posted the game on a popular Flash game website, Newgrounds[1], as well as though MochiMedia[2], which distributes through the main MochiGames[3] 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 version driven by our framework and 1,156 players played a version that included the exact, expert-designed level progression from the original *Refraction*.

When deciding what to measure, we are interested in actual player behavior rather than player opinions, so we measure player performance directly. We performed our test "in the wild," without notifying players that they were part of an experiment. We did not collect any personally identifiable information from players. Because we did not ask players directly,

---

[1]www.newgrounds.com

[2]www.mochimedia.com

[3]www.mochigames.com

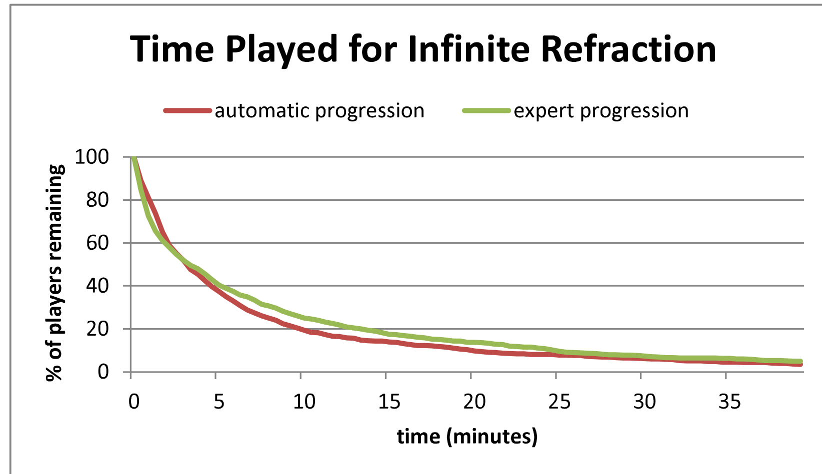


Figure 3.3: Comparison of our automatically-generated progression with the expert-generated progression from *Refraction*. The x-axis is time in minutes and the y-axis is the percentage of players who played for at least that much time. The median values are very similar: approximately 3 minutes. Although our framework's progression performs slightly worse, it is comparable to the expert progression. These results suggest that our framework was able to create an experience that engages players approximately as long as one crafted by hand with design expertise. In contrast to the original design process, which included many hours of crafting and organizing puzzles by hand, our framework 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 designer-tuned weights that control the order and speed of introduction of these units.

we must rely on proxy metrics to estimate engagement. One of the primary metrics previous "in the wild" experiments have used to estimate engagement is total play time [4, 5, 85], which we use here. Because the platforms for this experiment were websites with thousands of free games that players could be choosing to play instead of *Refraction*, getting players to spend a significant amount of time on a game suggests that the game is at least somewhat engaging.

We first performed a series of statistical tests on the distribution of time played in each condition. Our data was not normally distributed, so we relied on a nonparametric test, the Wilcoxon-Kruskal-Wallis test. Despite having over one thousand players in each condition, the 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. In an effort to quantify this small difference we observed, we looked at other engagement metrics, particularly the number of puzzles completed in both groups. We found a small but significant difference, $r = 0.13$, $p < 0.001$, with players playing the automatically generated progression completing a median of 8 puzzles versus 10 puzzles completed for the original, expert-designed progression. Since the puzzles are not the same between conditions, they are not directly comparable, so we feel that this is not the best measure of engagement. The median time is 8% lower for the automatically generated progression than the human-authored one. Figure 3.3 visualizes this data in greater detail. We can see from this graph that the automatically generated progression is able to retain players at rates qualitatively and quantitatively similiar to the original game. Though we can draw no definite conclusions from this lack of difference, the results do suggest that the game automatically produced by our framework was capable of engaging players for a comparable length of time to a hand-crated, expert-designed version.

While our experiment has a condition that represents a reasonable upper bound on progression quality (the human-authored progression), one major limitation in our experiment is the lack of a condition to represent a lower bound on progression quality. With our current results, we cannot be sure the progression had any effect on player engagement at all. In a followup experiment, a 3rd condition with a completely random progression should be added to the experiment. If, in this hypothetical experiment, the random condition was not significantly worse than the human-designed or automatically-generated progressions, it would suggest that perhaps progressions have little effect on that set of players. On the other hand, if random progressions were significantly worse than the other two conditions, it would suggest that progressions do have some effect on player engagement, and furthermore, that our automatic techniques were capturing at least some aspects of engaging progression design.

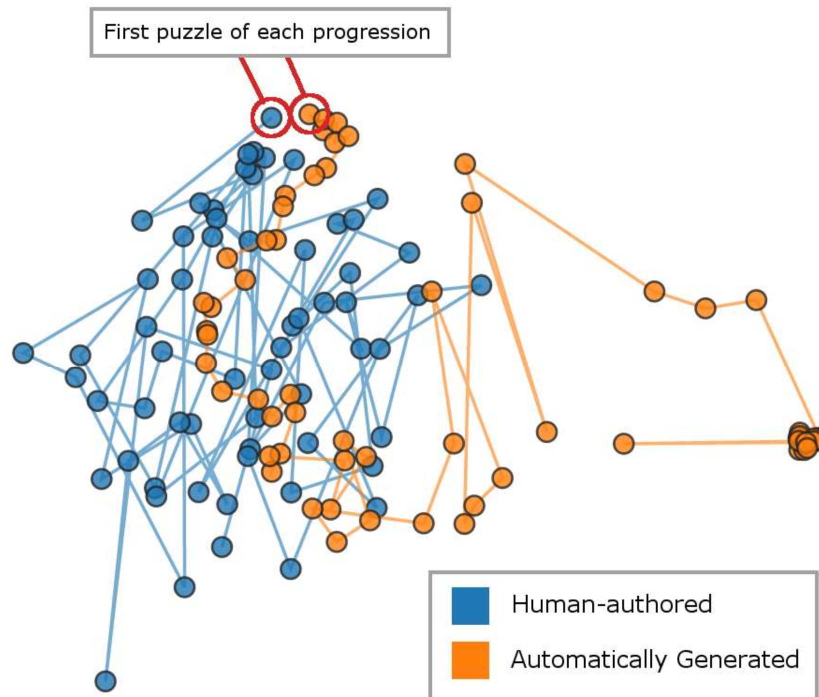


Figure 3.4: Visualization of the first 61 levels in the two progressions in our experiment, the human-authored (blue) and automatically genreated (orange). Nodes represent levels and edges are drawn between consecutive levels. Levels are represented as feature vectors of concepts and projected into two dimensions using Multidimensional Scaling. Nodes that are close together represent puzzles that are conceptually similar according to our solution features.

3.6.1 Difference Between Progressions

Figure 3.4 shows a visualization of the first 61 puzzles of each progression in our experiment, created using a tool by Butler and Banerjee [26]. To compute the visualization, levels are represented as a feature vector of concepts (one for each n-gram) and projected into two dimensions using Multidimensional Scaling [70]. Thus the visualization shows how different levels are from each other in the conceptual space determined by our solution features. We assume that this distance (i.e., Euclidean norm) in the feature space captures at least some information about the conceptual difference between two puzzles that players would perceive. Because these features are based on how a puzzle must be solved, there is enough meaning

to at least note some qualititative differences between the two progressions.

One major difference is that our automatically generated progression takes much smaller steps in this conceptual space than the human-authored one. Another difference we can see is that the automatically generated progression has several puzzles clustered together. This is because, at that point in the progression, our system had run out of new concepts to introduce. Our cost model then tends to choose very similar levels over and over. This exposes a limitation in our technique; by contrast, the human progression often makes large conceptual jumps and never stays in the same spot for an extended period of time. If we wished to more closely mimic the human progression, we could try to slow the rate at which concepts are introduced or add features to the cost model that penalized choosing consecutive levels that were too close in the conceptual space.

3.6.2 Discussion

This work presents a key building block towards data-driven automatic optimization of game progressions. It allows us to automatically create a progression that is comparable to an expert progression with minimal manual tuning. Because it exposes key high-level control knobs (e.g., rate at which concepts are introduced, when to favor introducing new concepts over combinations), data-driven optimization could further refine these parameters, which we hope to explore in a future study.

Due to the time and difficulty in implementing this system for a particular game, we expect that applying this technique to a particular game is often more laborious that manual progression design. It is theoretically possible that the automated method saves hundreds of hours of effort by avoiding having to manually craft puzzles. On the other hand, designers likely have to create and test many different levels to gain enough understanding to choose good features, and the cost of creating a level generator is often non-trivial. Unlike the previous trace-based framework that only needs a solution procedure, this method requires enough insight into the mechanics to create a reasonable labeling function that can extract features. For us, when the end-goal is data-driven optimization of game progressions, the

benefits are apparent. Other promising applications may include all-in-one game creation tools such as PuzzleScript,[4] especially those with a constrained language for expressing rules and mechanics. In such tools, the extent of possible rules and mechanics is (somewhat) known for most or all games created within the tool. Therefore, the labeling function and other parts required for system could be included in the tool and take much less effort to apply to any particular game. However, if the goal is merely to produce a good game with a reasonable level progression, it is unclear that this system would be beneficial, even for domains well-suited to this approach.

We believe this approach could be applied to games whose content centers around a progression of concrete skills or procedural knowledge, such as puzzle games and educational games. It currently requires a game designer to provide a breakdown of the individual *skill components* of each puzzle, and thus games for which this breakdown cannot be reasonably produced will not work well with this system. Games that primarily rely on aspects other than the level progression for engagement will also likely not benefit from this system.

We address the particular challenge of attempting to (semi) automatically discover these skill components with the RuleSy framework, presented from the following chapter onward. Though not instantiated for this particular game, this framework is designed precisely to aid in rapidly creating and exploring formal models of human skill in skill-based games or educational domains such as *Refraction*. Using this framework may ease the creation and deployment of systems like the one presented in this chapter.

3.7 Related Work

3.7.1 Human Computer Interaction

Previous work in HCI has employed a variety of methods to increase engagement in games. Some have used theoretically-grounded or interview-based approaches. For example, Linehan et al. [82] draw on ideas from behavioral psychology to provide a series of guidelines for

[4] http://www.puzzlescript.net/

designing effective educational games. Isbister et al. [58] interviewed design experts to gain insights for creating learning games. Also providing guidance and opportunities to game developers, our work offers automatic methods which can allow designers to iterate on their designs as they seek to engage players.

Others have attempted to optimize engagement or learning through large-scale design experiments and data-driven methods. Andersen et al. measured the effectiveness of tutorials [5] and aesthetics [4] through A/B tests, examining metrics such as time played, levels played, and return rate. Lomas et al. [85] ran a large experiment that explored several design dimensions of an educational game to maximize engagement. Harpstead et al. [54] demonstrate a toolkit and method for analyzing player learning in games based on replaying players' games. Our apporach allows designers to tweak high-level parameters and quickly produce machine-generated playable games, which could pair with these data-driven apporaches to more efficiently compare alternative designs. Bauckhage et al. [14] looked at large datasets of how long players played several games and developed techniques to analyze player data and draw conclusions about player engagement. In this work, we do only a simple statistical comparison between our experimental conditions, though a more sophisticated analysis could be performed in future work.

3.7.2 Intelligent Tutoring Systems

Intelligent Tutoring Systems researchers have explored several models for capturing student knowledge and selecting problems [39]. Cognitive Tutors, such as Knowledge Tracing tutors [34], model student knowledge in terms of knowledge components: the underlying facts, principles, or skills a student puts to use in larger problem-solving tasks. They predict from student data whether students know particular concepts or have particular misconceptions. In contrast, systems using Knowledge Space Theory [41] use only observable actions. Our system uses a simple model of observable variables, though it can be extended to integrate more sophisticated models. Many of these tutors use *mastery learning* when choosing content; they give students repeated problems on a particular concept until the model has 95%

confidence in their mastery, then move to the next concept [68].

A key part of progression generation is ordering problems appropriately. While many systems require experts to manually order content, several researchers have explored automatically learning the relationships between concepts, typically from user data [40]. Our work concerns capturing the effects of composite concepts; for example, if X and Y are concepts, a problem requiring X then Y has a composite concept XY. Liu et al. introduces a technique to learn relationships of composite concepts [83]. Standard models can capture composite concepts (e.g., XY) by modeling them as separate concepts, using prerequisite relationships to preserve ordering.

In games, there are often non-linear dependencies between base skills and skill combinations. For example, a player might be able to handle X or Y independently but struggles when doing them together. Therefore, consideration of concept combinations is crucial both for verifying mastery and also for driving the increase in complexity. However, if all combinations of base skills are modeled as an independent skill, this space grows exponentially large. Additionally, analysis of this joint conceptual space is often reduced to someone manually selecting combinations of skills to treat as standalone concepts. Therefore, a structured and automatic way of sampling this space is desirable.

Our system, rather than depending on a pre-existing database of user data or hand-designed relationships, determines the relationships from the structure of the solutions to problems, both for procedural and some non-procedural domains. Stern et al. describe a system that sequences problems using a scoring function of concepts and their prerequisites [125]. We build on this work in our sequencing policy. Another important aspect of creating progressions is controlling pacing. Beck et al. [15] explore choosing problems of appropriate difficulty by ranking problems based on the number of required skills and the student model's prediction of proficiency at those skills. We take a similar approach when choosing appropriate problems in our framework.

We do not directly innovate ITS methods in our work, nor do we use the most sophisticated ones. Instead, we seek to demonstrate that a framework inspired by ITS principles,

when combined with some additional machinery to drive the growth of complexity and reasonable parameter settings, can produce engaging game content that is comparable to content produced by an expert game designer. Researchers have previously looked into bringing ITS techniques into educational games [42]. The primary difficulty stems from an open-ended problem nature in most games that does not easily lend itself to ITS-based structures that are geared towards mastery of concepts rather than a large space of creative recombinations of concepts and skills required to solve an explorative problem. We address this in our framework by focusing on the ability to use combinations of concepts to reach a solution.

3.7.3 Games

Content for games has traditionally consisted of hand-crafted progressions and levels. Game design researchers have explored procedural content generation to automatically create games. Several systems aim to automatically generate game levels. To measure the quality or diversity of generated content, many approaches measure static properties of the content [80] or affect of players [138]. Other approaches focus on what the player must do, measuring properties related to player action. For example, Launchpad characterizes levels by the rhythm of the actions the player must perform [122]. We generalize such ideas; for example, actions in a platform game can be viewed as segments of a trace of a procedure for solving platformer levels.

Generated levels are not useful in isolation; they must be sequenced into some design-relevant order, called a *progression*. A good game progression is critical for game design [33]. For generated games, if only a small number of global parameters need to be controlled, the problem can be treated as dynamic difficulty adjustment, such as in the game Polymorph [62]. However, our domains have many independent conceptual building blocks, and adjusting a few parameters fails to capture the many different dimensions by which levels create difficulty. In the platform game Endless Web [122], the player explores a large design space of levels that are classified based on various features. The player makes choices that influence what levels

they will receive. Perhaps the closest work to our approach is *Square Logic*[5] by Everyday Genius, a Sudoku-style puzzle game that features 20,000 automatically constructed puzzles that are arranged into a progression based on which logical inference rules are required to solve them. Few details of the techniques used are available, but one difference with our problem is that *Refraction* does not have a clear set of inference rules used to form solutions. Further, many *Refraction* levels have more than one possible solution (though these solutions will often share high-level structure and concepts).

Several researchers have proposed mixed-initiative editors in which the system and human designer take turns producing the content [120, 80]. These systems were limited to the creation of single levels. Butler et al. [27] proposed a mixed-initiative system for end-to-end game production allowing for manual designer intervention at all stages of content generation. By contrast, the present work borrows inspiration from ITS to drive conceptual growth automatically, reducing the need for expertise in the form of manual authoring. Using our system, the designer does not manually edit levels or the progression on a per-level basis, but instead controls them through high-level parameters that describe the overall goals of the progression. These parameters allow the designer to make sweeping, purposeful changes to the progression.

3.8 Conclusion

We presented a framework that automates level progression design by analyzing features of solutions. This framework borrowed ideas from intelligent tutors to enable a method of design where content and progression are fully generated. We applied these ideas to create an implementation for the educational game *Refraction*. Our study with 2,377 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 that can engage players for a comparative length of time as content designed by hand.

---

[5] http://www.squarelogicgame.com/

Future work may include further exploring integrating technologies of intelligent tutoring systems in this framework. Our implementations used a simple learner model built on a simple domain model, but one could explore using technology from cognitive tutors and other models to drive the progression. While our system can order problems based on the combination of concepts, it cannot distinguish individual concepts (the 1-grams) and must resort to designer-specified weights or arbitrary decisions. Although the parameter settings we tried for *Refraction* were inspired by the original progression, we do not have a deep understanding of why they were effective. We hope that automatic methods (for example, the one presented in the following chapter) for learning these settings will increase the designer's ability to optimize a game for engagement. While previous work showed that ordering problems using n-grams correlated with user perception of difficulty, further studies should be performed to understand this relationship. We speculate that difficulties keeping the user engaged in ITS [59, 16] may be alleviated by deeper exploration of complexity, and the solution-trace methods described in this paper may help to accomplish this through automatic control and tracking of concept combinations.

3.9 Acknowledgements

This work was supported by the Office of Naval Research grant N00014-12-C-0158, the Bill and Melinda Gates Foundation grant OPP1031488, the Hewlett Foundation grant 2012-8161, Adobe, and Microsoft.

Chapter 4

A FRAMEWORK FOR AUTOMATICALLY LEARNING DOMAIN MODELS

In this chapter, we describe RuleSy, the general framework for automatic generation of procedural knowledge. This procedural knowledge takes the form of a formal model for solving problems in some problem domain, which, borrowing terminology from the intelligent tutoring, we call the *domain model* [67]. One major component of this model is a set of *rules* to use in problem solving. We discuss the motivation behind and manner of structuring these rules in a programming language and define *domain models* more precisely. We then discuss the program synthesis-powered framework structure, illustrated on an example domain of learning proof rules for propositional logic proofs [20]. This chapter provides a high-level overview of RuleSy; two instantiations of this framework for two different domains are described in detail in the following two chapters.

4.1 Domain Models by Example: Propositional Logic Proofs

We begin by describing the toy domain that will be used to illustrate the framework throughout the chapter: semantic proofs of the validity of propositional formulae [20]. Problems in this domain are propositional formulae such as $(p \wedge q) \rightarrow (p \rightarrow q)$. Formulae are given meaning by an interpretation (usually named I) which maps variables to truth values. For a given interpretation I and formula F, We write $I \vDash F$ to say that F is true under I (I satisfies F), and $I \nvDash F$ to say that F is false under I (I falsifies F). For example, with $I = \{p \mapsto \text{true}, q \mapsto \text{false}\}$, we have that $I \vDash p \vee q$.

The goal of a semantic proof problem is to prove whether a given formula is *valid*—that is is true under all interpretations. For a given a formula F, we prove its validity by preceding

with the assumption that a falsifying interpretation of F exists and working towards a contradiction. We proceed to use proof rules (whose axioms are defined in Table 4.1) to derive facts either until a contradiction is reached or no more facts can be derived. A contradiction proves the formula valid while failure to find a contradiction after exhaustive application of proof rules provides a falsifying interpretation. An example solution for the above problem is shown in Figure 4.1.

Assume $(p \wedge q) \rightarrow (p \rightarrow q)$ invalid.

1.	$I \not\models (p \wedge q) \rightarrow (p \rightarrow q)$	assumption
2.	$I \models p \wedge q$	by 1 and semantics of $\rightarrow$
3.	$I \not\models p \rightarrow q$	by 1 and semantics of $\rightarrow$
4.	$I \models q$	by 2 and semantics of $\wedge$
5.	$I \not\models q$	by 3 and semantics of $\rightarrow$
6.	$I \models \bot$	4 and 5 are contradictory

Figure 4.1: An example solution of a semantic proof problem. The $\bot$ symbol is used to notate that we have arrived at a contradiction (literally, the interpretation satisfies falsehood).

The axioms of Table 4.1 fully define the domain of propositional logic: any solvable problem can be solved using only those axioms (and, moreover, the full set is necessary to find counterexamples to validity). However, in practice, solutions can be dramatically shortened through the use of additional proof rules such as *modus ponens* (Table 4.2). As illustrated in Figures 4.2 and 4.3, a solution without using modus ponens requires branching into multiple cases, arriving at a contradiction in each. The solution with modus ponens is much shorter.

The *domain model* design task RuleSy aims to solve can be viewed as the same design task of a hypothetical curricula designer: what procedural knowledge needs to be taught in order to allow a student to solve problems in this domain? In this domain, clearly, the proof rules are a key component of that knowledge. There is more beyond that; the proof rules must be paired with a proof search strategy. However, for the purposes of this thesis, we assume the search strategy is given, and that the task of choosing a domain model is one

ID	Name	Description
p	Contradiction	If $I \vDash A$ and $I \nvDash A$ then $I \vDash \bot$
q	Branch elimination	If $I \vDash \bot \mid A$ then $I \vDash A$
r	And 1	If $I \vDash A \wedge _$ then $I \vDash A$
s	And 2	If $I \nvDash A \wedge B$ then $I \nvDash A \mid I \nvDash B$
t	Or 1	If $I \vDash A \vee B$ then $I \vDash A \mid I \vDash B$
u	Or 2	If $I \nvDash A \vee _$ then $I \nvDash A$
v	Not 1	If $I \vDash \neg A$ then $I \nvDash A$
w	Not 2	If $I \nvDash \neg A$ then $I \vDash A$
x	Implication 1	If $I \vDash A \rightarrow B$, then $I \nvDash A \mid I \vDash B$
y	Implication 2	If $I \nvDash A \rightarrow B$, then $I \vDash A$
z	Implication 3	If $I \nvDash A \rightarrow B$, then $I \nvDash B$

Table 4.1: The axioms [20] that define the domain of semantic proofs for propositional logic.

Name	Description
Modus Ponens	If $I \vDash A$ and $I \vDash A \rightarrow B$, then $I \vDash B$

Table 4.2: The modus ponens proof rule, which, while unnecessary, can greatly simplify proofs.

of choosing a set of proof rules. For this domain, we start with the axioms to define the domain, and RuleSy will generate additional useful proof rules like modus ponens.

There are several challenges to overcome to solve this task. First, *how do we represent these proof rules?* Clearly we'll need a formal representation for the computer to find, but, critically, this representation must be interpretable by humans. This goes beyond simply making rules understandable: the representation is trying to model the solving processes of human experts. We represent proof rules as programs in a domain-specific language of strategies, described in the next section. Next, *where do additional strategies like modus ponens come from, and how can we mechanically explore that space?* In the case of modus ponens, the rule is a combination of three axioms applied in succession (Figure 4.4). Once we have some kind of idea where to search, *how do we generate rules in the appropriate*

Assume $(p \rightarrow (q \rightarrow r)) \rightarrow (q \rightarrow (p \rightarrow r))$ invalid for sake of contradiction.

1.	$I \not\models (p \rightarrow (q \rightarrow r)) \rightarrow (q \rightarrow (p \rightarrow r))$	assumption
2.	$I \models p \rightarrow (q \rightarrow r)$	by 1 and semantics of $\rightarrow$
3.	$I \not\models q \rightarrow (p \rightarrow r)$	by 1 and semantics of $\rightarrow$
4.	$I \models q$	by 3 and semantics of $\rightarrow$
5.	$I \not\models p \rightarrow r$	by 3 and semantics of $\rightarrow$
6.	$I \models p$	by 5 and semantics of $\rightarrow$
7.	$I \not\models r$	by 5 and semantics of $\rightarrow$

We now must consider two branches of the proof.

8a.	$I \not\models p$	by 2 and semantics of $\rightarrow$
9a.	$I \models \bot$	6 and 9a are contradictory

In the other branch:

10b.	$I \models q \rightarrow r$	by 2 and semantics of $\rightarrow$

We must immediately branch again:

11ba.	$I \not\models q$	by 10b and semantics of $\rightarrow$
12ba.	$I \models \bot$	4 and 11ba are contradictory

And on the other branch:

13bb.	$I \models r$	by 10b and semantics of $\rightarrow$
14bb.	$I \models \bot$	7 and 13bb are contradictory

Having arrived at a contradiction on every branch, we can eliminate all branches and conclude the formula valid.

Figure 4.2: A semantic proof problem that requires branching multiple times.

representation?. As we'll see, we frame this as a programming synthesis problem, where rules are represented by programs. Finally, *what makes strategies like modus ponens a useful addition?* Clearly there are many possible ways to combine axioms. What makes this one worthy of adding to a curriculum? In part, this has to do with the facts that the rule is concise to describe but also practically useful on problems we care about, and the challenge is to formalize this notion. Our contributions in RuleSy address each of these questions with the three phases of our strategy learning framework discussed in Section 4.4.

As we will see in Chapter 5, this problem domain fits precisely into the framing of *axioms* and *tactics* used when learning strategies for solving algebra equations. The framework used

1.	$I \not\models (p \to (q \to r)) \to (q \to (p \to r))$	assumption
2.	$I \models p \to (q \to r)$	by 1 and semantics of $\to$
3.	$I \not\models q \to (p \to r)$	by 1 and semantics of $\to$
4.	$I \models q$	by 3 and semantics of $\to$
5.	$I \not\models p \to r$	by 3 and semantics of $\to$
6.	$I \models p$	by 5 and semantics of $\to$
7.	$I \not\models r$	by 5 and semantics of $\to$
8.	$I \models q \to r$	by 2, 6, and modus ponens
9.	$I \models r$	by 8, 4, and modus ponens
10.	$I \models \bot$	7 and 9 are contradictory

Figure 4.3: Solution to the same problem as Figure 4.2, but using modus ponens. This dramatically simplifies the proof, indicating that modus ponens is a good candidate to include in any domain model for semantic proofs.

Consider a proof state where both $I \models p \to q$ and $I \models p$ hold. We can apply three axiomatic proof steps that together are equivalent to the modus ponens rule:
1. By the semantics of $\to$, either $I \not\models p$ or $I \models q$. Our state now contains two new branches, b_1 (where $I \not\models p$) and b_2 (where $I \models q$). 2. In branch b_1, $I \models p$ and $I \not\models p$ are contradictory, meaning $I \models \bot$. 3. We can eliminate branch b_1, leaving us with b_2 where $I \models q$.

Figure 4.4: An illustration of the three axioms that comprise the modus ponens macro. By searching over potential combinations of axioms, RuleSy finds specifications for potential macro rules.

for this algebra domain was also instantiated for the semantic proof domain and was able to generate the modus ponens rule. In other domains, like the Nonograms domain we will see in Chapter 6, the structure of strategies and their relation to the formal description of the problem domain can be quite different.

4.2 Mathematical Preliminaries

Before jumping into formal details, let us introduce some mathematical notation we will use for the remainder of this thesis.

Tuples. We use the notation $\langle x, y \rangle$ to denote a pair of elements x and y. For any $p = \langle x, y \rangle$, we define $\mathit{fst}(p) = x$ and $\mathit{snd}(p) = y$ (the first and second elements of the pair, respectively).

Sequences. We use the notation $S = [x_1, \ldots, x_n]$ to denote that S is an ordered sequence of n values. We use $|S| = n$ to denote the length of S. For any positive integer $i \leq |S|$, we use $S_i = x_i$ or $S[i] = x_i$ (depending on which is easier to read in context) to denote the ith (1-indexed) value of S. We use $S = (x_i)_{i=1}^{n}$ as an alternative notation for the sequence S, particularly when defining one sequence in terms of another. The notation $[]$ denotes the empty sequence.

Other Notation. We use $\bot$ to represent the "bottom value." When used in a function definition, e.g., $f(x) = \bot$, we mean that f is a partial function and that f is not defined on x. The $\bot$ notation is also used to represent falsehood, such as showing that a semantic proof has arrived at a contradiction.

We use the $[\![\cdot]\!]$ notion when defining semantic functions that give meaning to program syntax. For instance, when defining a semantic function $\mathcal{F}$ which maps programs and inputs to values, we use the notation $\mathcal{F}[\![P]\!]x = e$, where P is program syntax, x is the input, and e is the result value.

When defining semantic functions, we use the notation $\doteq$ to mean equality on values. This is to distinguish it from the equality symbol used for function definition and in the examples of program semantics. The $\doteq$ symbol has higher preference than the $=$ symbol. For example, $\mathcal{F}[\![P]\!]x = x \doteq 2$ means that program P on input x evaluates to true if x equals 2 and false otherwise.

4.3 Procedural Knowledge as a Set of Condition-Action Programs

Before describing the RuleSy framework built to address these challenges, we must discuss the first challenge of how to represent proof rules or, more generally, how to represent procedural knowledge (which we call the *domain model*) for a variety of domains.

4.3.1 Problem States and the Solution Algorithm

We begin by more formally defining the solution algorithm for semantic proofs. At its core, the solution is a search through a state space of proof trees. The states are comprised of a set of deduced facts (e.g., $I \vDash p \rightarrow q$). They are trees because the proofs might branch (like in Figure 4.2) and each branch has a different set of facts. A goal state is one in which all branches have been eliminated through contradiction, or one for which no more axiomatic proof rules apply to a particular branch.

The solution process involves applying proof rules in succession (in any order) until a goal state is reached. There are two components of the process: (1) the set of available proof rules, which is the focus of RuleSy; and (2) the algorithm for choosing rules (described above), which we assume is given.[1] Thus, the task of RuleSy will be to choose this set of proof rules; when we say *domain model* we specifically mean the set of strategies used in a search-based problem-solving process. In the domain of semantic proofs, these are proof rules. We will later cover solving algebraic equations and solving logic puzzles, where the meaning of strategies changes, but idea of domain model-as-set of strategies still holds.

Particular proof rules can apply in multiple ways to a single state, as illustrated by Figure 4.5. Thus, they are most naturally thought of as a (non-functional) relation between states, or, alternatively, a function that returns the set of possible applications of a rule (which is how we frame *fire*, below). Such a representation makes both synthesizing and reasoning about rules a challenge; we would much prefer rules to be functions. As we describe below, our DSL achieves this by factoring out information about "where" a rule is applied as a separate input—which we call the *binding*—in order to make rule programs functions. This will prove critical in making synthesis tractable while preserving soundness guarantees. The framing as a non-functional relation between states will still prove useful

[1] The structure of the algorithm is an obvious part of formally defining the problem domain (e.g., applying rules until you hit the goal state), so taking that as given is a reasonable assumption. There are additional important design decisions (e.g., "how do we know when it's a good idea to apply proof rule X?") that are beyond the scope of this thesis but in the realm of short-term future work.

when analyzing the soundness and generality of our synthesized rules.

Given a state where
$I \models p$, and $I \models p \rightarrow q$, and $I \models p \rightarrow r$:
Modus ponens can be used to deduce:
$I \models q$
but also:
$I \models r$

Figure 4.5: An example state where the modus ponens rules applies in multiple ways, each on different parts of the state. That rules can apply multiple ways is a challenge our framework must overcome to synthesize rules.

Patterns, Conditions, Actions

We formally represent the strategies (e.g., proof rules) that comprise the domain model as programs in a domain-specific language (DSL) comprised of three parts: patterns, conditions, and actions. For example, a representation of modus ponens in our proof rule DSL is shown in Figure 4.6. The *pattern* describes *where* a proof rule applies. In the case of modus ponens, it applies to any proof branch where the facts include $I \models A \rightarrow B$ and $I \models C$, where A, B, and C are arbitrary formulae. The *condition* describes *when* a proof rule applies by adding more constraints to the pattern. For modus ponens, A and C must be syntactically identical for the rule to apply. The *action* describes *how* a proof rule applies. Modus ponens adds $I \models B$ to the set of derived facts. In the case of semantic proofs, patterns apply nondeterministically: a proof rule might be applied in multiple ways to the same state, either to different proof branches or to different facts within a particular branch. In order to make programs functions rather than non-functional relations, we capture this nondeterminism with the concept of a *binding*, a mapping between the elements of a pattern and some *substate* of the problem state to which the proof rule applies. The condition and action are defined with respect to the pattern: patterns (partially) map states and bindings to some *context*, which maps pattern variables to the matched parts of the problem state. Conditions and actions are

deterministic functions of this context, meaning they can only reference the pattern rather than the problem state directly. Thus, rule programs are partial functions from states and bindings to states.

```
1 def modus_ponens:
2     with s@(facts (I⊨ a) (I⊨ (⇒ b c)) etc):
3     if a = b:
4     do replace s with (I⊨ c) :: s
```

Figure 4.6: The modus ponens rule, represented in our DSL for rules. Line 2 is the pattern, line 3 is the condition, and line 4 is the action. We describe details of the syntax and semantics of this language in the following chapter, in Section 5.3.

Though the representation of states, bindings, patterns, conditions, and actions change across domains, the overall structure of the RuleSy is fixed. Figure 4.7 shows the schema for semantics of the meta-language and a description of the domain-specific elements, and Figure 4.8 shows the parts of these semantics that are the same for all domains. The *fire* function (defined over these semantics) captures the intuitive notion of applying a rule in every possible way: it is a function from program-state pairs to the set of all possible output states that arise from applying the program across all possible bindings. This is useful for applying rules in practice to solve problems. Defining the RuleSy DSL for a given domain requires providing representations for states, bindings, and contexts; semantics for patterns, conditions, and actions; and a method of enumerating over all possible bindings.[2]

Figure 4.9 shows a general solving algorithm built on top of *fire*. This algorithm performs backtracking search through the state space by applying tactics to exhaustion. The particular search algorithm to use (e.g., depth-first search vs. breadth-first search) depends on the problem domain. For example, semantic proofs would use depth-first search, and in fact

[2]This *bindings* function isn't strictly necessary; it can be the trivial constant function that returns the set of *all* bindings and *fire* would remain sound. But a tighter bound makes *fire* (and hence problem solving and synthesis) more efficient, and *bindings* can be easily defined for our example domains to produce a reasonably tight bound of plausible bindings.

Program semantics	:	$\mathit{program} \times \text{state} \times \text{binding} \rightarrow (\text{state} \cup \{\bot\})$
$\mathcal{P}$	:	$\mathit{pattern} \times \text{state} \times \text{binding} \rightarrow (\text{context} \cup \{\bot\})$
		Partial function from states and bindings to contexts
$\mathcal{C}$	:	$\mathit{condition} \times \text{context} \rightarrow \text{boolean}$
		Predicate on contexts
$\mathcal{A}$	:	$\mathit{action} \times \text{state} \times \text{context} \rightarrow \text{state}$
		Function from states and contexts to states
$\mathit{fire}(P, s)$	:	Set of states resulting from all applications of P to s.
bindings	:	$\text{program} \times \text{state} \rightarrow 2^{\text{binding}}$
		function from programs and states to a (super) set
		of matching bindings

Figure 4.7: The schema for the semantics of the RuleSy meta language, including the types of the holes that each instantiation of the framework needs to provide: the semantics for patterns, conditions, and actions (illustrated here as semantic functions $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{A}$ respectively); and a method of enumerating potential bindings of a pattern for any state. The representations of states, bindings, and contexts are domain-specific. The overall semantics of programs is that programs are partial functions from states and bindings to states

does not need to backtrack since a goal state is by definition reached when we run out of rules to apply. A method for mechanically solving problems with a given set of tactics will prove necessary for multiple of RuleSy's algorithms.

Both domains treated in detail in Chapters 5 and 6 have search-based solution algorithms, where patterns apply nondeterministically. Even for domains where greedy or procedural algorithms are used (e.g., elementary arithmetic), the subprocedures of such algorithms can be modeled as pattern-condition-action rules with trivial patterns or conditions (e.g., "this rule applies at step k"). In this way, RuleSy can be viewed as a generalization of prior work on modeling procedural solution processes (e.g., [3]). The rules synthesized by RuleSy, along with the general solving procedure of Figure 4.9 can be used in applications ranging from analyzing problem difficulty to generating progressions of problems.

The domain-specific details that must be implemented for each instantiation of RuleSy correspond roughly to the concepts of that domain. For example, in semantic proof prob-

$$
\begin{aligned}
[\![\texttt{with}\ p\texttt{:}\ \texttt{if}\ c\texttt{:}\ \texttt{do}\ a]\!](s,\beta) &= \textbf{if}\ \sigma \neq \bot \wedge \mathcal{C}[\![c]\!]\sigma\ \textbf{then}\ \mathcal{A}[\![a]\!](s,\sigma)\ \textbf{else}\ \bot \\
&\quad\ \textbf{where}\ \ \sigma = \mathcal{P}[\![p]\!](s,\beta) \\
\mathit{fire}(P,s) &= \{[\![P]\!](s,\sigma) \mid \sigma \in \mathit{bindings}(P,s), [\![P]\!](s,\sigma) \neq \bot\}
\end{aligned}
$$

Figure 4.8: The shared semantics of the RuleSy meta language. The top-level semantics of programs are the same for all instantiations of RuleSy. Namely, given a state and a binding, if the pattern matches the state under the binding and the condition applies, then the action is applied to that state. The *fire* function also has the same definition for all domains.

function SuccessorStates(s: problem state, $\mathcal{M}$: set of rule programs)
  $\mathcal{S} \leftarrow \{\}$
  for all $p \in \mathcal{M}$ **do**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathit{fire}(p, s)$
  return $\mathcal{S}$

function Solve(s: starting state, $\mathcal{M}$: set of rule programs, goals: state $\rightarrow$ boolean)
  return Search(s, SuccessorStates, goals)

Figure 4.9: A general solving algorithm for problems. Given a starting state, a set of RuleSy rules, a goal predicate on states, Solve returns the goal states reachable from the starting state by applying *fire* on the given rules. The Search function depends on the domain: it takes a starting state, transition function, and goal predicate and returns all reachable goal states.

lems, the patterns reference propositional logic, satisfaction/falsification, and proof branches; conditions use boolean arithmetic; and actions define a new derived fact to be added to the state. Clearly, the design of the DSL is a critical component in the success of RuleSy in any specific domain. We discuss this challenge in further detail in Chapter 7.

4.3.2 Motivation for the Structure of Rules

Our approach to learning interpretable strategies by representing them as pattern-condition-action rules defined over domain-specific concepts is motivated by cognitive psychology and education research. In this approach, a set of rules represents (a part of) domain-specific

procedural knowledge of the problem domain—the strategies a person takes when solving problems. Such domain-specific knowledge is crucial for expert problem solving in a variety of domains [45, 21], from math to chess to professional activities. The DSL defines the (domain-specific) concepts and objects to which the learner can refer when solving puzzles, thus constraining the space of strategies that can be learned to human-friendly ones. Our system takes this space as input provided by a designer, and produces the procedural knowledge to be used by people. In that sense, a human designer trying to create a domain model with RuleSy works iteratively with the framework, with the designer refining the concepts and RuleSy deriving solution procedures with respect to those concepts. The designer also provides a cost function mapping programs to numbers (defined over the syntax of the DSL) that used as a proxy for the complexity of rules, which allows our system to bias the learning process toward concise, interpretable rules. The eventual goal of this line of research is automatically discovering strategies that people are likely to use. In this thesis, we focus on the immediate task of finding human-interpretable rules in a structure compatible with evidence of how humans behave.

4.4 Synthesizing Procedural Knowledge as Programs

This section illustrates the key steps of RuleSy's architecture and its inputs and outputs, as shown in Figure 4.10. Having established how domain models will be represented, RuleSy's algorithms are designed to solve the remaining three core technical challenges outlined earlier in the chapter: (1) *specification mining*, using a formal specification of the domain and some example problems to derive specifications for potential rules; (2) *program synthesis*, synthesizing sound and maximally general programs according to those rule specifications; and (3) *domain model optimization*, choosing a useful subset of the synthesized programs that optimize a given objective function. We discuss each in turn.

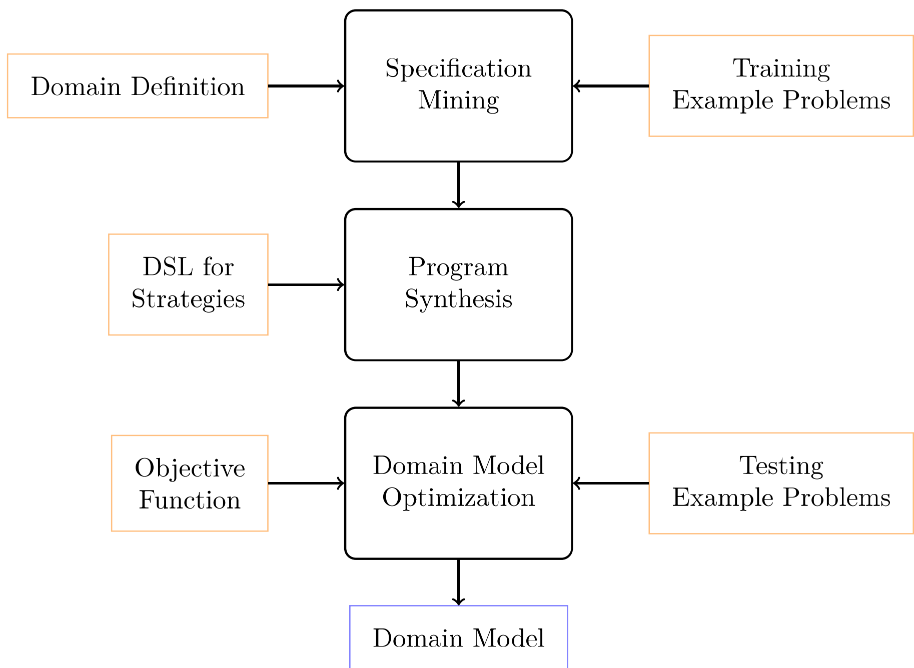


Figure 4.10: The system architecture of the RuleSy framework, consisting of three phases: specification mining, program synthesis, and domain model optimization.

4.4.1 Specification Mining

Synthesizing interpretable strategies requires specifications of the behavior for *sound* and *useful* rules. RuleSy derives these specifications from a formal definition of the domain and training example problems. Soundness is defined by the domain description. In the case of proof rules, the natural way to specify soundness is via *axiomatization*, the rules of which are shown in Table 4.1. More generally, characterizing soundness for a domain entails describing a set of valid transitions through the state space. A specification is sound if it describes a subset of this transition space.

When we say rules should be useful, we mean that the synthesized rules should enable more efficient problem solving than without. In the case of proof rules, we're looking for rules such as modus ponens which can solve problems in fewer steps than with the axioms alone. Such rules, which we call *tactics*, are a composition of two or more axioms. Usefulness can be

quantified in multiple ways: first, rules should apply to problems the framework user cares about: RuleSy determines which rules to synthesize, and how those rules should behave, by mining tactic specifications from solutions to a given set of example training problems. Any specification generated by RuleSy will improve solving efficiency of at least one of these problems.

Second, rules should be general, in that they apply to wide set of states. There is a trade-off here: we want rules that are as general as possible but also as simple as possible, and the two are in direct conflict. Indeed, the provided domain definition is by construction the most general possible rule, but also incredibly expensive to apply. At the other end of the spectrum, rules that map a single input state to a single output state are very simple but not at all general. In the specification mining phase, RuleSy mines (and in later phases learns programs for) a wide variety of rules at various points in this space, leaving it up to the domain model optimization phase to navigate these trade-offs.

In the domain of semantic proofs, we mine specifications by first solving the training examples using the axioms, focusing specifically on shortest solutions. RuleSy looks at subpaths of these solutions as potential specifications for tactics. Note that the training examples are *not* used for soundness, which instead is defined entirely by the domain definition. Thus, RuleSy guarantees soundness of results regardless of the training examples given. Changing the training examples changes the usefulness of the synthesized rules, but they will always be correct rules.[3]

4.4.2 Rule Synthesis

Given a tactic specification, the next challenge is to find a rule that implements it. We discussed the representation in Section 4.3.1. The rule synthesis phases takes this DSL and a mined specification from the specification mining phase and finds a program that soundly represents this specification using an algorithm built on standard synthesis techniques. In

[3]In contrast to, say, a programming-by-example system such as PROSE [104] which guarantees correctness only with respect to the examples themselves.

the case of proof rules, these proof rule programs act on inputs of unbounded size, so we face the additional challenge (discussed in Chapter 5) that the rule synthesis query cannot be expressed in existing systems for syntax-guided synthesis (e.g., [124, 133, 1]). The example problem for modus ponens output by this system in shown in Figure 4.6.

The key technical challenges this phase must solve, beyond finding sound rules, is to ensure the rules are general and concise. For all domains, we want the most concise possible representation of a rule to minimize the cognitive load required for a human to use it. Conciseness is measured by the cost function (over programs) provided by the designer, which, for our implementations, is a function of the size of the syntax of the rule. In the domain of proof rules, the way RuleSy does specification mining produces a complete specification of behavior, making generality relatively straightforward to guarantee. However, as we'll see in Chapter 6, that is not the case for all domains, and finding general rules is a core challenge of the Nonograms domain.

4.4.3 Rule Set Optimization

The final challenge is to search the synthesized strategies for a domain model that both solves the examples and optimizes the input objective. The rule synthesis phase produces (potentially) a strategy for each mined specification, and thus potentially massive set of rules must be pruned down to a useful core set. In general, this phase involves selecting a subset of synthesized strategies that maximize a given objective function. In our implementations, the objective function we use is one that balances a conciseness with generality; we both want the smallest set of strategies possible (to minimize how much a student would need to learn) but also the most generality (to maximize how many problems a student can efficiently solve).

In the case of proof rules, this set of strategies potentially includes the axioms (indeed, the correctness of the semantic proof relies on all axioms being in the domain model). So any domain model produced by RuleSy would include all the axioms and some set of synthesized tactics. A proof rule like modus ponens is both very concise and very general, and RuleSy

will select such a rule under our objective function.

4.5 Conclusion

In this chapter, we presented an overview of the RuleSy framework, describing how it models the problem of learning interpretable rules as a problem of program synthesis. Given as input a formal domain description, a DSL for rules, example problems, and an educational objective function, RuleSy automatically learns programs in this DSL that describe sound and useful rules. In the following two chapters, we present detailed descriptions of two instantiations of RuleSy for different domains.

Chapter 5

LEARNING DOMAIN MODELS FOR ALGEBRA AND TREE REWRITE DOMAINS

The first domain we explore for RuleSy is solving algebraic equations of the kind one may find in high school math curricula. This is a special case of the general problem of finding rewrite rules between semantically equivalent expressions represented as abstract syntax trees. The semantic proof domain explored in the previous chapter also falls under this category, and we implemented instantiations of RuleSy for both concrete domains. While we focus on these domains due to our educational motivations, this problem is a fundamental one that encompasses applications such as finding optimization rules for compilers. We expect portions of our contributions to be applicable more broadly. Portions of the chapter are reused from or based on prior technical publications [30, 28].

5.1 Background

A key challenge in the design of educational applications is modeling the operational knowledge that captures the expertise for a given domain. This knowledge takes the form of the *domain model* described in the previous chapter, consisting of *pattern-condition-action* rules that experts apply to solve problems in the domain. For example, factoring, $ax + bx \rightarrow (a + b)x$, is one such rule for K-12 algebra: its condition recognizes problem states that trigger rule application, and the action specifies the result. Educational applications rely on domain models to automate tasks such as problem and progression generation [3], hint and feedback generation [100], student modeling [6], and misconception detection [136].

At present, domain models are created by hand, taking hundreds of hours of development time to model a single hour of instructional material [94]. This limits applications to using

one out of many possible models that capture the operational knowledge for a domain. Yet recent research [84] shows that some students need over six times more content than others to master a domain. To best serve a broad population of students, applications therefore need multiple models that optimize different educational objectives [37, 106].

In the case of algebra, these pattern-condition-action rules take the form of rewrite rules on trees. A valid algebra rule is one that rewrites an algebra term into a semantically equivalent term. A solution to an algebraic equation is a sequence of such rewrites until the variable is isolated, alone, on one side. The semantic proof domain (from the previous chapter) has a very similar framing: rewrite rules move between logically equivalent states, and the goal is to either arrive at a contradiction or get to a state where no more rules apply.

In both cases, the problem state is a tree with some kind of semantic meaning—which we call *terms*—and the overall problem is to find a semantically equivalent term with a particular property (e.g., isolated variable). Due largely to the existence of variables in terms, finding such equivalences algorithmically is a challenge: in the case of propositional logic, this problem is NP-hard (it is literally SAT), and, in the case of algebra over non-linear integer arithmetic, it is undecidable.

Thus in practice, educators teach students to use a set of greedy rules (like the factoring rule described above) to solve some interesting class of problems. In the case of semantic proofs this covers all problems, but for algebra it is a subset of possible problems. The domain-model authoring problem is one of creating and then deciding which rules to teach students. To illustrate the difficulty of model authoring, consider creating a domain model for K-12 algebra. Each rule represents a mapping between equivalent states, and these rules can be chained together to solve a problem, as in the example below:

$$3x + 2x = 10 \rightarrow (3 + 2)x = 10 \rightarrow 5x = 10 \rightarrow (5x)/5 = 10/5 \rightarrow x = 2$$

A given set of basic rules can solve all the problems educators care about for an algebra curriculum. In this thesis we call these primitive rules *axioms*: rules like factoring or combining

constants.

However, not all rules are as useful as others. We can solve the same problem using fewer rules that combine steps together:

$$3x + 2x = 10 \rightarrow 5x = 10 \rightarrow x = 2$$

This example of moving straight from $3x + 2x$ to $5x$ is so useful and common in problems that it is taught in many algebra curricula as the *combine like terms* rule (e.g., [31]). The rule of combining like terms is what we call a *tactic*—a rule that combines two or more axioms in a particular way. The primary design question in domain model authoring is: which axioms or tactics should we teach to students? On one hand, combine like terms isn't any more expressive than factoring and adding constants are, in that applies the two of them in sequence. Since educators are probably going to need to teach factoring and adding constants anyway, adding in a new rule adds more work for students even though they could hypothetically solve the exact same problems without it. On the other hand, combine like terms substantially simplifies the solution for a large set of problems, making it easier (in that is requires less effort and fewer steps) for students to solve some of the problems that we care about. This idea of solution efficiency can, for example, have a significant impact on the cognitive load required to solve problems and thus impacts learning gains [129]. How a curriculum designer decides which rules to include is based on educational objective criteria such as minimizing the amount they need to teach to students or minimizing the cognitive load required to solve a given set of problems. And, before they decide which tactics to use, they need to figure out what tactics are even possible and how to concisely define them.

This chapter presents a new, computer-aided approach to help tackle these challenges. We realize this approach in RuleSy, a framework, based on program synthesis, that creates tactics and domain models that optimize desired objectives. The RuleSy framework was motivated by practical experience: the author works for Enlearn, an educational technology company building adaptive K12 applications that need custom domain models. Developers

of such applications are the intended users of this work, and Enlearn is in the process of adopting key ideas from RuleSy.

In the algebra domain, RuleSy must rely on the user (e.g., a hypothetical curriculum designer) to define the domain with a set of axiom rules. These axioms define a space of semantically equivalent terms. Give a set of training problems, RuleSy automatically searches for tactic rules that are sound with respect to the axioms and that can be used solve the training examples more efficiently than the axioms alone. RuleSy then selects a subset of the axioms and generated tactics to form a domain model by optimizing given objective criteria.

Our domains of focus were chosen because we are motivated by educational goals. However, the problem of finding rewrites between equivalent terms with some kind of underlying semantic meaning (usually with variables in them) covers a broad class of important and interesting problems. One such problem is program optimization: the goal of an optimizing compiler is to find a program that is semantically equivalent to the input and that has a desirable property (is smaller, runs faster, etc.). This, as with algebra, is impossible to do algorithmically since program equivalence is undecidable for sufficiently complex programming languages. Practical compilers therefore use rewrite rules. Though we do not explore this application in further detail, we expect many of the ideas and contributions presented here to be applicable to a broader range of problems than educational domains alone.

To evaluate our algorithms, we used RuleSy to model the domain of introductory K-12 algebra, comparing the output to a standard textbook [31]. Applying RuleSy to examples and axioms from the textbook, we find that it both recovers the tactics presented in the book and discovers new ones. We also find that RuleSy can recover the textbook's domain model, as well as discover variants that optimize different objectives.

To show that RuleSy generalizes beyond K-12 algebra, we used it to model the domain of propositional logic proofs presented in Chapter 4. Applying RuleSy to textbook [20] axioms and exercises, we find that it synthesizes both new and standard tactics for this domain (e.g., modus ponens), just as it did for K-12 algebra.

5.2 Overview

This section illustrates RuleSy's functionality on a toy algebra domain. We show the specifications, tactics, and models that RuleSy creates for this domain, given a set of example problems, axioms, and an objective.

5.2.1 Examples, Axioms, and Objectives

Figure 5.1 shows our example problems, axioms, and objective for toy algebra.

Examples. The problems (5.1b) are represented as Lisp-like syntax trees. We consider a tiny subset of algebra that includes equations of the form $x + \sum_i c_i = c_k$, where x is a variable and c_i, c_k are integer constants. Experts solve these problems by applying rewrite rules until they obtain an equation of the form $x = c$.

Axioms. The axioms (5.1a) are expressed as programs in the RuleSy language (Section 5.3). A rule program consists of a *pattern*, which matches a syntax tree with a specific shape (up to isomorhpism of commutative operators); a *condition*, which further constrains when a rule may apply; and an *action*, which creates a new tree by applying editing operations (such as replacing or removing nodes) to the matched tree. For example, the axiom **A** matches trees of the form `(+ 0` e ...`)`, where the order of subtrees is ignored, and it rewrites such trees by removing the constant 0 to produce `(+` e ...`)`. The shown axioms can solve all problems in the toy algebra domain.[1] For example, we can solve p_1 in two steps by applying $\mathbf{B} \circ \mathbf{A}$ to obtain $x + 1 + -1 = 5 \rightarrow_{\mathbf{B}} x + 0 = 5 \rightarrow_{\mathbf{A}} x = 5$. RuleSy uses the axioms to synthesize tactic rules (Figure 5.3) that can solve the example problems in fewer steps.

Objective. The educational objective (5.1c) is expressed as a function of rule and solution costs. Rule cost measures the complexity of a rule's syntactic representation. Solution

---

[1] Along with some axioms—omitted here for brevity and clarity—that make transformations such as pruning unary addition or applying associativity rules. These axioms are listed in Section 5.11.

cost measures the efficiency of a solution in terms of the tree edits performed to solve an example problem. These costs are proxy measures for the difficulty of learning and applying knowledge in a given domain model [129]. RuleSy selects a domain model that best balances the trade-off between rule complexity and solution efficiency specified by the objective.

5.2.2 Specifications, Tactics, and Domain Models

Specifications. To help with domain modeling, RuleSy first needs to synthesize a set of useful tactics, which can solve the input problems more efficiently than the axioms alone. For example, we could solve p_1 in one step if we had a "cancelling opposite constants" tactic that composes the axioms **B** and **A**. RuleSy determines which rules to synthesize, and how those rules should behave, by mining tactic specifications (Section 5.5.1) from the shortest solutions to the example problems that are obtainable with the axioms (Figure 5.2a).

A RuleSy specification takes the form of a *plan* for applying a sequence of axioms (Figure 5.2b). A plan describes which axioms to apply, in what order, and how. Since an axiom may be applied to a problem in multiple ways, a plan associates each axiom the binding for the axiom's application, consisting of an application index and a permutation. The application index identifies a subtree in the expression's abstract syntax tree (AST), and the permutation specifies a mapping from the index space of the axiom's subtree to the index space of the pattern. The plan in Figure 5.2b specifies a generic tactic for cancelling opposite constants; for example, it solves p_1 in one step by reducing the expression `(+ x 1 -1)` to `(+ x)` (Figure 5.2c). In essence, plans are functional specifications of tactics that can help solve the example problems in fewer steps—and that are amenable to sound and efficient synthesis.

Tactics. Given a set of plans, RuleSy synthesizes the corresponding tactics (Section 5.5.2), expressed in the same language (Section 5.3) as the input axioms. Figure 5.3 shows two sample tactics synthesized for the plans (e.g., Figure 5.2b) mined from the toy examples and axioms. These tactics perform fewer tree edits than the axiom sequences they replace, leading

```
def A: # additive identity (+0e ...) → (+e ...)
    with (+ a@num _ etc):
    if a = 0:
    do remove a

def B: # constant folding (+c1 c2 ...) → (+c ...), c = c1 + c2
    with (+ a@num b@num _ etc):
    if true:
    do remove b and replace a with apply(+ a b)

def C: # adding opposites (= (+e0 ...) e1) → (= (+(−e0) e0 ...) (+e1 (−e0)))
    with (= L@(+ a etc) R@_):
    if true:
    do replace L with (- a) :: L and
       replace R with (+ (- a) R)
```

(a) Axioms in the RuleSy language (Section 5.3).

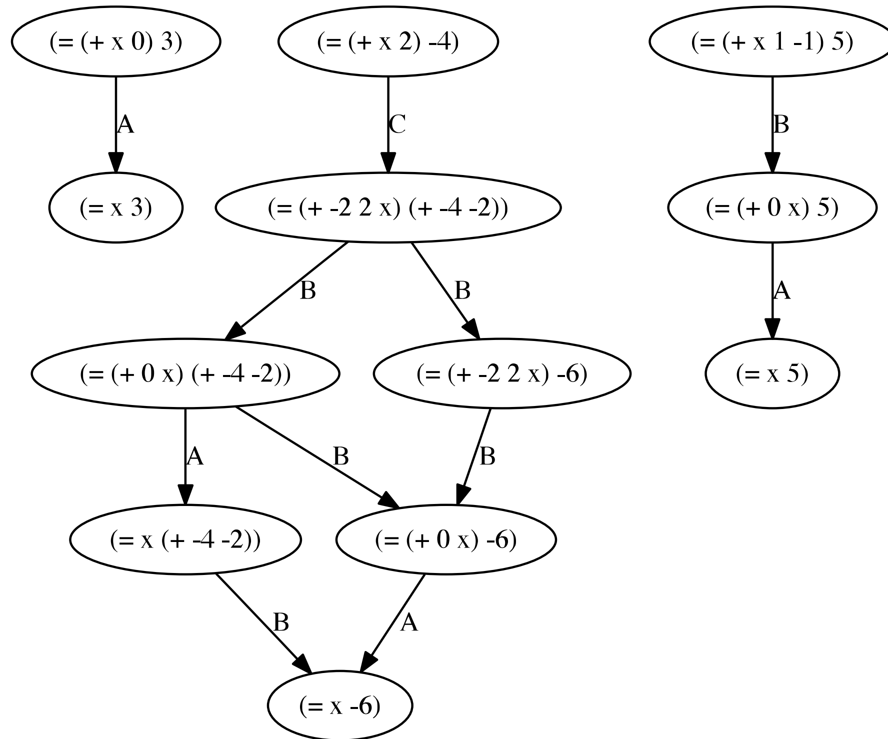
Problem	Representation
$p_0 : x + 0 = 3$	`(= (+ x 0) 3)`
$p_1 : x + 1 + -1 = 5$	`(= (+ x 1 -1) 5)`
$p_2 : x + 2 = -4$	`(= (+ x 2) -4)`

(b) Example problems.

$$f(\mathcal{R}, \mathcal{G}) = \alpha \sum_{R \in \mathcal{R}} \mathit{RuleCost}(R) + (1 - \alpha) \frac{\sum_{G \in \mathcal{G}} \mathit{SolCost}(G)}{|\mathcal{G}|}$$

(c) A sample objective function, where $\alpha \in [0, 1]$, and $\mathcal{G}$ contains the shortest solutions to the example problems obtained with the rules $\mathcal{R}$.

Figure 5.1: The inputs to RuleSy for the toy algebra domain.



(a) All shortest solutions for the toy algebra problems and axioms (Figure 5.1).

$$[\langle \mathbf{B}, \langle [], \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3], [3] \mapsto [1]\}\rangle\rangle,$$
$$\langle \mathbf{A}, \langle [], \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [1]\}\rangle\rangle]$$

(b) A plan for applying the axioms $\mathbf{B} \circ \mathbf{A}$.

Input	Axiom	Output
(= (+ x 1 -1) 5)	**B**	(= (+ x 0) 5)
(= (+ x 0) 5)	**A**	(= (+ x) 5)

(c) Using the plan in (b) to solve the problem p_1 (Figure 5.1b).

Figure 5.2: A sample plan (b) mined from the shortest solutions (a) to the toy algebra problems. The plan specifies a tactic for canceling opposite constants (c).

to cheaper solutions. For example, the tactic **BA** performs two tree edits, while the axiom sequence $\mathbf{B} \circ \mathbf{A}$ performs three such edits. But the tactic also applies to fewer problem states than the axioms. RuleSy uses discrete optimization, guided by the input objective, to decide which axioms and tactics to include in a domain model.

```
# Canceling opposite constants: (+c −c e...) → (+ e ...).
define BA:
    with (+ x@num y@num _ etc):
    if x = apply(- y)
    do remove x and remove y

# Move negated constant to other side with only one other term:
# (= (+ c1 e ...) c2) → (= (+ e ...) c), c = c2 − c1 .
define CBAB:
    with (= (+ x@num _ etc) y@num):
    if true:
    do remove x and
       replace y with apply(- y x)
```

Figure 5.3: Sample tactics synthesized for the toy plans (e.g., Figure 5.2b).

Domain Models. The RuleSy optimizer (Section 5.5.3) searches the axioms and tactics for a domain model that is sufficient to solve the example problems, while minimizing the objective over all shortest solutions obtainable with such models. Figure 5.4 shows two sample optimal models for the toy algebra domain. The models $\mathcal{R}_{0.1}$ and $\mathcal{R}_{0.9}$ minimize the toy objective (Figure 5.1c) for different values of the weighting factor α (0.1 and 0.9, respectively). The model $\mathcal{R}_{0.1}$ includes more rules because lower values of α emphasize solution efficiency over domain model economy. RuleSy helps with rapid navigation of such design tradeoffs.

$$\mathcal{R}_{0.1} = \{\mathbf{A}, \mathbf{BA}, \mathbf{CBAB}\} \quad \mathcal{R}_{0.9} = \{\mathbf{BA}, \mathbf{CBAB}\}$$

Figure 5.4: Optimal domain models for toy algebra (Figures 5.1 and 5.3).

5.3 A Domain-Specific Language for Tree Rewrite Rules

This section presents the RuleSy domain-specific language (DSL) for specifying pattern-condition-action rules in the algebra domain. The language is parametric in its definition of problem states. For concreteness, we present an instantiation of RuleSy for the domain of K-12 algebra. Section 5.6 generalizes this presentation to other domains, particularly the semantic proof domain described in Chapter 4.

5.3.1 Specifying Problems and Rules.

RuleSy for algebra represents problems and problem states as abstract syntax trees, which we call *terms* (Definition 5.1, Figure 5.5). These trees have algebraic operators at every internal node and variables or constants at the leaf nodes.[2] In algebra, constants range over integers, and variables range over symbolic literals that are equal up to alpha renaming. The definition of terms specifies which operators are commutative; RuleSy assumes that that two terms are semantically equivalent if they are equal up to reordering of commutative operators (Definition 5.3). As we shall see, rewrite rules are designed to abstract over commutativity, and the solving procedure searches for ways to apply rules under any possible permutation of the term.

$$
\begin{array}{rcl}
\textit{term} & ::= & \texttt{(}\textit{op}\ \ \{\textit{term}\}^{*}\texttt{)} \mid \textit{value} \\
\textit{value} & ::= & \text{numeric literal} \mid \text{variable identifier} \\
\textit{op} & ::= & \textit{commutativeop} \mid \texttt{-} \mid \texttt{/} \\
\textit{commutativeop} & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{=}
\end{array}
$$

Figure 5.5: Syntax for states in the algebra DSL, which we call *terms*. The notation $\{\textit{form}\}^*$ means zero or more repetitions of the given form. RuleSy relies on operators being partitioned by commutativity.

[2] Note that, as defined, term syntax admits nonsensical terms such as `(+ 2 (= 7 (* 1) 8))`. RuleSy does not reason about the validity of terms in this sense, and the guarantees of the framework are with respect to valid terms.

Rules in the algebra domain can be viewed as tree rewrite rules between semantically equivalent terms. Patterns describe a (partial) structure that an input term must match. Conditions specify further constraints on the subterms of the input. Actions are sequences of term editing operations, namely, removing or replacing a subterm. The toy problems (Figure 5.1b) and rules (Figures 5.1a and 5.3) from Section 5.2 are all valid terms and programs in the algebra DSL.

Definition 5.1 (Terms)**.** A *term* is a tree with operators on internal nodes and variables or constants at the leaf nodes that defines a problem state (Figure 5.5), where the set of operators is partitioned into commutative and non-commutative operators. For presentation, we sometimes write $\mathit{term}(o, v_1, \ldots, v_n)$ to mean the term $\texttt{(}o\ \ v_1\ \ \texttt{...}\ \ v_n\texttt{)}$.

Definition 5.2 (Tree Indices)**.** A *tree index* is a finite sequence of positive integers that identifies a subterm of a term as follows:

$$\begin{array}{lcl} \mathit{ref}(t, []) & = & t; \\ \mathit{ref}(\texttt{(}o\ \ t_1 \ldots t_k\texttt{)}, [i]) & = & t_i \text{ if } 1 \leq i \leq k; \\ \mathit{ref}(\texttt{(}o\ \ t_1 \ldots t_k\texttt{)}, [i, j, \ldots]) & = & \mathit{ref}(t_i, [j, \ldots]) \text{ if } 1 \leq i \leq k; \\ \mathit{ref}(t, \mathit{idx}) & = & \bot \text{ otherwise.} \end{array}$$

That is, each element of a tree index represents child indices for successively deeper parts of a tree. We write $\mathit{refs}(t)$ for the set $\{\mathit{idx} \,|\, \mathit{ref}(t, \mathit{idx}) \neq \bot\}$. That is, $\mathit{refs}(t)$ is the set of all tree indices on which ref is defined.

Definition 5.3 (Tree Permutations)**.** Let t, t^π be terms where $|\mathit{refs}(t)| = |\mathit{refs}(t^\pi)|$. Let π be a bijective mapping from $\mathit{refs}(t)$ to $\mathit{refs}(t^\pi)$. This function π is a *tree permutation* for t if it preserves structure (parent-child relationships are preserved) and respects commutativity (reorders only children of commutative operators). That is, for all $[i_1, \ldots, i_n] \in \mathit{refs}(t)$ where $n \geq 1$, if $\pi([i_1, \ldots, i_n]) = [j_1, \ldots, j_n]$, then $\pi([i_1, \ldots, i_{n-1}]) = [j_1, \ldots, j_{n-1}]$, and either $i_n = j_n$ or $\mathit{ref}(t, [i_1, \ldots, i_{n-1}]) = \texttt{(}\mathit{op}\ \ \texttt{...)}$ where op is commutative. We write $\Pi(t)$ to denote the set of all tree permutations of t, and t^π to denote the term obtained by applying π to t. That is, the term where $\mathit{refs}(t^\pi) = \{\pi(\mathit{idx}) \mid \mathit{idx} \in \mathit{refs}(t)\}$ and $\forall \mathit{idx} \in \mathit{refs}(t).\ \mathit{ref}(t, \mathit{idx}) = \mathit{ref}(t^\pi, \pi(\mathit{idx}))$.

The *identity permutation* for a term t is the identity mapping from $refs(t)$ to itself.

As a notational convenience, for any function ϕ with a domain that is a superset of $refs(t)$ and where the restriction $\phi\restriction_{refs(t)}$ of ϕ to the domain of $refs(t)$ is a tree permutation for t, we write t^{ϕ} to mean $t^{\phi\restriction_{refs(t)}}$. Thus, we use the identity function I to mean the identity permutation, using this notational shorthand to implicitly mean $I\restriction_{refs(t)}$.

Definition 5.4 (Term Isomorphisms). A tree permutation is an isomorphism: if there exists a tree permutation between terms s and t, then s and t are semantically equivalent since they are equal up to reordering of commutative operators. We say s and t are *isomorphic*, written $s \simeq t$.[3]

5.3.2 Representation of Bindings and the Semantics of Rule Firing

In considering how rules apply to problem states, we must consider both applying rules to various subterms and applying rules with respect to commutative operators in the domain. To illustrate, consider firing the rule **A** from Figure 5.1a on the term $t =$ `(= (+ x 0) 2)` (we will return to this example throughout the section). **A** is defined to apply to a term with `+` at the root and a `0` as the immediate first child of the root. Applying **A** to t requires both specifying a subterm of t and a permutation of t's commutative operators (in this case to addition) that align t with **A**'s pattern. In that sense, **A** is a functional template for a more general relation that may apply **A** in multiple ways to the same state. In order to keep rules as functions while allowing for multiple applications, we factor out this information about how a rule is applied in a second input to the program, called a *binding*.[4] We formalize this definition of a binding below. This definition relies on the definition of patterns. Patterns are defined in more detail later in this section; their syntax is shown in Figure 5.6. Briefly,

[3]When we say two terms are isomorphic, we are specifically limiting ourselves to say they are permutations of each other, not that they are semantically equivalent in general.

[4]One might ask why we are are introducing machinery to permute the pattern to match the term rather than the (arguably more natural) approach of having *fire* take care of such permutations itself. While this would keep the rule semantics simpler, our approach better supports specification mining: in order to capture the behavior of a sequence of axioms, we also need to capture information about the permutations used to make that sequence work. The binding does precisely that.

patterns describe a subtree to be matched against a term.

Definition 5.5 (Scopes)**.** Scopes describe when tree indices are valid for patterns. A tree index *idx* is in the *scope* of a pattern p if $\mathit{scope}(p, \mathit{idx}) \neq \bot$ where:

$$
\begin{aligned}
&\mathit{scope}(p, []) &&= p;\\
&\mathit{scope}(a\texttt{@}p, i) &&= \mathit{scope}(p, i);\\
&\mathit{scope}(\texttt{(}o\ \ p_1 \ldots p_k\texttt{)}, [i]) &&= p_i \text{ if } 1 \leq i \leq k;\\
&\mathit{scope}(\texttt{(}o\ \ p_1 \ldots p_k\texttt{)}, [i, j, \ldots]) &&= \mathit{scope}(p_i, [j, \ldots]) \text{ if } 1 \leq i \leq k;\\
&\mathit{scope}(\texttt{(}o\ \ p_1 \ldots p_k\ \ \texttt{etc)}, \mathit{idx}) &&= \mathit{scope}(\texttt{(}o\ \ p_1 \ldots p_k\texttt{)}, \mathit{idx});\\
&\mathit{scope}(p, \mathit{idx}) &&= \bot \text{ otherwise.}
\end{aligned}
$$

We write $\mathit{scope}(p)$ to denote the set $\{\mathit{idx} \mid \mathit{scope}(p, \mathit{idx}) \neq \bot\}$. We write $\mathit{varindex}(p, a) = i$ if $\mathit{scope}(p, i) = a\texttt{@}q$ otherwise $\bot$ to denote the index (relative to the pattern) of the subpattern associated with variable a, and $\mathit{reference}(p, a) = q$ if $\mathit{varindex}(p, a) \neq \bot$ otherwise $\bot$ with $\mathit{scope}(p, \mathit{varindex}(p, a)) = a\texttt{@}q$ to denote the subpattern associated with variable a.

Definition 5.6 (Bindings)**.** Let γ be a tree index and π be a tree permutation with range $\mathit{ran}(\pi)$. We say $\beta = \langle \gamma, \pi \rangle$ is a *binding* for a term t if π is a tree permutation for $\mathit{ref}(t, \gamma)$. We say that the tuple $\beta = \langle \gamma, \pi \rangle$ is a binding for a pattern p iff π covers the scope of p, that is, $\mathit{scope}(p) \subseteq \mathit{ran}(\pi)$. Intuitively, γ says which subterm of t should be matched to p, and π says how to permute that subterm to align it with p. That is, $\mathit{ref}(t, \gamma)^\pi$ aligns with p. The *identity binding* I_β is $\langle [], I \rangle$.[5] We write $\mathit{root}(\langle \gamma, \pi \rangle) \coloneqq \gamma$ and $\mathit{permutation}(\langle \gamma, \pi \rangle) \coloneqq \pi$ for the root index and permutation of a binding, respectively.

Example 5.7 (Bindings)**.** To illustrate bindings for algebra, consider again the example of firing the rule **A** on $t =$ `(= (+ x 0) 2)`. The set $\mathit{refs}(t)$ of all valid tree indices for t consists of the indices $[], [1], [1, 1], [1, 2], [2]$, which identify the subterms t, `(+ x 0)`, `x`, `0`, `2`, respectively (Definition 5.2). Since both `+` and `=` are commutative, valid tree permutations $\Pi(t)$ for t consist of the following mappings (Definition 5.3):

---

[5]The identity binding I_β is using the described notational shorthand for the indentity permutation, where $\langle [], I \rangle$ for a given term t is notation for $\langle [], I\restriction_{\mathit{refs}(t)} \rangle$.

$$
\begin{aligned}
&t^{\pi_0} = t && \pi_0 = \{[] \mapsto [], [1] \mapsto [1], [1,1] \mapsto [1,1], [1,2] \mapsto [1,2], [2] \mapsto [2]\} \\
&t^{\pi_1} = \texttt{(= (+ 0 x) 2)} && \pi_1 = \{[] \mapsto [], [1] \mapsto [1], [1,1] \mapsto [1,2], [1,2] \mapsto [1,1], [2] \mapsto [2]\} \\
&t^{\pi_2} = \texttt{(= 2 (+ x 0))} && \pi_2 = \{[] \mapsto [], [1] \mapsto [2], [1,1] \mapsto [2,1], [1,2] \mapsto [2,2], [2] \mapsto [1]\} \\
&t^{\pi_3} = \texttt{(= 2 (+ 0 x))} && \pi_3 = \{[] \mapsto [], [1] \mapsto [2], [1,1] \mapsto [2,2], [1,2] \mapsto [2,1], [2] \mapsto [1]\}
\end{aligned}
$$

Considering the subtrees of t, valid tree permutations for $\mathit{ref}(t,[1])$ consist of:

$$
\begin{aligned}
&\mathit{ref}(t,[1])^{\pi_4} = \texttt{(+ 0 x)} && \pi_4 = \{[] \mapsto [], [1] \mapsto [1], [2] \mapsto [2]\} \\
&\mathit{ref}(t,[1])^{\pi_5} = \texttt{(+ x 0)} && \pi_5 = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [1]\}
\end{aligned}
$$

The subtrees at the remaining indices ($[1,1]$, $[1,2]$ and $[2]$) consist of a single value, so the only valid tree permutation for those subtrees is the mapping $\pi_6 = \{[] \mapsto []\}$.

Next, we observe that the scope (Definition 5.5) of **A**'s pattern consists of the indices $\{[], [1], [2]\}$. The permutations for which their range is a superset of the scope of **A**'s pattern are $\{\pi_0, \pi_1, \pi_4, \pi_5\}$. Finally, we use these permutations of t and its subterms and the scope of **A** to compute all valid bindings for **A** and t (Definition 5.6):

$$
\{\langle [], \pi_0 \rangle, \langle [], \pi_1 \rangle, \langle [1], \pi_4 \rangle, \langle [1], \pi_5 \rangle\}
$$

Though all of these bindings are valid, only one ($\langle [1], \pi_5 \rangle$) is a binding under which **A** actually applies to t. We will return to this example to illustrate that application after defining the syntax and semantics of the DSL.

5.3.3 Semantics of Patterns, Conditions, and Actions.

Rule programs (Figure 5.6) denote partial functions from terms and bindings to terms. The semantics of these programs is an instantiation of the general RuleSy semantics (Figure 4.8). Here, we give domain-specific meaning to contexts, patterns, conditions, and actions. Figure 5.7 shows the types for the various semantic functions and the definition of those functions is shown in Figure 5.8. These semantics are well-defined only when the binding is valid for both the pattern and the input term (Definition 5.6).

program	::=	`with` *pattern*`:` `if` *condition*`:` `do` *action*
pattern	::=	`_` | *var* | *var*`@`*pattern* | *matcher* | `(`*op* {*pattern*}* [`etc`]`)`
condition	::=	*pred* | *pred* `and` *pred*
pred	::=	`true` | *ref* `=` *const* | *ref* `!=` *const*
action	::=	*cmd* {`and` *cmd*}*
cmd	::=	`remove` *ref* | `replace` *ref* `with` *expr*
expr	::=	*const* | *obj*
obj	::=	`(`*op* {*expr*}*`)` | *expr* `::` *ref* | *expr* `::` *obj*
const	::=	*value* | *ref* | `apply(`*op* {*const*}*`)`
ref	::=	*var*
var	::=	identifier
matcher	::=	`num` | `var` | `base`

Figure 5.6: Syntax for the algebra DSL. The notation {*form*}* means zero or more repetitions of the given form, and the notation [*form*] means zero or one repetitions. Both *op* and *value* are the same as for term syntax (Figure 5.5).

The function $\mathcal{P}$ gives meaning to patterns as partial functions from terms and bindings to contexts. Patterns for tree-rewrite rules represent trees to be matched against some subterm of term. Potential patterns include matching against a basic value (using e.g., `num`, `var`), a complex term with a particular operator *op* and its arguments (using `(`*op* ...`)`), or any term (using `_`). Patterns may also include variables (using x`@`p), which are used to reference parts of the matched term in conditions and actions. For tree-rewrite rules, a context maps a variable x of the pattern to values representing subtrees of the input term. These values have two parts and are defined as a tuple $\langle i, t \rangle$, representing an *address* and *value*, respectively. The address i, for applying actions, is the tree index of x in the original input term. The value t is the subterm of the input term pointed to by x, used in expressions within the condition and action.

The semantics of $\mathcal{P}$ are defined by the semantic functions $\mathcal{M}$ and $\mathcal{B}$. The function $\mathcal{M}$ (for "match") gives meaning to the pattern, checking that the input term fits the pattern

under the binding. The function $\mathcal{B}$ (for "bind") gives meaning to the variables in the pattern, extracting the tree indices to which they point. The function *matches* gives meaning to the domain-specific *matcher* patterns.

The function $\mathcal{C}$ gives meaning to conditions as predicates on contexts. The function $\mathcal{A}$ gives meaning to actions as functions from contexts and terms to terms. Actions apply a set of parallel functional edits to disjoint subterms of the input term t. The function $\mathcal{E}$ (for "evaluate") gives meaning to subexpressions of the condition and action that map contexts to terms. As defined by $\mathcal{E}$, actions can create new terms, and both conditions and actions can evaluate expression terms with literal arguments (via **`apply`**). *operator* gives meaning to **`apply`** by giving meaning to the domain-specific operators that appear in terms.

$$
\begin{aligned}
\text{binding} &= \text{tree index} \times \text{tree permutation} \\
\text{context} &= \mathit{var} \to \text{tree index} \times \text{term} \\
\mathcal{P} &: \mathit{pattern} \times \text{term} \times \text{binding} \to \text{context} \cup \{\bot\} \\
\mathcal{C} &: \mathit{condition} \times \text{context} \to \text{boolean} \\
\mathcal{A} &: \mathit{action} \times \text{term} \times \text{context} \to \text{term} \\
\mathcal{M} &: \mathit{pattern} \times \text{term} \to \text{boolean} \\
\mathcal{B} &: \mathit{pattern} \times \text{tree index} \to 2^{\mathit{var} \times \text{tree index}} \\
\mathcal{E} &: \mathit{expr} \times \text{context} \to \text{term} \\
\mathit{matches} &: \mathit{matcher} \times \mathit{value} \to \text{boolean} \\
\mathit{operator} &: \mathit{operator} \times \text{term sequence} \to \text{term}
\end{aligned}
$$

Figure 5.7: Types for the algebra DSL's inputs/outputs and semantic functions. A binding $\langle \gamma, \pi \rangle$ is a tuple of a tree index and a tree permutation (see Definition 5.6), and contexts are mappings from variables in the pattern to a tuple representing the *address* of that variable (for applying actions) and the *value* of the subterm of that variable (for evaluating expressions).

Well-formed Rule Programs

The meaning of rules is defined only for *well-formed programs* (Definition 5.8), which contain no invalid references. If a term matches the pattern of a well-formed program, then every reference in that program specifies a unique variable declared in the pattern. Additionally, `apply` and `::` expressions reference subterms of the right kind and have appropriate arity;

$$
\begin{aligned}
\llbracket \texttt{with}\ p\texttt{:}\ \texttt{if}\ c\texttt{:}\ \texttt{do}\ a \rrbracket(t,\langle\gamma,\pi\rangle) &= \textbf{if}\ \sigma \neq \bot \wedge \mathcal{C}\llbracket c\rrbracket\sigma\ \textbf{then}\ \mathcal{A}\llbracket a\rrbracket(t,\sigma)\ \textbf{else}\ \bot \\
&\quad\ \textbf{where}\ \ \sigma = \mathcal{P}\llbracket p\rrbracket(t,\langle\gamma,\pi\rangle) \\
\mathcal{P}\llbracket p\rrbracket(t,\langle\gamma,\pi\rangle) &= \textbf{if}\ \mathcal{M}\llbracket p\rrbracket t'\ \textbf{then} \\
&\quad \{x \mapsto \langle concat(\gamma,\pi^{-1}(i)), ref(t',i)\rangle \mid \langle x,i\rangle \in \mathcal{B}\llbracket p\rrbracket[]\} \\
&\quad \textbf{else}\ \bot\ \ \textbf{where}\ \ t' = ref(t,\gamma)^{\pi} \\
\mathcal{M}\llbracket \texttt{(}o\ \ p_1 \dots p_k\texttt{)}\rrbracket t &= t \doteq term(o,t_1,\dots,t_k) \wedge \forall_{1\le i\le k}\mathcal{M}\llbracket p_i\rrbracket t_i \\
\mathcal{M}\llbracket \texttt{(}o\ \ p_1 \dots p_k\ \ \texttt{etc)}\rrbracket t &= t \doteq term(o,t_1,\dots,t_n) \wedge k \ge n \wedge \forall_{1\le i\le k}\mathcal{M}\llbracket p_i\rrbracket t_i \\
\mathcal{M}\llbracket \texttt{_}\rrbracket t &= \text{true} \\
\mathcal{M}\llbracket var\rrbracket t &= \text{true} \\
\mathcal{M}\llbracket var\texttt{@}p\rrbracket t &= \mathcal{M}\llbracket p\rrbracket t \\
\mathcal{M}\llbracket constructor\rrbracket t &= t\ \text{is a}\ value \wedge matches\llbracket constructor\rrbracket t \\
\mathcal{B}\llbracket \texttt{(}o\ \ p_1 \dots p_k\texttt{)}\rrbracket idx &= \textstyle\bigcup_{1\le i\le k}\mathcal{B}\llbracket p_i\rrbracket concat(idx,[i]) \\
\mathcal{B}\llbracket \texttt{(}o\ \ p_1 \dots p_k\ \ \texttt{etc)}\rrbracket idx &= \mathcal{B}\llbracket \texttt{(}o\ \ p_1 \dots p_k\texttt{)}\rrbracket idx \\
\mathcal{B}\llbracket \texttt{_}\rrbracket idx &= \emptyset \\
\mathcal{B}\llbracket var\rrbracket idx &= \mathcal{B}\llbracket var\texttt{@_}\rrbracket idx \\
\mathcal{B}\llbracket var\texttt{@}p\rrbracket idx &= \mathcal{B}\llbracket p\rrbracket idx \cup \{\langle var, idx\rangle\} \\
\mathcal{B}\llbracket constructor\rrbracket idx &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\llbracket \texttt{true}\rrbracket\sigma &= \text{true} \\
\mathcal{C}\llbracket r\ \texttt{=}\ e\rrbracket\sigma &= \mathcal{E}\llbracket r\rrbracket\sigma \doteq \mathcal{E}\llbracket e\rrbracket\sigma \\
\mathcal{C}\llbracket r\ \texttt{!=}\ e\rrbracket\sigma &= \mathcal{E}\llbracket r\rrbracket\sigma \not\doteq \mathcal{E}\llbracket e\rrbracket\sigma \\
\mathcal{C}\llbracket b_1\ \texttt{and}\ b_2\rrbracket\sigma &= \mathcal{C}\llbracket b_1\rrbracket\sigma \wedge \mathcal{C}\llbracket b_2\rrbracket\sigma
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}\llbracket a_1\ \texttt{and}\ a_2\rrbracket(t,\sigma) &= (\mathcal{A}\llbracket a_1\rrbracket \parallel \mathcal{A}\llbracket a_2\rrbracket)(t,\sigma) \\
\mathcal{A}\llbracket \texttt{remove}\ r\rrbracket(t,\sigma) &= remove(t,i)\ \ \textbf{where}\ \ \langle i,v\rangle = \sigma[r] \\
\mathcal{A}\llbracket \texttt{replace}\ r\ \texttt{with}\ e\rrbracket(t,\sigma) &= replace(t,i,\mathcal{E}\llbracket e\rrbracket\sigma)\ \ \textbf{where}\ \ \langle i,v\rangle = \sigma[r]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}\llbracket \texttt{(}o\ \ e_1 \dots e_k\texttt{)}\rrbracket\sigma &= term(o,\mathcal{E}\llbracket e_1\rrbracket\sigma,\dots,\mathcal{E}\llbracket e_k\rrbracket\sigma) \\
\mathcal{E}\llbracket e_1\ \texttt{::}\ e_2\rrbracket\sigma &= constree(\mathcal{E}\llbracket e_1\rrbracket\sigma,\mathcal{E}\llbracket e_2\rrbracket\sigma) \\
\mathcal{E}\llbracket \texttt{apply(}o\ \ e_1 \dots e_k\texttt{)}\rrbracket\sigma &= operator\llbracket o\rrbracket(\mathcal{E}\llbracket e_1\rrbracket\sigma,\dots,\mathcal{E}\llbracket e_k\rrbracket\sigma) \\
\mathcal{E}\llbracket value\rrbracket\sigma &= value \\
\mathcal{E}\llbracket var\rrbracket\sigma &= v\ \ \textbf{where}\ \ \langle i,v\rangle = \sigma[var]
\end{aligned}
$$

$$
\begin{aligned}
cons(h,[i_1,\dots,i_n]) &= [h,i_1,\dots,i_n] \\
replace(t,[],s) &= s \\
replace(\texttt{(}o\ \ t_1 \dots t_k\texttt{)}, cons(i,lst), s) &= term(o,t_1,\dots,t_{i-1},replace(t_i,lst,s),t_{i+1},\dots,t_k) \\
remove(\texttt{(}o\ \ t_1 \dots t_k\texttt{)}, [i], s) &= term(o,t_1,\dots,t_{i-1},t_{i+1},\dots,t_k) \\
remove(\texttt{(}o\ \ t_1 \dots t_k\texttt{)}, cons(i,lst), s) &= term(o,t_1,\dots,t_{i-1},remove(t_i,lst),t_{i+1},\dots,t_k) \\
constree(t,\ \texttt{(}o\ \ t_1 \dots t_k\texttt{)}) &= term(o,t,t_1,\dots,t_k) \\
concat([i_1,\dots,i_n],[j_1,\dots,j_m]) &= [i_1,\dots,i_n,j_1,\dots,j_m]
\end{aligned}
$$

Figure 5.8: Semantics for the tree-rewrite DSL. The notation $\parallel$ stands for parallel function composition (actions are independent); other notation is described in Section 4.2 and Definitions 5.1–5.6. Semantics for the functions *operator* and *matches* are shown in Figure 5.9.

$$
\begin{aligned}
&\mathit{operator}[\![\texttt{+}]\!](x_1, \ldots, x_k) &=& \textstyle\sum_{i \in \{1,\ldots,k\}} x_i \\
&\mathit{operator}[\![\texttt{*}]\!](x_1, \ldots, x_k) &=& \textstyle\prod_{i \in \{1,\ldots,k\}} x_i \\
&\mathit{operator}[\![\texttt{-}]\!](x) &=& -x \\
&\mathit{operator}[\![\texttt{-}]\!](x, y) &=& x - y \\
&\mathit{operator}[\![\texttt{/}]\!](x, y) &=& x/y \\
&\mathit{matches}[\![\texttt{num}]\!]t &=& t \text{ is a numeric literal } \mathit{value}. \\
&\mathit{matches}[\![\texttt{var}]\!]t &=& t \text{ is a variable identifier } \mathit{value}. \\
&\mathit{matches}[\![\texttt{base}]\!]t &=& \mathit{matches}[\![\texttt{num}]\!]t \vee \mathit{matches}[\![\texttt{var}]\!]t
\end{aligned}
$$

Figure 5.9: Algebra-specific semantics for the DSL. The meaning of these can change to support other domains like semantic proofs.

the program's actions edit disjoint subtrees of the term's AST; and (in)equality predicates only compare subterms matched by terminal patterns. Well-formed programs are therefore free of runtime errors caused by invalid references to subterms.

Definition 5.8 (Well-Formed Programs)**.** Let $R =$ **`with`** p`:` **`if`** c`:` **`do`** a_1 **`and`** ... **`and`** a_n be a program in the DSL (Figure 5.6). We say that R is *well-formed* if the following hold:

- The pattern p contains no duplicate variable names.
- For all references r contained in c and a_1 through a_n, $\mathit{varindex}(p, r) \neq \bot$.
- $\mathit{reference}(p, r) =$ `(`op ...`)` for all expressions e `::` r.
- $\mathit{reference}(p, r) \neq$ `(`op ...`)` for all references r in all (in)equality predicates.
- Let r_k denote the first argument to a command a_k in R. For all distinct a_i, a_j in R, $\mathit{varindex}(p, r_i)$ is not a prefix of $\mathit{varindex}(p, r_j)$ and vice versa (this means that actions must apply to disjoint subtrees and are therefore independent of the order in which they are applied).
- For all **`apply`** expressions, the arguments are of the right kind and arity; this is domain-specific. For algebra, this means that $\mathit{reference}(p, r) =$ `num` for all references r in **`apply`** expressions and all **`apply`** expressions have operator arguments and arities compatible with the semantics in Figure 5.9.

Example 5.9 (A Complete Example Rule Application)**.** To illustrate the semantics of rules, we return to our example of firing the rule **A** on the term $t =$ `(= (+ x 0) 2)`. We'll consider the binding $\beta = \langle \gamma, \pi \rangle$ where $\gamma = [1]$ and $\pi = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [1]\}$, and define p, c, and a to be the pattern, condition, and action of **A**, respectively. Thus, $\mathit{ref}(t, \gamma) =$ `(+ x 0)` and $\mathit{ref}(t, \gamma)^{\pi} =$ `(+ 0 x)`. Looking first at the pattern semantics, we have:

$$
\begin{aligned}
\mathcal{M}[\![p]\!]\mathit{ref}(t,\gamma)^{\pi} &= \mathcal{M}[\![\texttt{(+ } a\texttt{@num _ etc)}]\!]\texttt{(+ 0 x)} \\
&= \texttt{(+ 0 x)} \doteq \texttt{(+ } t_1 \ldots t_n\texttt{)} \wedge 2 \geq n \wedge \mathcal{M}[\![a\texttt{@num}]\!]\texttt{0} \wedge \mathcal{M}[\![\texttt{_}]\!]\texttt{x} \\
&= \text{true} \wedge \text{true} \wedge \mathcal{M}[\![\texttt{num}]\!]\texttt{0} \wedge \text{true} \\
&= \text{true} \wedge \text{true} \wedge \text{true} \wedge \text{true} \\
&= \text{true}
\end{aligned}
$$

Thus, t matches p under β. Next, $\mathcal{B}$ is defined only with respect to the pattern rather than the input term. We have:

$$
\begin{aligned}
\mathcal{B}[\![p]\!][] &= \mathcal{B}[\![\texttt{(+ } a\texttt{@num _ etc)}]\!][] \\
&= \mathcal{B}[\![a\texttt{@num}]\!]\mathit{concat}([],[1]) \cup \mathcal{B}[\![\texttt{_}]\!]\mathit{concat}([],[2]) \\
&= \mathcal{B}[\![a\texttt{@num}]\!][1] \cup \mathcal{B}[\![\texttt{_}]\!][2] \\
&= (\mathcal{B}[\![\texttt{num}]\!][1] \cup \langle a, [1]\rangle) \cup \emptyset \\
&= (\emptyset \cup \langle a, [1]\rangle) \cup \emptyset \\
&= \{\langle a, [1]\rangle\}
\end{aligned}
$$

That is, a is the only variable in the pattern and it points to the subterm at index [1] (relative to the pattern). So by the definition of $\mathcal{P}$, we have that the context for this application is

$$
\begin{aligned}
\mathcal{P}[\![p]\!](t, \langle \gamma, \pi \rangle) &= \textbf{if } \mathcal{M}[\![p]\!] \mathit{ref}(t,\gamma)^{\pi} \textbf{ then} \\
&\quad \{x \mapsto \langle \mathit{concat}(\gamma, \pi^{-1}(i)), \mathit{ref}(\mathit{ref}(t,\gamma)^{\pi}, i)\rangle \mid \langle x, i\rangle \in \mathcal{B}[\![p]\!][\,]\} \\
&\quad \textbf{else } \bot \\
&= \textbf{if } \text{true } \textbf{then} \\
&\quad \{x \mapsto \langle \mathit{concat}(\gamma, \pi^{-1}(i)), \mathit{ref}(\mathit{ref}(t,\gamma)^{\pi}, i)\rangle \mid \langle x, i\rangle \in \{\langle a, [1]\rangle\}\} \\
&\quad \textbf{else } \bot \\
&= \{\ a \mapsto \langle \mathit{concat}(\gamma, \pi^{-1}([1])), \mathit{ref}(\mathit{ref}(t,\gamma)^{\pi}, [1])\rangle\} \\
&= \{\ a \mapsto \langle \mathit{concat}([1],[2]), \mathit{ref}(\texttt{(+ 0 x)}, [1])\rangle\} \\
&= \{\ a \mapsto \langle [1,2], \texttt{0}\rangle\}
\end{aligned}
$$

That is, there is one variable a, the index in t to which a was bound is $[1, 2]$, and the value of that subterm is 0. Let $\sigma = \{\ a \mapsto \langle [1,2], \texttt{0}\rangle\}$ be this context.

Looking now at the condition, we have:

$$
\begin{aligned}
\mathcal{C}[\![c]\!]\sigma &= \mathcal{C}[\![a \texttt{ = 0}]\!]\sigma \\
&= \mathcal{C}[\![a]\!]\sigma = \mathcal{E}[\![\texttt{0}]\!]\sigma \\
&= \mathit{snd}(\sigma[a]) = \texttt{0} \\
&= \texttt{0} = \texttt{0} \\
&= \text{true}
\end{aligned}
$$

Thus, the condition is true. With the pattern matching and the condition holding, we can apply the action:

$$
\begin{aligned}
\mathcal{A}[\![a]\!](t,\sigma) &= \mathcal{A}[\![\texttt{remove } a]\!](t,\sigma) \\
&= \mathit{remove}(t, i) \ \ \textbf{where} \ \ \langle i, v\rangle = \sigma[a] \\
&= \mathit{remove}(\texttt{(= (+ x 0) 2)}, [1,2]) \\
&= \texttt{(= (+ x) 2)}
\end{aligned}
$$

Putting it all together, the semantics of **A** applied to the term `(= (+ x 0) 2)` and the binding $\langle [1], \{[\,] \mapsto [\,], [1] \mapsto [2], [2] \mapsto [1]\}\rangle$ is the term `(= (+ x) 2)`.[6]

[6]Note the unary + in the result term. As RuleSy does not implicitly understand the semantics of terms, it does not know that this can be simplified to `(= x 2)`. The framework relies on such a rewrite rule being provided as an axiom. The axioms used in our implementation are listed in Section 5.11.

5.4 *Characterizing Sound and Useful Rules*

Before we can use our DSL to synthesize sound and useful tactic rules, we must be precise about exactly what we mean by sound and useful. In this section we formalize both of these notions. They center around using a set of *axiom* rules that (along with the theory of integers and term isomorphism) characterize a computable equivalence relation used both to verify tactics and ensure they are useful.

Within the domains of sovling algebra equations, doing semantic logic proofs, or solving term-equivalence problems in general, educators teach students to use a set of rules rather than a single general algorithm. Thus, rather than use an oracle for all equivalences, RuleSy will work with a sound subset of equivalences, defined by the axioms RuleSy takes as input (along with a bit more, explained in this section). Of course, the reason RuleSy has any works it needs to do at all is that, while these axioms make it *possible* to solve a wide range of problems, they do not do so *efficiently*. The tactics RuleSy aims to learn are no more expressive that the set of axioms, but they can be more efficient (for humans) on a wide range of problems we care about teaching.[7] Characterizing a subset of term equivalences using axioms is, incidentally, useful because we might not even be able to write down a fully general algorithm (if, say, we have non-linear integer arithmetic). However, the more salient motivation in this context for dealing with axioms and tactics is because that is what humans use to solve algebra problems.

5.4.1 Soundess of Rules

RuleSy relies on the domain definition to guarantee the soundness of synthesized rules. As discussed in Section 4.4.1, our ideal domain description would take the form of an oracle specifying what it means for any rule to be correct: namely, a relation $\mathcal{E}$ describing all semantically equivalent terms. Given any program R (a partial function from terms and

[7] And, as we'll see in the technical section, we only bother attempting to learn rule programs for specifications which are guaranteed to shorten a solution path on at least one problem in the given set of training examples.

bindings to terms), we can say that R is sound if and only if all pairs of terms mapped by R are equivalent under $\mathcal{E}$. That is, the set $\{\langle s, t\rangle \mid t \in \mathit{fire}(R, s)\}$ is a subset of $\mathcal{E}$. The question before us is how to compute this equivalence relation in order to verify rules on top of a set of axiomatic rules.

First, what does semantic equivalence mean, in general? Algebraic terms have meaning given by assigning values to variables. In our setting, we can say that two terms s and t are equivalent if and only if, for any assignment of the variables in s and t to integer values, the terms s and t with their variables substituted according to this assignment evaluate to the same value under the theory of integer arithmetic. Incidentally, because our terms are non-linear (they have division), the theory of integers is undecidable, meaning that we cannot compute this relation directly.

We can, however, compute a useful subset of this relation on which we can base soundness, which will in turn form the basis for verification. For example, the trivial identity relation is such a subset: two terms are obviously semantically equivalent if they are syntactically identical. The term isomorphism described in Definition 5.4 is a slightly better one: the meaning of commutative operators (by definition) does not change when the arguments are reordered, so any two terms that are permutations of each other will evaluate to the same value under any variable assignment. Here, "better" means that the equivalence relation defined by term isomorphism is a strict superset of the identity relation. We want the largest possible equivalence relation so that we can verify as many potential rules as possible.

As another example, because the semantics of algebra are defined recursively on the structure of the term, we can replace subterms with semantically equivalent subterms and preserve overall equivalence. That is, given a term t and an index γ, if $\mathit{ref}(t, \gamma)$ is semantically equivalent to another term s, then t is semantically equivalent to $\mathit{replace}(t, \gamma, s)$. This, on its own, does not specify an equivalence relation, but it does take another equivalence relation and make it larger by merging equivalence classes.

In order to reach a sufficiently large equivalence relation that can characterize the soundness of interestingly complex rules, RuleSy relies on *axiom* rules, taking a set of axioms $\mathcal{A}$

as input.[8] In our implementation, $\mathcal{A}$ is a set of programs in the DSL, though in principle these axioms could be any kind of (partial) function of terms to terms. As made clear by their name, RuleSy assumes these axioms to be sound programs with respect to a hypothetical equivalence-deciding oracle. Each individual axiom describes pairs of terms that belong to the same equivalence class. Two terms are equivalent if there is a sequence of axiom applications from one to the other.

Taken together, the transitive closure of term isomorphisms, equivalent subterm replacement, and axiom application define a computable equivalence relation of semantically equivalent terms. We formally characterize this with Definition 5.10. The *fire* function captures the closure of a given axiom over isomorphism and equivalent subterm replacement. Thus, intuitively, a program is sound if, for any application of that program, there exists a sequence of axioms that can be chained to match that behavior using *fire*.

Definition 5.10 (Soundess of Rules)**.** Let R be a program in the algebra DSL. We say that R is sound with respect to a set of axioms $\mathcal{A}$ if, for every t, β, t_n where $[\![R]\!](t, \beta) = t_n \neq \bot$, there exist terms $t_0, \ldots, t_{n-1}$ with $t \simeq t_0$, and for all $i \in \{1, \ldots, n\}$, there exists term s and $A \in \mathcal{A}$ such that $t_i \simeq s$ and $s \in \mathit{fire}(A, t_{i-1})$.

5.4.2 Useful Rules

As discussed in Section 4.4.1, the useful of tactics has multiple aspects. First, rules should apply to problems the framework user cares about (i.e., the input training problems), and those rules should enable more efficient solving of those problems than the axioms alone. As we discuss in Section 5.5.3, RuleSy measures this efficiency by looking at both the number of steps required and the relative size of rules used. Our specification mining algorithm only considers specifications that could potentially lead to rules that lead to more efficient solving.

The other major aspect of usefulness is how general rules are. We want rules to apply to as wide a variety of states as possible. In the context of tree-rewrite rules, characteriz-

[8] The axioms used in our evaluation are listed in Section 5.11.

ing generality is fairly straightforward: because specifications are defined by a sequence of axioms, specifications describe general behavior. Thus, our goal is to synthesize rules that cover as many of the state transitions defined by the specification as possible. Section 5.5.2 describes how we achieve this.

5.5 Rule Mining, Synthesis, and Optimization

Given an educational objective, example problems, and axioms for solving those problems, RuleSy produces an optimal domain model in three stages: (1) specification mining, (2) rule synthesis, and (3) domain model optimization. The framework has the same overall structure as presented in Figure 4.10. This section presents the algorithms underlying each stage and states their guarantees. Proofs for theorems can be found in Section 5.10.

5.5.1 Specification Mining

Specification mining takes as input a set of examples and a domain definition as a set of axioms, and produces a set of specifications for tactic rules. The core technical contribution behind this stage is our definition of tactic specifications and the algorithm for mining them. We describe the key challenge in specifying tactics; show how our notion of *execution plans* addresses it; and present our FindSpecs algorithm for computing these plans.

Specifying Sound Rule Behavior

Our output specifications must describe sound and useful rule behavior, and RuleSy must produce them using the domain definition and example problem inputs.

Given a set of axioms and example problems, our first challenge is how to represent specifications for sound tactics. To enable efficient synthesis of useful rules, a tactic specification should capture the semantics of a rule—i.e., a partial function—that can help solve some problems in fewer steps than the axioms alone. But natural forms of specification, such as axiom sequences, do not satisfy this requirement, as we illustrate in Example 5.11.

Example 5.11. To see why, consider the axiom sequence $\mathbf{I} \circ \mathbf{B}$, where **I** implements factoring (Figure 5.10) and **B** implements constant folding (Figure 5.1a).[9] Intuitively, we would like $\mathbf{I} \circ \mathbf{B}$ to specify the tactic **IB** for combining like terms (Figure 5.10). Obvious interpretations of this sequence do not capture the meaning of the tactic. If we interpret $\mathbf{I} \circ \mathbf{B}$ using the *fire* semantics, the result is a non-functional relation that includes the meaning of multiple tactics. For example, firing $\mathbf{I} \circ \mathbf{B}$ on the term `(+ (* 2 x) (* 3 x) (* 4 y) (* 5 y))` produces both `(+ (* 5 x) (* 4 y) (* 5 y))` and `(+ (* 9 y) (* 2 x) (* 3 x))`. So specifications need to constrain the application of the axioms. If we instead interpret $\mathbf{I} \circ \mathbf{B}$ as the composition of the partial functions denoted by its axioms naïvely fixed with identity bindings—i.e., as $\lambda t.[\![\mathbf{B}]\!]([\![\mathbf{I}]\!](t, I_\beta), I_\beta)$—the resulting relation is empty and thus fails to specify a useful tactic.

```
#  (+ (* e0 e) (* e1 e) ...) → (+ (* (+ e0 e1) e) ...)
def I:
    with (+ (* a x) r@(* b y) etc):
    if x = y:
    do remove r and replace a with (+ a b)

#  (+ (* c0 e) (* c1 e) ...) → (+ (* c e) ...), c = c0 + c1
def IB:
    with (+ (* a@num x) r@(* b@num y) etc):
    if x = y:
    do remove r and replace a with apply(+ a b)
```

Figure 5.10: The tactic **IB** for combining like terms combines factoring (**I**) and constant folding (**B** in Figure 5.1a), but no interpretation of $\mathbf{I} \circ \mathbf{B}$ captures its behavior.

We address the challenge of specifying tactic rules with *execution plans*. An execution plan (Definition 5.14) is a partial function from terms to terms, encoded as a sequence of execution steps (Definition 5.13). An execution step combines a program R with a binding β for R's pattern. An execution plan composes a sequence of such rule applications. For

[9] The names **I** and **B** come from the axiom table shown in a later section in Table 5.1b.

example, the plan $[\langle \mathbf{I}, \langle [], I \rangle \rangle, \langle \mathbf{B}, \langle [1,1], I \rangle \rangle]$ captures the behavior of the combine-like-terms rule on terms of the form `(+ (*` c_0 e`) (*` c_1 e`) ...)`. Moreover, firing a program that implements this plan (e.g., **IB**) captures the common understanding of what it means to combine like terms when solving algebra problems. Execution plans thus satisfy our requirement for tactic specifications by defining useful functional relations, and as a sequence of axiom applications, execution plans capture sound rule behavior as defined by Definition 5.10. RuleSy will attempt to synthesize rules for each individual execution plan.

Definition 5.12 (Tree Permutation Extensions)**.** We say a permutation ϕ *extends* a permutation π iff ϕ agrees with π on all places π is defined; that is, $\pi \subseteq \phi$. In particular, if ϕ is a permutation for term t that extends π, we say that ϕ is an extension of π to t.

Definition 5.13 (Execution Step)**.** An *execution step* $\langle R, \beta \rangle$ is a tuple consisting of a rule program R and a valid binding β for R's pattern.

Definition 5.14 (Execution Plan)**.** An *execution plan* S is a finite sequence of execution steps $[\langle R_1, \langle \gamma_1, \pi_1 \rangle \rangle, \ldots, \langle R_n, \langle \gamma_n, \pi_n \rangle \rangle]$. The plan S composes its steps as follows: $[\![S]\!]t_0 = t_n$ if there exist permutations $\phi_1, \ldots, \phi_n$ such that $[\![R_i]\!](t_{i-1}, \langle \gamma_i, \phi_i \rangle) = t_i$, ϕ_i extends π_i, and $t_i \neq \bot$ for all $i \in \{1, \ldots, n\}$; otherwise, $[\![S]\!]t_0 = \bot$. The plan S is *general* if the step indices $\mathit{root}(\beta_1), \ldots, \mathit{root}(\beta_n)$ have the empty index [] as their longest common prefix.

Given a term t and binding $\beta = \langle \gamma, \pi \rangle$, we define the function *apply* as:

$$\mathit{apply}(S, t, \langle \gamma, \pi \rangle) = \mathit{replace}(t, \gamma, [\![S]\!]\mathit{ref}(t, \gamma)^{\pi})$$

That is, $\mathit{apply}(S, t, \beta)$ applies the plan to the subterm of t specified by β.

Note that, as defined, a plan doesn't fix the bindings exactly, but rather fixes them partially. Specifically, permutations can be extended (Definition 5.12) when applying a plan. Example 5.15 illustrates why this is necessary: we want plans to be able to apply to terms of various shapes, where extra subterms that are not impacted by the rule applications are allowed. Defining the semantics of plans in this way, with respect to extended permutations, is necessary to allow for tactics that apply to various shapes (e.g., have **`etc`** or `_` in the

pattern).

Example 5.15 (Plan Semantics)**.** We return to the example from Example 5.11 to illustrate plan semantics, explicitly writing out the permutations (rather than using the notational shorthand) to make it clear why extensions are necessary. Let $S = [\langle \mathbf{I}, \langle [], \pi_1 \rangle \rangle, \langle \mathbf{B}, \langle [1,1], \pi_2 \rangle \rangle]$ be the plan capturing the intuitive notion of combining like terms, where $\pi_1 = \{[] \mapsto [], [1] \mapsto [1], [1,1] \mapsto [1,1], [1,2] \mapsto [1,2], [2] \mapsto [2], [2,1] \mapsto [2,1], [2,2] \mapsto [2,2]\}$ and $\pi_2 = \{[] \mapsto [], [1] \mapsto [1], [2] \mapsto [2]\}$. Given the term $t_1 =$ `(+ (* 3 x) (* 5 x))`, we have:

$$[\![B]\!]([\![I]\!](t_1, \langle [], \pi_1 \rangle), \langle [1,1], \pi_2 \rangle) = [\![B]\!](\texttt{(+ (* (+ 3 5) x))}, \langle [1,1], \pi_2 \rangle) = \texttt{(+ (* 8 x))}$$

Any permutation trivially extends itself, so by Definition 5.14 we have $[\![S]\!]t_1 =$ `(+ (* 8 x))`.

But now consider $t_2 =$ `(+ (* 3 x) (* 5 x) 4)`. Intuitively, the plan S should still apply to t_2 because the extra `4` on the end doesn't impact the ability of either rule in the plan to apply. However, $\langle [], \pi_1 \rangle$ is *not* a valid binding for t_2 because the domain doesn't cover all of $\mathit{refs}(t_2)$. This is why the semantics of plans allows for extensions of permutations; extending π_1 to $\pi_1' = \pi_1 \cup \{[3] \mapsto [3]\}$, we find that $\langle [], \pi_1' \rangle$ *is* a valid binding for t_2. Thus:

$$\begin{aligned}[\![B]\!]([\![I]\!](t_2, \langle [], \pi_1 \rangle), \langle [1,1], \pi_2 \rangle) &= [\![B]\!](\texttt{(+ (* (+ 3 5) x 4))}, \langle [1,1], \pi_2 \rangle) \\ &= \texttt{(+ (* 8 x) 4)}\end{aligned}$$

Therefore, $[\![S]\!]t_2 =$ `(+ (* 8 x) 4)`.

Computing Plans

RuleSy mines execution plans from a set of example problems and axioms using the FindSpecs procedure shown in Figure 5.11. These example problems serve to bias RuleSy into finding useful rules; the synthesized tactics will apply to problem states similar to those given as example problems. FindSpecs first obtains a *solution graph* (Definition 5.16) of all shortest solutions to each example problem (line 3). It then applies the FindPlan procedure

to compute an execution plan for every path between every pair of nodes in each resulting graph (line 6). These plans specify a set of partial functions that can soundly shorten the solution to at least one example problem (Theorem 5.20).

Definition 5.16 (Solution Graph)**.** A directed multigraph $G = \langle N, E\rangle$ is a *solution graph* for a term t, predicate SOLVED, and rules $\mathcal{R}$ if N is a set of terms, $t \in N$; E is a set of labeled edges $\langle \mathit{src}, \mathit{tgt}\rangle_R$ such that $\mathit{src}, \mathit{tgt} \in N$, $R \in \mathcal{R}$, and $\mathit{tgt} \in \mathit{fire}(R, \mathit{src})$; G is acyclic; t is the only term in G with no incoming edges; G contains at least one sink term with no outgoing edges; and each sink satisfies the SOLVED predicate.

FINDPLAN takes as input a path p in a solution graph and produces a general execution plan (Definition 5.14) for *replaying* that path (Definition 5.17). The first loop, at lines 10–13, creates a plan replaying the path p from n_0 to n_k exactly: i.e., $[\![S]\!]n_0 = n_k$. The second loop, at lines 14–17, generalizes S to be more widely applicable, while still replaying the path p.

Definition 5.17 (Replaying Paths)**.** Let $p = n_0 \rightarrow_{R_1} \ldots \rightarrow_{R_k} n_k$ be a path in a solution graph, consisting of a sequence of k edges labeled with rules $R_1, \ldots, R_k$. An execution plan S *replays* the path p if S is a sequence of k steps $[\langle R_1, \beta_1\rangle, \ldots, \langle R_k, \beta_k\rangle]$, one for each edge in p, and there exists binding β such that $\mathit{apply}(S, n_0, \beta) \simeq n_k$.

Example 5.18. To illustrate, consider applying FINDPLAN to the path `(= (+ x 1 -1) 5)` $\rightarrow_{\mathbf{B}}$ `(= (+ 0 x) 5)` $\rightarrow_{\mathbf{A}}$ `(= x 5)` in Figure 5.2a. The SOLVE procedure computes this path p by firing **B** with $\mathit{idx} = [1]$, $\beta_{\mathbf{B}} = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3]\}$, and **A** with $\mathit{idx} = [1]$, $\beta_{\mathbf{A}} = \{[] \mapsto [], [1] \mapsto [1]]\}$. As a result, the loop at lines 10–13 executes twice to produce the plan $S = [\langle \mathbf{B}, \mathit{idx}, \beta_{\mathbf{B}}\rangle, \langle \mathbf{A}, \mathit{idx}, \beta_{\mathbf{A}}\rangle]$. The plan S replays p exactly: it describes a tactic for applying the axioms $\mathbf{B} \circ \mathbf{A}$ to a term whose first child has two opposite constants as its second and third children. The loop at lines 14–17 generalizes S to produce the plan in Figure 5.2b. This plan replays p but applies to *any* term with opposite constants as its second and third children.

Definition 5.19 (Shortcuts)**.** A path p is a *potential shortcut path* in a solution graph G if p contains more than one edge and p is a subpath of a shortest path from G's source to one of

1: **function** FINDSPECS(T: set of terms, $\mathcal{A}$: set of well-formed programs)
2: $\quad \mathcal{S} \leftarrow \{\}$
3: $\quad$ **for all** $\langle N, E\rangle \in \{\textsc{Solve}(t, \mathcal{A}) \mid t \in T\}$ **do**
4: $\quad\quad$ **for all** $\mathit{src}, \mathit{tgt} \in N$ **do**
5: $\quad\quad\quad \mathit{paths} \leftarrow \mathit{allPaths}(\mathit{src}, \mathit{tgt}, \langle N, E\rangle)$ $\quad \triangleright$ *All paths from src to tgt*
6: $\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\textsc{FindPlan}(p) \mid p \in \mathit{paths} \land |p| > 1\}$
7: $\quad$ **return** $\mathcal{S}$ $\quad \triangleright$ *Execution plans for T and $\mathcal{A}$*

8: **function** FINDPLAN($p : n_0 \rightarrow_{R_1} n_1 \rightarrow_{R_2} \ldots \rightarrow_{R_k} n_k$)
9: $\quad S \leftarrow$ an empty array of size k with indices starting at 1
10: $\quad$ **for all** $i \in \{1, \ldots, k\}$ **do**
11: $\quad\quad$ **for all** $\beta \in \mathit{bindings}(R_i, n_{i-i})$ **do**
12: $\quad\quad\quad$ **if** $[\![R_i]\!](n_{i-1}, \beta) = n_i$ **then**
13: $\quad\quad\quad\quad S[i] \leftarrow \langle R_i, \beta\rangle$ $\quad \triangleright$ *Recover the binding that engendered n_i*
14: $\quad \mathit{root} \leftarrow \mathit{greatestCommonPrefix}(\{\mathit{idx} \mid \langle R, \langle \mathit{idx}, \beta\rangle\rangle \in S\})$
15: $\quad$ **for all** $i \in \{1, \ldots, k\}$ **do** $\quad \triangleright$ *Drop the common prefix from all indices*
16: $\quad\quad \langle R, \langle \mathit{idx}, \pi\rangle\rangle \leftarrow S[i]$
17: $\quad\quad S[i] \leftarrow \langle R, \langle \mathit{dropPrefix}(\mathit{idx}, \mathit{root}), \pi\rangle\rangle$
18: $\quad \mathit{src} \leftarrow \mathit{dropPrefix}(n_0, \mathit{root})$
19: $\quad$ **return** $\langle S, \mathit{src}, [\![S]\!]\mathit{src}\rangle$ $\quad \triangleright$ *A general execution plan for replaying p*

20: **function** SOLVE(t: term, $\mathcal{R}$: set of well-formed programs)
21: $\quad N \leftarrow \{t\}$ $\quad \triangleright$ *Set of terms reachable from t via the rules in $\mathcal{R}$*
22: $\quad E \leftarrow \{\}$ $\quad \triangleright$ *Edges from N to N, labeled with rules from $\mathcal{R}$*
23: $\quad$ **while** $(\forall n \in N.\, \neg\textsc{Solved}(n))$ **do**
24: $\quad\quad$ **for all** $\mathit{src} \in \{n \in N \mid \forall e \in E.\, n \neq \mathit{source}(e)\}$ **do**
25: $\quad\quad\quad$ **for all** $R \in \mathcal{R}$ **do** $\quad \triangleright$ *Apply all rules to src*
26: $\quad\quad\quad\quad$ **for all** $\mathit{tgt} \in \mathit{fire}(R, \mathit{src})$ **do**
27: $\quad\quad\quad\quad\quad N \leftarrow N \cup \{\mathit{tgt}\}$
28: $\quad\quad\quad\quad\quad E \leftarrow E \cup \{\langle \mathit{src}, \mathit{tgt}\rangle_R\}$
29: $\quad \mathit{paths} \leftarrow \bigcup_{\hat{t} \in N \land \textsc{Solved}(\hat{t})} \mathit{allShortestPaths}(t, \hat{t}, \langle N, E\rangle)$
30: $\quad E \leftarrow \bigcup_{p \in \mathit{paths}} \mathit{pathEdges}(p)$ $\quad \triangleright$ *Edges comprising the shortest paths*
31: $\quad N \leftarrow \{t\} \cup \{n \mid \exists e \in E.\, \mathit{source}(e) = n \lor \mathit{target}(e) = n\}$
32: $\quad$ **return** $\langle N, E\rangle$ $\quad \triangleright$ *Solution graph with all shortest solutions to t*

Figure 5.11: FINDSPECS takes as input a set of example problems T and axioms $\mathcal{A}$, and produces a set of plans $\mathcal{S}$ for composing the axioms into tactics.

its sinks. Potential shortcuts paths represent paths that could potentially be replaced with the application of a single tactic rule.

Theorem 5.20 (Guarantees of Specification Mining)**.** Let T be a set of terms, SOLVED a predicate over terms, and $\mathcal{A}$ a set of rules. If every term in T can be SOLVED using $\mathcal{A}$, then FINDSPECS$(T, \mathcal{A})$ terminates and produces a set $\mathcal{S}$ of plan and term triples with the following properties: (1) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, S is a general execution plan using only rules in $\mathcal{A}$ where $[\![S]\!]\mathit{src} = \mathit{tgt}$, and (2) for every potential shortcut path p from src to tgt in a solution graph for $t \in T$, $\mathcal{A}$, and SOLVED, there is a triple $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$ such that S replays p.

5.5.2 Rule Synthesis

RULESY synthesizes tactics by searching for well-formed, sound programs that satisfy specifications $\langle S, \mathit{src}, \mathit{tgt}\rangle$ produced by FINDSPECS. This search is a form of syntax-guided synthesis [1]: it draws candidate programs from a given syntactic space, and uses an automatic verifier to check if a chosen candidate satisfies the specification. To enable sound, complete, and efficient synthesis, RULESY needs (1) an automatic verifier for its DSL, and (2) a method for pruning the candidate space without omitting any correct implementations. We address both challenges by reformulating the classic syntax-guided synthesis query to exploit the structure of well-formed rule programs and specifications produced by FINDSPECS. This reformulated query, together with the algorithm for solving it, is the key technical contribution of the rule synthesis stage of our system. We illustrate the challenges of classic syntax-guided synthesis for rule programs; show how our *best-implements* query addresses them; and present the FINDRULES algorithm for sound, complete, and efficient solving of this query.

Classic Synthesis for Rule Programs

In our setting, the classic synthesis query takes the form $\exists R.\forall t, \beta.[\![R]\!](t, \beta) \simeq apply(S, t, \beta)$, where R is a well-formed program and S is an execution plan. This represents a query where the input is the specification S and the output is the program R. Existing tools [1, 124, 133] cannot solve this query soundly because it involves verifying candidate programs over terms of unbounded size. But even if we weaken the soundness guarantee to functional correctness over bounded inputs, these tools can fail to find useful rules because the classic query is overly strict for our purposes. We illustrate why with Example 5.21.

Example 5.21. To see why the classic synthesis query is overly strict, consider the specification $\langle S, src, tgt \rangle$ where S is $[\langle \mathbf{A}, \langle [1], I \rangle \rangle, \langle \mathbf{A}, \langle [2], I \rangle \rangle]$, *src* is `(+ (+ 0 x) (+ 0 y))`, *tgt* is `(+ x y)`, and **A** is the additive identity axiom (Figure 5.1a). The plan S specifies a general tactic for transforming a term of the form `(`op `(+ 0` e_1`) (+ 0` e_2`))` to the term `(`op e_1 e_2`)`, where op is any binary operator in our algebra DSL. Such a tactic cannot be expressed as a well-formed program (Definition 5.8). But many useful specializations of this tactic are expressible, e.g.:

```
with (+ a@(+ x@num u) b@(+ y@num v) etc):
if x = 0 and y = 0:
replace a with u and replace b with v
```

Since we aim to generate useful tactics for domain model optimization, an ideal synthesis query for RuleSy would admit many such specialized yet widely applicable implementations of S.

In addition to being overly strict, the classic synthesis query also leads to intractable search spaces in our setting. The generic sketch for rule programs shown above defines a space of $O(2^{|p|} * 2^{|c|} * 2^{|a|})$ candidate programs, where $|p|$, $|c|$, and $|a|$ are the number of control bits used for selecting expressions (of some finite depth) from the pattern, constraint, and action grammars. For the candidate space to include realistic rules (e.g., Figure 5.16), these

control parameters need to be sufficiently large, leading to exponential explosion. An ideal synthesis query for RuleSy would therefore enable the synthesizer to explore an exponentially smaller subset of the generic sketch, without missing any rules that satisfy the query. Furthermore, reasoning about arbitrary bindings (and thus arbitrary tree permuations) is computationally expensive, and an ideal synthesis query would enable the synthesisizer to ignore term isomorphisms and commutativity.

The Best-Implements Synthesis Query

To address the challenges of classic synthesis, we reformulate the synthesis task for RuleSy as follows: given $\langle S, \mathit{src}, \mathit{tgt}\rangle$, find *all* rules R that fire on *src* to produce *tgt*, that are sound with respect to S, and that capture a locally maximal subset of the behaviors specified by S. We say that such rules *best implement* S for $\langle \mathit{src}, \mathit{tgt}\rangle$ (Definition 5.22), and we search for them with the FindRules algorithm (Figure 5.12), which is a sound and complete synthesis procedure for the best-implements query (Theorem 5.27).

Definition 5.22 (Best Implementation)**.** Let S be an execution plan that replays a path from a term *src* to a term *tgt*. A well-formed rule R with pattern p *best implements* S *for* $\langle \mathit{src}, \mathit{tgt}\rangle$ if $[\![R]\!](\mathit{src}, I_\beta) = \mathit{tgt}$ and $\forall t, \beta. \mathcal{P}[\![p]\!](t, \beta) \neq \bot \implies [\![R]\!](t, \beta) \simeq \mathit{apply}(S, t, \beta)$. That is, R maps *src* to *tgt* under the identity binding, and R matches S's behavior (up to isomorphism) for any term/binding on which R's pattern matches.

The best implementation is defined with respect to a particular pattern. We want to allow for useful specializations by weakening the pattern while maintaining maximally general rules by enforcing—on terms where the pattern matches—that the condition and action mimic the plan's behavior (up to isomorphism). To ensure we do not miss any such rules, our approach will include enumerating over all (finitely many) possible patterns for a given specification.

The best implementation is defined over all term/binding inputs, but reasoning about isomorphic terms is computationally expensive. To make synthesis tractable, we remove the need for such reasoning by further restricting our synthesis query to not have to reason

about arbitrary bindings. This more limited definition of *best implements* (Definition 5.23) only describes program behavior for the identity binding, restricting the query to the inputs where the program's semantics matches the plan's semantics exactly. The construction of the DSL makes such a reduction sound; Theorem 5.24 shows that any program that best implements an execution plan under the identity binding is sound in general.

Definition 5.23 (Best Implementation under the Identity Binding)**.** Let S be an execution plan that replays a path from a term *src* to a term *tgt*. A well-formed rule R with pattern p *best implements* S *for* $\langle src, tgt \rangle$ *under the identity binding* if $[\![R]\!](src, I_\beta) = tgt$ and $\forall t.\mathcal{P}[\![p]\!](t, I_\beta) \neq \bot \implies [\![R]\!](t, I_\beta) = [\![S]\!]t$. That is, R maps *src* to *tgt* under the identity binding, and R matches S's behavior for any term to which R's pattern matches under the identity binding.

Theorem 5.24 (Soundness of Best Implements)**.** Let $\mathcal{A}$ be a set of axioms, $\langle S, src, tgt \rangle$ a specification where S uses only rules in $\mathcal{A}$, and R be a program in the algebra DSL that best implements S for $\langle src, tgt \rangle$ under the identity binding. Then R is sound with respect to $\mathcal{A}$.

1: **function** FINDRULES(S: plan, *src*, *tgt*: terms, $\bar{k}$: ints)
2: $p_0 \leftarrow termToPattern(s)$ ▷ *Most refined pattern that matches s*
3: $\mathcal{R} \leftarrow \bigcup_{p_0 \sqsubseteq p} \text{FINDRULE}(p, S, src, tgt, \bar{k})$
4: **return** $\mathcal{R}$ ▷ *Rules that best implement S for* $\langle src, tgt \rangle$

5: **function** FINDRULE(p: pattern, S: plan, s,t: terms, $\bar{k}$: ints) ▷ $[\![p]\!]s \wedge t = [\![S]\!]s$
6: $bind \leftarrow \lambda\tau.\mathcal{P}[\![p]\!](\tau, I_\beta)$
7: $C \leftarrow \text{WELLFORMEDCONSTRAINTHOLE}(p, \bar{k})$ ▷ *Condition sketch*
8: $A \leftarrow \text{WELLFORMEDCOMMANDHOLES}(p, \bar{k})$ ▷ *Action sketch*
9: $\mathbb{T} \leftarrow \{t \mid \mathcal{M}[\![p]\!]t\}$ ▷ *Symbolic representation of all terms that satisfy p*
10: $c \leftarrow \text{CEGIS}(\mathcal{C}[\![C]\!]bind(s) \wedge (\forall_{\tau \in \mathbb{T}} \mathcal{C}[\![C]\!]bind(\tau) \iff [\![S]\!]\tau \neq \bot))$
11: $a \leftarrow \text{CEGIS}(\mathcal{A}[\![A]\!](s, bind(s)) = t \wedge (\forall_{\tau \in \mathbb{T}} [\![S]\!]\tau \neq \bot \implies \mathcal{A}[\![A]\!](\tau, bind(\tau)) = [\![S]\!]\tau))$
12: **return** $\{$`with` p: `if` c: `do` $a \mid c \neq \bot \wedge a \neq \bot\}$

Figure 5.12: FINDRULES takes as input a bound $\bar{k}$ on program size and an execution plan S that replays a path from *src* to *tgt*. Given these inputs, it synthesizes all rule programs of size $\bar{k}$ that best implement S with respect to *src* and *tgt*.

Sound and Complete Verification.

Verifying that a program R best implements a plan S (under the identity binding) involves checking that R produces the same output as S on all terms t to which R applies under the identity binding. The verification task is therefore to decide the validity of the formula $\forall t.\mathcal{P}[\![pattern(R)]\!](t, I_\beta) \neq \bot \implies [\![R]\!](t, I_\beta) = [\![S]\!]t$. We do so by observing that this formula has a small model property when R is well-formed (Definition 5.8): if the formula is valid on a carefully constructed finite set of terms $\mathbb{T}$, then it is valid on all terms (proved in Lemma 5.34). At a high level, $\mathbb{T}$ consists of terms that satisfy R's pattern in a representative fashion. For example, $\mathbb{T} = \{x\}$ for the pattern `var` because all terms that satisfy `var` are isomorphic up to alpha-renaming (i.e., changing variable names does not change the meaning of a term). Encoding the set $\mathbb{T}$ symbolically (rather than explicitly) enables FindRules to discharge its verification task efficiently with an off-the-shelf SMT (Satisfiability Modulo Theories) solver [93]. Our implementation is built on top of an off-the-shelf synthesizer using counter-example guided inductive synthesis (CEGIS) [133].

Efficient Search.

As noted in Definition 5.22, the best implementation is defined with respect to a particular pattern. As part of synthesis, RuleSy must enumerate over all potential patterns to find all potential rules. FindRules accelerates this process by exploiting the observation that a best implementation of $\langle S, src, tgt\rangle$ must fire on *src* to produce *tgt*. Thus, any best-implementing rule must have a pattern that accepts *src*. The RuleSy DSL's pattern language is defined in such a way that there are only finitely many patterns that accept s. Moreover, there is a unique "most specific" pattern that accepts *src* while accepting the minimal number of terms. We formalize this with the notion of *pattern refinement* in Definition 5.25. Intuitively, a pattern p_1 refines another pattern p_2 if p_1 is a specialization of p_2 in that it matches a subset of the terms that p_2 does. The pattern language of RuleSy forms a semi-lattice of refinement, with `_` at the top. Any sound program's pattern can be replaced with a

more refined pattern while preserving soundness. We exploit this structure of refinement for efficient pattern enumeration.

Specifically, if a rule accepts a term s, its pattern must be refined by the (unique) most refined pattern p_0 (line 2) that accepts s (as a consequence of Theorem 5.26). To construct p_0, we replace each value in s with the corresponding most refined matcher for that value, and a term `(`o t_1 ... t_k`)` with `(`o $\mathit{termToPattern}(t_1)$ ... $\mathit{termToPattern}(t_k)$`)`. For algebra, that means we replace variables with `var` and numeric literals with `num`. We additionally attach a fresh variable to every part of the pattern, later removing unreferenced variables from synthesized programs.

Since p_0 refines finitely many patterns p (from Definition 5.25), we can enumerate all of them (line 3). Once p is fixed through enumeration, FindRule can efficiently search for a best implementation R with that pattern, by using an off-the-shelf synthesizer [133] to perform two independent searches for R's condition (line 10) and action (line 11). These two searches explore an exponentially smaller candidate space than a single search for the condition and action, without missing any correct rules (Theorem 5.27). Given a pattern p, FindRule explores a space of size $O(2^{|c|} + 2^{|a|})$, while the generic sketch contains $O(2^{|c|} * 2^{|a|})$ candidates. As a result, FindRules is asymptotically more efficient than classic syntax-guided synthesis, searching a space of $O(2^{|p|} * (2^{|c|} + 2^{|a|}))$ rather than $O(2^{|p|} * 2^{|c|} * 2^{|a|})$ candidates.

Definition 5.25 (Pattern Refinement)**.** A pattern p_1 *refines* a pattern p_2 if $p_1 \sqsubseteq p_2$, where $\sqsubseteq$ is defined as follows (implicitly quantified over all patterns):

- $p \sqsubseteq p$
- $x@p \sqsubseteq p$
- $p \sqsubseteq _$
- $m_1 \sqsubseteq m_2$ if $\forall t.\, \mathit{matches}[\![m_1]\!]t \implies \mathit{matches}[\![m_2]\!]t$
  (for algebra, this means `num` $\sqsubseteq$ `base` and `var` $\sqsubseteq$ `base`)

- `(`o p_1 ... p_k`)` $\sqsubseteq$ `(`o q_1 ... q_k`)` if $p_i \sqsubseteq q_i$ for all $i \in \{1, \ldots, k\}$
- `(`o p_1 ... p_n`)` $\sqsubseteq$ `(`o q_1 ... q_k **`etc`**`)` if $n \geq k$ and $p_i \sqsubseteq q_i$ for all $i \in \{1, \ldots, k\}$.

Note that the pattern `_` (which matches any term) refines nothing but itself and everything refines it. From the definition of refinement we can see that any given pattern refines finitely many other patterns.

Theorem 5.26 (Partial Ordering of Pattern Refinement)**.** Let p_1, p_2 be patterns, c be a condition, a be an action, and R_1, R_2 be the well-formed programs $R_1 =$ **`with`** p_1`:` **`if`** c`:` **`then`** a and $R_2 =$ **`with`** p_2`:` **`if`** c`:` **`then`** a. If p_1 refines p_2 ($p_1 \sqsubseteq p_2$), then R_2 is as general as R_1 and has identical behavior for applicable inputs to R_1. That is, $\forall t, \beta.\, [\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$.

Theorem 5.27 (Guarantees of Rule Synthesis)**.** Let S be an execution plan where $[\![S]\!]\mathit{src} = \mathit{tgt}$, and $\bar{k}$ a bound on the size of rule programs. FINDRULES($S, \mathit{src}, \mathit{tgt}, \bar{k}$) returns a set of rules $\mathcal{R}$ with the following properties: (1) every $R \in \mathcal{R}$ best implements S for $\langle \mathit{src}, \mathit{tgt} \rangle$; (2) $\mathcal{R}$ includes a sound rule R of size $\bar{k}$ if one exists; and (3) for every pattern p that refines or is refined by R's pattern, $\mathcal{R}$ includes a sound rule with pattern p and size $\bar{k}$ if one exists.

5.5.3 Rule Set Optimization

After synthesizing the tactics $\mathcal{T}$ for the examples T and axioms $\mathcal{A}$, RULESY applies discrete optimization to find a subset of $\mathcal{A} \cup \mathcal{T}$ that minimizes the objective function f. We formulate this optimization problem in a way that guarantees termination. In particular, our OPTIMIZE algorithm (Figure 5.13) returns a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that can solve each example in T and that minimize f over all *shortest*-solution graphs for T and $\mathcal{A} \cup \mathcal{T}$ (Theorem 5.28). Restricting the optimization to shortest solutions enables us to decide whether an arbitrary rule set $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ can solve an example $t \in T$ without having to invoke SOLVE($t, \mathcal{R}$), which may not terminate for an arbitrary term t and rule set $\mathcal{R}$ in our DSL.

The OPTIMIZE procedure works in three steps. First, for each example term $t \in T$, lines 4–5 construct a solution graph $\langle N, E_{\mathcal{A}} \cup E_{\mathcal{T}} \rangle$ that contains shortest solutions for t and

```
1:  function OPTIMIZE(T: set of terms, 𝒜,𝒯: set of rules, f: objective)
2:    𝒢_{𝒜∪𝒯} ← {}
3:    for t ∈ T such that ¬SOLVED(t) do
4:      ⟨N, E_𝒜⟩ ← SOLVE(t, 𝒜)                          ▷ Solve with axioms
5:      E_𝒯 ← ⋃_{R∈𝒯} ⋃_{s,t∈N} {⟨s, t⟩ | t ∈ fire(R, s)}     ▷ Tactic edges
6:      𝒢_{𝒜∪𝒯} ← 𝒢_{𝒜∪𝒯} ∪ {⟨N, E_𝒜 ∪ E_𝒯⟩}
7:    f_∅ ← λℛ.𝒢. if ⟨∅, ∅⟩ ∈ 𝒢 then return ∞ else return f(ℛ, 𝒢)
8:    return min_{ℛ⊆𝒜∪𝒯} f_∅(ℛ, {RESTRICT(G, ℛ) | G ∈ 𝒢_{𝒜∪𝒯}})
9:  function RESTRICT(⟨N, E⟩: solution graph, ℛ: set of rules)
10:   t ← source of the graph ⟨N, E⟩
11:   E_ℛ ← {⟨src, tgt⟩_R ∈ E | R ∈ ℛ}              ▷ Edges with labels in ℛ
12:   paths ← ⋃_{t̂∈N∧SOLVED(t̂)} allPaths(t, t̂, ⟨N, E_ℛ⟩)
13:   E ← ⋃_{p∈paths} pathEdges(p)
14:   N ← {n | ∃e ∈ E . source(e) = n ∨ target(e) = n}
15:   return ⟨N, E⟩                 ▷ Solution graph for t and ℛ or ⟨∅, ∅⟩
```

Figure 5.13: OPTIMIZE takes as input a set of terms T, axioms $\mathcal{A}$ for reducing T, tactics $\mathcal{T}$ synthesized from $\mathcal{A}$ and T using FINDRULES and FINDSPECS, and an objective function f. The output is a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that minimizes f.

all subsets of $\mathcal{A} \cup \mathcal{T}$. Next, line 7 creates a function $f_\emptyset$ that takes as input a set of rules $\mathcal{R}$ and a set of graphs $\mathcal{G}$, and produces ∞ if $\mathcal{G}$ contains the empty graph (indicating that $\mathcal{R}$ cannot solve some term in T) and $f(\mathcal{R}, \mathcal{G})$ otherwise. Finally, line 8 searches for $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that minimizes f over $\mathcal{G}_{\mathcal{A}\cup\mathcal{T}}$. This search relies on the procedure RESTRICT$(G, \mathcal{R})$ to extract from G a solution graph for $t \in T$ and $\mathcal{R}$ if one is included, or the empty graph otherwise. For linear objectives f, the search can be delegated to an optimizing SMT solver [93]. For other objectives (e.g., Figure 5.1c), we use a greedy algorithm to find a locally minimal solution (thus weakening the optimality guarantee in Theorem 5.28).

Theorem 5.28 (Guarantees of Model Optimization)**.** Let $\mathcal{T}$ be a set of tactics synthesized by RULESY for terms T and axioms $\mathcal{A}$, and let f be a total function from sets of rules and solution graphs to positive real numbers. OPTIMIZE$(T, \mathcal{A}, \mathcal{T}, f)$ returns a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that can solve each term in T, and for all such $\mathcal{R}' \subseteq \mathcal{A} \cup \mathcal{T}$, $f(\mathcal{R}, \{\text{SOLVE}(t, \mathcal{R}) \mid t \in$

$T\}) \leq f(\mathcal{R}', \{\textsc{Solve}(t, \mathcal{R}') \mid t \in T\})$.

5.5.4 Implementation

RuleSy is built on top of Rosette [133], a language for bounded verification and synthesis based on SMT [93]. This section highlights three key details of our implementation.

Term Canonicalization. Our implementation of Solve partially canonicalizes terms so that isomorphic terms map to the same syntactic representation and share a node in the solution graph. This makes Solve exponentially more time and space-efficient, but slightly complicates the implementation of FindPlan, which must de-canonicalize paths before computing plans for them.

Plan Filtering. We implement FindSpecs to filter out plans that apply axioms to disjoint subterms. Such plans are unlikely to yield macros that perform fewer tree edits than the plan's axioms, so filtering them biases the system toward producing more useful macros.

Plan Generalization. As discussed, our definition of plans (Definition 5.14) lays out the idea of a *general* plan—one in which the steps have the empty index as their greatest common prefix. This is for cases where a training example describes behavior on a subterm; RuleSy automatically generalizes the plan so the synthesized rule applies in many situations. In practice, there are further kinds of plan generalization that can be done, most notably around permutations. RuleSy reorders plans according to commutative operators to enable the most general possible pattern. As an example, consider the following training example for combine like terms: `(+ 5 (* 2 x) (* 3 x))` $\rightarrow$ `(+ 5 (* 5 x))`. A naïve specification drawn from this example would lead to the pattern `(+ _ (* num _) (* num _) etc)`. Our implementation of specification mining instead rearranges this example to `(+ (* 2 x) (* 3 x) 5)` $\rightarrow$ `(+ (* 5 x) 5)`, leading to synthesizing the more general pattern of `(+ (* num _) (* num _) etc)`.

Decomposing Actions. In addition to decomposing condition and action synthesis as described in Section 5.5.2, our implementation also decomposes action synthesis, when possible, into multiple synthesis queries. This optimization exploits the observation that many rules and plans modify independent subterms of a term; e.g., algebra rules often edit each child of an equality term without editing the root of the term. Such cases can be straightforwardly detected by inspecting which indices are modified in an execution plan. Our synthesis implementation decomposes actions into independent sub-actions (if any) and synthesizes their commands separately. As with decomposing condition and action synthesis, this optimization results in exponentially smaller search spaces, and was necessary to tractably synthesize some of our results.

Finding Concise Rules. An additional desirable property for synthesized tactics—beyond soundness, usefulness, and generality—is conciseness. In the interest of, for example, making rules easier to apply and to teach, we want our tactics to be maximally concise. Exactly how to express a rule in the most readable or concise way involves complex human factors, but we can approximate this quantitatively with a measure of program size. The smaller the program, the more concise it is.

Finding maximally small programs is a straightforward extension to the FindRule algorithm from Section 5.5.2. Upon finding a sound condition and action for a tactic, we can repeat the CEGIS procedures with the additional constraint that the program must be strictly smaller than the one already found. Doing this iteratively, trying to find successively smaller programs, until failure, we will be left with the smallest possible sound rule. This basic method of superoptimization [88] can be implemented efficiently on top of our CEGIS system using incremental SMT solving.

5.6 Generalizing Beyond Algebra

The formalisms presented thus far have been specific to solving algebraic equations over integers. However, RuleSy as presented in this chapter can apply more generally. For example,

we can apply RuleSy to algebra over reals or other numbers. More broadly, the framework from this chapter can apply to a wide range of domains that fall under the general category of "finding semantic equivalences between expressions represented as abstract syntax trees." In this section, we revisit and generalize the formalisms and algorithms necessary to generalize RuleSy, using the semantic proof domain from Chapter 4 as a concrete example of an alternate domain.

When instantiating RuleSy for a new domain, the implementor primarily has to change the syntax of terms and the bits of the DSL related to matching on those terms. In particular, the operators and leaf node values can change. For example, Figure 5.14 shows the term syntax for the semantic proof domain (contrast with Figure 5.5). Values still include variables, but constants have changed from integers to booleans.

$$\begin{array}{rcl} \textit{term} & ::= & \texttt{(}\textit{op}\ \{\textit{term}\}^*\texttt{)} \mid \textit{value} \\ \textit{value} & ::= & \top \mid \bot \mid \text{variable identifier} \\ \textit{op} & ::= & \textit{commutativeop} \mid \mathtt{I}\!\models \mid \mathtt{I}\!\not\models \mid \Rightarrow \mid \neg \\ \textit{commutativeop} & ::= & \texttt{branch} \mid \texttt{facts} \mid \wedge \mid \vee \mid \Leftrightarrow \end{array}$$

Figure 5.14: Syntax for terms in the semantic proof domain. The notation $\{\textit{form}\}^*$ means zero or more repetitions of the given form.

Chaning the operators and values obviously changes the available patterns and **`apply`** constructs. The syntax of the *matcher* constructs must change to support the new values. For semantic proofs, `num` is replaced with `bool` to support the change in constant values. Likewise, both of the semantic functions *matches* and *operator* must be changed for the new operators and matchers. Figure 5.15 shows these functions for the semantic proof domain.

In order for RuleSy to be able to synthesize and verify programs, there are some restrictions on these *matcher* constructs:

1. Matchers may match only to *value* terms.

2. For every *value* v, there must be a unique matcher m that accepts v and is not re-

$$
\begin{aligned}
operator[\![\wedge]\!](x_1, \dots, x_k) &= \textstyle\bigwedge_{i \in \{1,\dots,k\}} x_i \\
operator[\![\vee]\!](x_1, \dots, x_k) &= \textstyle\bigvee_{i \in \{1,\dots,k\}} x_i \\
matches[\![\texttt{bool}]\!]t &= t \text{ is a boolean literal } \textit{value}. \\
matches[\![\texttt{var}]\!]t &= t \text{ is a variable identifier } \textit{value}. \\
matches[\![\texttt{base}]\!]t &= matches[\![\texttt{bool}]\!]t \vee matches[\![\texttt{var}]\!]t
\end{aligned}
$$

Figure 5.15: The semantic functions that change for the semantic proof domain.

fined (Definition 5.25) by any other pattern. That is, $\forall v.\, \exists m.\, \forall m'.\, matches[\![m]\!]v \implies matches[\![m']\!]v$. This ensures that pattern enumeration (Figure 5.12) works. One straightforward way to ensure this is to have partition the value space and have one matcher per partition.

3. For every *matcher*, there must be a way to verify programs over all possible terms accepted by that matcher. This is necessary to prove the soundess of FindRule; the parts of the proof for the algebra domain's matchers are in Lemma 5.34. One sufficient way to ensure this is to use finite domains (e.g., we use integers up to a fixed maximum) or finitizable domains (e.g., variables are equivalent up to alpha-renaming so the domain only needs to be as big as the number of variables that show up in the pattern).

There are additional restrictions on the behaviors of operators; broadly speaking, they must be decidable. This is the case for integer/boolean arithmetic over finite domains and equality over any domain.

5.7 Evaluation

To evaluate RuleSy's effectiveness at synthesizing domain models, we answer the following four research questions:

RQ 1. Can RuleSy's synthesis algorithm recover standard tactics from a textbook and discover new ones?

RQ 2. Can RuleSy's optimization algorithm recover textbook domain models and discover variants of those models that optimize different objectives?

RQ 3. Can RuleSy support different educational domains?

The first two questions assess the quality of RuleSy's output by comparing the synthesized tactics and domain models to a textbook [31] written by domain experts. The third question assesses the generality of our approach. We conducted two case studies to answer these questions, finding positive answers to each. The implementation source code and evaluation data are available online.[10]

5.7.1 Case Study with Algebra (RQ 1–3)

We performed three experiments in the domain of K-12 algebra to answer RQ 1–2. Each experiment was executed on an Intel 2$^{\text{nd}}$ generation i7 processor with 8 virtual threads. The system was limited to a synthesis timeout of 20 minutes per mined specification. The details and results are presented below.

Quality of Synthesized Rules (RQ 1). To evaluate the quality of the rules synthesized by RuleSy, we apply the system to the examples (P_T in Table 5.1a) and axioms (Table 5.1b) from a standard algebra textbook [31], and compare system output (607 tactics) to the tactics from the textbook. Since the book demonstrates rules on examples rather than explicitly, determining which rules are shown involves some interpretation. For example, we interpret the transformation $5x + 2 - 2x = 2x + 14 - 2x \rightarrow 3x + 2 = 14$ as demonstrating two independent tactics, one for each side of the equation, rather than one tactic with unrelated subparts. The second column of Table 5.2 lists all the tactics presented in the book. We find that RuleSy recovers each of them or a close variation.

In addition to recovering textbook tactics, RuleSy finds interesting variations on rules commonly taught in algebra class. Figure 5.16 shows an example, which isolates a vari-

[10] `https://github.com/edbutler/algebra-rule-synthesis`

ID	Source	Count
P_T	Chapter 2, Sections 1–4 of Hall et al. [31]	92

(a) Example problems.

ID	Name	Example
A	Additive Identity	$x + 0 \rightarrow x$
B	Adding Constants	$2 + 3 \rightarrow 5$
C	Multiplicative Identity	$1x \rightarrow x$
D	Multiplying by Zero	$0(x + 2) \rightarrow 0$
E	Multiplying Constants	$2 * 3 \rightarrow 6$
F	Divisive Identity	$\frac{x}{1} \rightarrow x$
G	Canceling Fractions	$\frac{2x}{2y} \rightarrow \frac{x}{y}$
H	Multiplying Fractions	$3\left(\frac{2x}{4}\right) \rightarrow \frac{(2*3)x}{4}$
I	Factoring	$3x + 4x \rightarrow (3 + 4)x$
J	Distribution	$(3 + 4)x \rightarrow 3x + 4x$
K	Expanding Terms	$x \rightarrow 1x$
L	Expanding Negatives	$-x \rightarrow -1x$
M	Adding to Both Sides	$x + -1 = 2 \rightarrow x + -1 + 1 = 2 + 1$
N	Dividing Both Sides	$3x = 2 \rightarrow \frac{3x}{3} = \frac{2}{3}$
O	Multiplying Both Sides	$\frac{x}{3} = 2 \rightarrow 3\left(\frac{x}{3}\right) = 2 * 3$

(b) Axioms.

Table 5.1: Example problems (a) and axioms (b) for the algebra case study.

able from a negated fraction and an addend. This rule composes 9 axioms, demonstrating RuleSy's ability to discover advanced tactics.

```
# Isolate a variable from a negated fraction and an addend:
# (= (+ (− (/ (∗x ...) b)) c) e) → (= (∗x ...) (∗b (− c e)))
def MBALNGOHG:
    with (= left@(+ (- (/ num@(* var etc) den@base))
                       x@num)
            right@_):
    if true:
    do replace left with num and
       replace right with (* den (- x right))
```

Figure 5.16: A custom algebra tactic discovered by RuleSy (variable names were chosen by the author for presentation).

Quality of Synthesized Domain Models (RQ 2). We next evaluate RuleSy's ability to recover textbook domain models along with variations that optimize different objectives. An important part of creating domain models for educational tools (and curricula in general) is choosing the *progression*—the sequence in which different concepts (i.e., rules) should be learned. We use RuleSy and the objective function shown in Figure 5.1c to find a progression of optimal domain models for the problems (P_T in Table 5.1a) and axioms (Table 5.1b) in [31], and we compare this progression to the one in the book.

We create a progression by producing a sequence of domain models for Sections 1–4 of Chapter 2 in [31]. Every successive model is constrained to be a superset of the previous model(s): students keep what they learned and use it in subsequent sections. To generate a domain model D_n for section n, we apply RuleSy's optimizer to the exercise problems from section n; the objective function in Figure 5.1c with $\alpha \in \{.05, .125, .25\}$; and all available rules (axioms and tactics), coupled with the constraint that $D_1 \cup \ldots \cup D_{n-1} \subseteq D_n$.

Table 5.2 shows the resulting progressions of optimal domain models for [31], along with the rules that are introduced in the corresponding sections. For each rule presented in a

Section	Textbook Rules	ODM $\alpha = 0.05$	ODM $\alpha = 0.125$	ODM $\alpha = 0.25$
2-1	**B**, **M**, **N**, **G**, **O**, **BA**, **HG**	**M**, **A**, **K**, **L**, **NG**, **LNG**, **OHG**, **IBD**, **MBA**	**NG**, **OHG**, **MBA**	**NG**, **OHG**, **MBA**
2-2	**L**, **E**	**LE**	**LNG**, **LE**	**E**, **L**
2-3	**J**, **IB**, **KIB**, **JB**	**E**, **J**, **KIB**, **IB**, **BMBA**	**E**, **K**, **L**, **J**, **B**, **IB**	**I**, **K**, **J**, **B**
2-4	**LEIBDA**, **LEIB**	**C**, **BD**, **LEIB**, **MLEI**	**M**, **C**, **BD**, **IBD**, **LEIB**, **MLEI**	**M**, **C**, **D**, **LEIB**

Table 5.2: A textbook [31] progression and the corresponding optimal domain models found by RuleSy, using 3 settings of α (Figure 5.1c). Row i shows the rules that the ith model adds to the preceding models.

section, the corresponding optimal model for $\alpha = .05$ contains either the rule itself or a close variation. Increasing α leads to new domain models that emphasize rule set complexity over solution efficiency. This result demonstrates that RuleSy can recover textbook domain models, as well as find new models that optimize different objectives.

5.7.2 Case Study with Propositional Logic (RQ 3)

To evaluate the extensibility and generality of RuleSy, we applied it to the domain of semantic proofs for elementary propositional logic theorems, as described in Chapter 4. Many students have trouble learning how to construct proofs [53], so custom educational tools could help by teaching a variety of proof strategies.

We applied this instantiation of RuleSy to the axioms (Table 4.1) and proof exercises (3 in total) from a textbook [20]. The system synthesized 85 rules in 72 minutes. The resulting rules includes interesting general proof rules for each of the exercises. For example, given $(p \wedge (p \to q)) \to q$, RuleSy mines and synthesizes the modus ponens tactic shown in Figure 4.6. These results show RuleSy's applicability and effectiveness extend beyond the domain of K-12 algebra.

5.8 Related Work

Automated Rule Learning. Automated rule learning is a well-studied problem in Artificial Intelligence and Machine Learning. RuleSy is most closely related to rule learning approaches in discrete planning domains, such as cognitive architectures [74]. Its learning of tactics from axioms is similar to chunking in SOAR [71], knowledge compilation in ACT [7], and macro-learning from AI planning [69]. But unlike these systems, RuleSy can learn rules for transforming problems represented as trees, and express objective criteria over rules and solutions.

Inductive Logic Programming. Within educational technology, researchers have investigated automated learning of rules and domain models for intelligent tutors [67]. Previous efforts have focused on applying inductive logic programming to learn a domain model from a set of expert solution traces [90, 77, 78, 61, 111]. RuleSy, in contrast, uses a small set of axioms and example problems to synthesize an exhaustive set of sound tactics, and it searches the axioms and tactics for a model that optimizes a desired objective.

Program Synthesis. Prior educational applications of program synthesis and automated search include problem and solution generation [49, 3], hint and feedback generation [75, 113, 131], and checking of student proofs [76]. RuleSy solves a different problem: generating condition-action rules and domain models. General approaches to programming-by-example [104, 79] have investigated the problem of learning useful programs from a small number of input-output examples, with no general soundness guarantees. RuleSy, in contrast, uses axioms to verify that the synthesized programs are sound for all inputs, relying on examples only to bias the search toward useful programs (i.e., tactics that shorten solutions).

Term Rewrite Systems. RuleSy helps automate the construction of rule-based domain models, which are related to term rewrite systems [38]. Our work can be seen as an approach for learning rewrite rules, and selecting a cheapest rewrite system that terminates on a given

finite set of terms. RuleSy terms are a special case of recursive data types, which have been extensively studied in the context of automated reasoning [99, 128, 11]. Our rule language is designed to support effective automated reasoning by reduction to the quantifier-free theory of bitvectors.

5.9 Conclusion

This chapter presented an instantiation of RuleSy for rewrite rules of semantic term equivalence RuleSy is based on new algorithms for mining specifications of tactic rules from examples and axioms, synthesizing sound implementations of those specifications, and selecting an optimal domain model from a set of axioms and tactics. Thanks to these algorithms, RuleSy efficiently recovers textbook tactic rules and models for K-12 algebra, discovers new ones, and generalizes to other domains. As the need for tools to support personalized education grows, RuleSy can help tool developers rapidly create domain models that target individual students' educational objectives.

Acknowledgements. This research was supported in part by NSF CCF-1651225, 1639576, and 1546510; Oak Foundation 16-644; and Hewlett Foundation.

5.10 Appendix: Proofs of Theorems

In this section, we restate and provide proofs for all theorems stated earlier in the chapter.

Lemma 5.29. If a term t can be transformed to a Solved form $\hat{t}$ by applying the rules $\mathcal{R}$, then $\text{Solve}(t, \mathcal{R})$ terminates and returns a solution graph that represents all shortest solutions to t using $\mathcal{R}$.

Proof. Solve performs breadth-first exploration of the set of all terms reachable from t by applying the rules $\mathcal{R}$ (lines 23–28). This ensures that a reduced term $\hat{t}$ will either be reached after the smallest number of rule applications, or the search diverges. If the search terminates, the multigraph $\langle N, E\rangle$ contains all shortest sequences of rule applications that reduce t to

$\hat{t}$ at line 29. Lines 30–31 preserve these sequences, while eliminating all cycles from $\langle N, E\rangle$ and ensuring that the reduced terms $\hat{t}$ are the sole sinks in $\langle N, E\rangle$. The graph returned at line 32 therefore satisfies the definition of a solution graph, completing the proof. □

Lemma 5.30. Given a path p in a solution graph, FINDPLAN produces a triple $\langle S, \mathit{src}, \mathit{tgt}\rangle$ where S is a general execution plan S that replays p and $[\![S]\!]\mathit{src} = \mathit{tgt}$.

Proof. The proof consists of three parts. First, we show that $[\![\langle R_i, \beta\rangle]\!]n_{i-1} = n_i$ at line 13. Because $\langle n_{i-1}, n_i\rangle_{R_i}$ is an edge in a solution graph, it follows from Definition 5.16 and the definition of *fire* (Figure 4.8) that (on line 11) $\mathit{bindings}(R_i, n_{i-i})$ contains a binding β such that $[\![R_i]\!](n_{i-1}, \beta) = n_i$. That binding is assigned to $S[i]$ at line 13. Next, by induction on i and Definition 5.14, we conclude that $[\![S]\!]n_0 = n_k$ at line 14 and S replays p (Definition 5.17). Finally, observe that *root* holds the longest common prefix of the step indices $\mathit{idx}_1, \ldots, \mathit{idx}_k$ for the steps in S. Since the loop at lines 14–17 drops *root* from these indices and from the starting state, the plan S at line 19 is general (by Definition 5.14), and $n_k = \mathit{replace}(n_0, \mathit{root}, [\![S]\!]\mathit{ref}(n_0, \mathit{root}))$ so S replays p. Furthermore, from line 32, in the returned tuple $\langle S, \mathit{src}, \mathit{tgt}\rangle$, $[\![S]\!]\mathit{src} = \mathit{tgt}$. □

Theorem 5.20 (Guarantees of Specification Mining)**.** Let T be a set of terms, SOLVED a predicate over terms, and $\mathcal{A}$ a set of rules. If every term in T can be SOLVED using $\mathcal{A}$, then FINDSPECS$(T, \mathcal{A})$ terminates and produces a set $\mathcal{S}$ of plan and term triples with the following properties: (1) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, S is a general execution plan using only rules in $\mathcal{A}$ where $[\![S]\!]\mathit{src} = \mathit{tgt}$, and (2) for every potential shortcut path p from *src* to *tgt* in a solution graph for $t \in T$, $\mathcal{A}$, and SOLVED, there is a triple $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$ such that S replays p.

Proof. Termination follows from Lemma 5.29 and the fact that all loops in FINDSPECS and FINDPLAN iterate over finite structures. Correctness (1) follows from line 6 of FINDSPECS, Definition 5.14, and Lemmas 5.29 and 5.30. Completeness (2) follows from lines 3–6, Lemmas 5.29 and 5.30, and Definitions 5.17 and 5.19. □

Lemma 5.31. Let R be a program, t a term, δ a tree index, and $\langle \gamma, \pi \rangle$ a binding where $[\![R]\!](\mathit{ref}(t, \delta), \langle \gamma, \pi \rangle) \neq \bot$. Then, $\mathit{replace}(t, \delta, [\![R]\!](\mathit{ref}(t, \delta), \langle \gamma, \pi \rangle)) = [\![R]\!](t, \langle \mathit{concat}(\delta, \gamma), \pi \rangle)$.

Proof. This follows from the semantics of patterns (Figure 5.8). Let p, c, and a be the pattern, condition and action of R, respectively. We proceed by comparing the application of R to $\mathit{ref}(t, \delta)$ and $\langle \gamma, \pi \rangle$ and of R to t and $\langle \mathit{concat}(\delta, \gamma), \pi \rangle$ and in particular comparing the contexts (σ_1 and σ_2 respectively) of those applications.

By assumption, $[\![R]\!](\mathit{ref}(t, \delta), \langle \gamma, \pi \rangle) \neq \bot$. It follows by the semantics of programs that the context $\sigma_1 = \mathcal{P}[\![p]\!](\mathit{ref}(t, \delta), \langle \gamma, \pi \rangle) \neq \bot$. and further follows by definition of $\mathcal{P}$ that $\mathcal{M}[\![p]\!]\mathit{ref}(\mathit{ref}(t, \delta), \gamma)^{\pi}$ is true. We know (by Definition 5.2) that $\mathit{ref}(\mathit{ref}(t, \delta), \gamma) = \mathit{ref}(t, \mathit{concat}(\delta, \gamma))$, and thus $\mathcal{M}[\![p]\!]\mathit{ref}(t, \mathit{concat}(\delta, \gamma))^{\pi}$ is also true. Therefore, the context $\sigma_2 = \mathcal{P}[\![p]\!](t, \langle \mathit{concat}(\delta, \gamma), \pi \rangle) \neq \bot$.

More specifically, where $t' = \mathit{ref}(\mathit{ref}(t, \delta), \gamma)$, we have:

$$
\begin{aligned}
\sigma_1 &= \{x \mapsto \langle \mathit{concat}(\gamma, \pi^{-1}(i)), \mathit{ref}(t', i) \rangle \mid \langle x, i \rangle \in \mathcal{B}[\![p]\!][]\} \\
\sigma_2 &= \{x \mapsto \langle \mathit{concat}(\mathit{concat}(\delta, \gamma), \pi^{-1}(i)), \mathit{ref}(t', i) \rangle \mid \langle x, i \rangle \in \mathcal{B}[\![p]\!][]\}
\end{aligned}
$$

That is, the contexts for both applications of this pattern differ only in that the addresses of σ_2 are the addresses of σ_1 with δ prepended to them. Both contexts have the same domain, and, for every mapping $x \mapsto \langle i, v \rangle$ in σ_1, $\sigma_2[x] = \langle \mathit{concat}(\delta, i), v \rangle$.

The only place these differing addresses are used is in the semantics of **`remove`** and **`replace`**. So we immediately know that $\mathcal{C}[\![c]\!]\sigma_1 = \mathcal{C}[\![c]\!]\sigma_2$, so both are true. Because actions are applied in parallel to disjoint subtrees, we may assume without loss of generality that a consists of either a single removal or single replacement.

Fist we consider the case where $a =$ **`remove`** x. For the first application, we have that $\mathcal{A}[\![a]\!](\mathit{ref}(t, \delta), \sigma_1) = \mathit{remove}(\mathit{ref}(t, \delta), \mathit{fst}(\sigma_1[x]))$, and for the second, we have that $\mathcal{A}[\![a]\!](t, \sigma_2) = \mathit{remove}(t, \mathit{fst}(\sigma_2[x])) = \mathit{remove}(t, \mathit{concat}(\delta, \mathit{fst}(\sigma_1[x])))$. By the definitions of *remove* and *replace*, for any α, $\mathit{replace}(t, \delta, \mathit{remove}(\mathit{ref}(t, \delta), \alpha)) = \mathit{remove}(t, \mathit{concat}(\delta, \alpha))$. It follows that $\mathit{replace}(t, \delta, \mathcal{A}[\![a]\!](\mathit{ref}(t, \delta), \sigma_1)) = \mathcal{A}[\![a]\!](t, \sigma_2)$.

The other case to consider is where $a = \texttt{replace}\ x\ \texttt{with}\ e$. First, as the semantics of expressions don't reference the parts of the mappings that differ from each other, we know $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$. Then, for the first application, we have that $\mathcal{A}[\![a]\!](\mathit{ref}(t,\delta),\sigma_1) = \mathit{replace}(\mathit{ref}(t,\delta), \mathit{fst}(\sigma_1[x]), \mathcal{E}[\![e]\!]\sigma_1)$, and for the second application we have that $\mathcal{A}[\![a]\!](t,\sigma_2) = \mathit{replace}(t, \mathit{fst}(\sigma_2[x]), \mathcal{E}[\![e]\!]\sigma_2) = \mathit{replace}(t, \mathit{concat}(\delta, \mathit{fst}(\sigma_1[x])), \mathcal{E}[\![e]\!]\sigma_1)$. By the definition of *replace*, for any α, s, $\mathit{replace}(t,\delta,\mathit{replace}(\mathit{ref}(t,\delta),\alpha,s)) = \mathit{replace}(t,\mathit{concat}(\delta,\alpha),s)$. It follows that $\mathit{replace}(t,\delta,\mathcal{A}[\![a]\!](\mathit{ref}(t,\delta),\sigma_1)) = \mathcal{A}[\![a]\!](t,\sigma_2)$, completing both cases.

Bringing everything together, we know that, in both applications, the pattern applies and the condition is true, so the result of the program in each application is the value of applying the action. As we have established that $\mathit{replace}(t,\delta,\mathcal{A}[\![a]\!](\mathit{ref}(t,\delta),\sigma_1)) = \mathcal{A}[\![a]\!](t,\sigma_2)$, it immediately follows that $\mathit{replace}(t,\delta,[\![R]\!](\mathit{ref}(t,\delta),\langle\gamma,\pi\rangle)) = [\![R]\!](t,\langle \mathit{concat}(\delta,\gamma),\pi\rangle)$. ☐

An immediate corollary is that $[\![R]\!](t,\langle\gamma,\pi\rangle) = \mathit{replace}(t,\gamma,[\![R]\!](\mathit{ref}(t,\gamma),\langle[],\pi\rangle))$.

Lemma 5.32. Let R be a program, t be a term, and π be a valid tree permutation for t where $[\![R]\!](t,\langle[],\pi\rangle) \neq \bot$. Then $[\![R]\!](t,\langle[],\pi\rangle) \simeq [\![R]\!](t^\pi, I_\beta)$.

Proof. This proof follows the same structure as Lemma 5.31, where we show that the contexts differ only in the addresses but not the values. Let p, c, and a be the pattern, condition and action of R, respectively.

By assumption, $[\![R]\!](t,\langle[],\pi\rangle) \neq \bot$. It follows by the semantics of programs that the context $\sigma_1 = \mathcal{P}[\![p]\!](t,\langle[],\pi\rangle) \neq \bot$ and further follows by definition of $\mathcal{P}$ that $\mathcal{M}[\![p]\!]t^\pi$ is true. Thus, by the semantics of patterns, $\sigma_2 = \mathcal{P}[\![p]\!](t^\pi, I_\beta) \neq \bot$.

Specifically: we have:

$$\begin{aligned}\sigma_1 &= \{x \mapsto \langle \pi^{-1}(i), \mathit{ref}(t^\pi, i)\rangle \mid \langle x, i\rangle \in \mathcal{B}[\![p]\!][]\} \\ \sigma_2 &= \{x \mapsto \langle i, \mathit{ref}(t^\pi, i)\rangle \mid \langle x, i\rangle \in \mathcal{B}[\![p]\!][]\}\end{aligned}$$

That is, the contexts for both applications of this pattern differ only in that the addresses of σ_1 are the addresses of σ_2 permuted by π^{-1} (or, equivalently, the addresses of σ_2 are the addresses of σ_1 permuted by π). The values are the same in both contexts. Thus,

$\mathcal{C}[\![c]\!]\sigma_1 = \mathcal{C}[\![c]\!]\sigma_2$, so both are true. As in the proof of Lemma 5.31, the only place these differing addresses are used is in the semantics of **remove** and **replace**, and we need to show our conclusion holds for two cases: where a is a single **remove** or a single **replace**.

Fist we consider the case where $a =$ **remove** x. For the first application we have $\mathcal{A}[\![a]\!](t, \sigma_1) = \mathit{remove}(t, \mathit{fst}(\sigma_1[x]))$, while in the second application we have $\mathcal{A}[\![a]\!](t^\pi, \sigma_2) = \mathit{remove}(t^\pi, \mathit{fst}(\sigma_2[x]))$. Because $\mathit{fst}(\sigma_2[x]) = \pi(\mathit{fst}(\sigma_1[x]))$, in both applications we are removing the same (up to isomorphism) subterm, and therefore $\mathcal{A}[\![a]\!](t, \sigma_1) \simeq \mathcal{A}[\![a]\!](t^\pi, \sigma_2)$.

The other case to consider is where $a =$ **replace** x **with** e. First, as the semantics of expressions don't reference the parts of the mappings that differ from each other, we know $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$. This case follows analogously to the case with **remove**, where the the same subterm (up to isormorphism) is being replaced with an identical term, and therefore $\mathcal{A}[\![a]\!](t, \sigma_1) \simeq \mathcal{A}[\![a]\!](t^\pi, \sigma_2)$. □

Lemma 5.33. Let s, t be terms and idx a tree index where $\mathit{ref}(t, \mathit{idx}) \neq \bot$. If $\mathit{ref}(t, \mathit{idx}) \simeq s$, then $t \simeq \mathit{replace}(t, \mathit{idx}, s)$.

Proof. By assumption, exists π where $s = \mathit{ref}(t, \mathit{idx})^\pi$. Let $\mathit{idx} = [b_1, \ldots, b_k]$. Define extended permutation ϕ as:

$$\phi([a_1, \ldots, a_n]) = \begin{cases} \mathit{concat}([a_1, \ldots, a_k], \pi([a_{k+1}, \ldots, a_n])) & \text{if } n \geq k \wedge \forall i \in \{1, \ldots, k\}.\, a_i = b_i \\ [a_1, \ldots, a_n] & \text{otherwise} \end{cases}$$

We can check that $\mathit{replace}(t, \mathit{idx}, s)^\phi = t$. □

Theorem 5.24 (Soundness of Best Implements)**.** Let $\mathcal{A}$ be a set of axioms, $\langle S, \mathit{src}, \mathit{tgt}\rangle$ a specification where S uses only rules in $\mathcal{A}$, and R be a program in the algebra DSL that best implements S for $\langle \mathit{src}, \mathit{tgt}\rangle$ under the identity binding. Then R is sound with respect to $\mathcal{A}$.

Proof. We show that the required condition for soundness holds by construction. Let s, t, $\langle \gamma, \pi\rangle$ be arbitrary such that $[\![R]\!](s, \langle \gamma, \pi\rangle) = t \neq \bot$. First, by Lemma 5.31, we have that

$t = \mathit{replace}(s, \gamma, [\![R]\!](\mathit{ref}(s, \gamma), \langle [], \pi \rangle))$. Then, by Lemma 5.32 and by Lemma 5.33, it follows that $t \simeq \mathit{replace}(s, \gamma, [\![R]\!](\mathit{ref}(s, \gamma)^{\pi}, I_{\beta}))$.

Let $S = [\langle R_1, \langle \gamma_1, \rho_1 \rangle \rangle, \ldots, \langle R_k, \langle \gamma_k, \rho_k \rangle \rangle]$ be the given execution plan. Define $\tau_0 = \mathit{ref}(s, \gamma)^{\pi}$ and $\tau_k = [\![R]\!](\tau_0, I_{\beta})$. By assumption that R best implements S under the identity binding, there exist terms $\tau_1, \ldots, \tau_{k-1}$ and permutations $\pi_1, \ldots, \pi_k$ such that $\forall i \in \{1, \ldots, k\}. \, \tau_i = [\![R_i]\!](\tau_{i-1}, \langle \gamma_i, \pi_i \rangle)$.

Let $t_i = \mathit{replace}(s, \gamma, \tau_i)$ for $i \in \{0, \ldots, k\}$. By construction, for all $i \in \{0, \ldots, k\}$, $\tau_i = \mathit{ref}(t_i, \gamma)$, and for all $i \in \{1, \ldots, k\}$, $t_i = \mathit{replace}(t_{i-1}, \gamma, \tau_i)$. Thus by Lemma 5.31, it follows that $\forall i \in \{1, \ldots, k\}. \, \mathit{replace}(t_{i-1}, \gamma, [\![R_i]\!](\mathit{ref}(t_{i-1}, \gamma), \langle \gamma_i, \pi_i \rangle)) = [\![R_i]\!](t_{i-1}, \langle \mathit{concat}(\gamma, \gamma_i), \pi_i \rangle)$, or, equivalently, $\forall i \in \{1, \ldots, k\}. \, t_i = [\![R_i]\!](t_{i-1}, \langle \mathit{concat}(\gamma, \gamma_i), \pi_i \rangle)$. Therefore, we have established an execution plan $S' = [\langle R_1, \langle \mathit{concat}(\gamma, \gamma_i), \pi_i \rangle \rangle, \ldots, \langle R_k, \langle \mathit{concat}(\gamma, \gamma_k), \pi_k \rangle \rangle]$ such that $[\![S']\!] t_0 = t_k$. Thus, for all $i \in \{1, \ldots, k\}$, $t_i \in \mathit{fire}(R_i, t_{i-1})$.

Lastly, we know by Lemma 5.33 that $t_0 \simeq s$, and we further know $t_k = \mathit{replace}(s, \gamma, \tau_k) = \mathit{replace}(s, \gamma, [\![R]\!](\mathit{ref}(s, \gamma)^{\pi})) \simeq t$. Thus, by Definition 5.10, R is sound with respect to $\mathcal{A}$. ☐

Theorem 5.26 (Partial Ordering of Pattern Refinement)**.** Let p_1, p_2 be patterns, c be a condition, a be an action, and R_1, R_2 be the well-formed programs $R_1 = \texttt{with } p_1\texttt{: if } c\texttt{: then } a$ and $R_2 = \texttt{with } p_2\texttt{: if } c\texttt{: then } a$. If p_1 refines p_2 ($p_1 \sqsubseteq p_2$), then R_2 is as general as R_1 and has identical behavior for applicable inputs to R_1. That is, $\forall t, \beta. \, [\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$.

Proof. The proof follows from the semantics of patterns, inducting on the cases in Definition 5.25. For any term t and binding β, we have that both $\mathcal{M}[\![p_2]\!](t, \beta) \implies \mathcal{M}[\![p_1]\!](t, \beta)$ and $\mathcal{B}[\![p_2]\!][] \subseteq \mathcal{B}[\![p_1]\!][]$. From this and the assumptions that R_1 and R_2 are well-formed and share a condition and action, it follows that $[\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$. ☐

Lemma 5.34. Let p be a pattern, S be a transition space, and s, t be terms such that $\mathcal{P}[\![p]\!] s \neq \bot$ and $t = [\![S]\!] s$. Also let $\texttt{with } p\texttt{: if } C\texttt{: do } A$ be a sketch (a program with holes to be filled by the synthesizer) with well-formed holes of size $\bar{k}$. If this sketch includes

a rule that best implements S for $\langle s, t\rangle$ under the identity binding, FINDRULE$(p, S, s, t, \bar{k})$ returns a singleton set containing such a rule; otherwise, it returns $\emptyset$.

Proof. The proof consists of two parts. First, we show that the synthesis query for the sketch $R_{CA} =$ `with` p`:` `if` C`:` `then` A can be decomposed into two queries for the sketches C and A. Then, we show that the CEGIS engine [133] invoked at lines 10–11 is sound and complete for these two queries. Hence, if R_{CA} contains a rule that best implements S for $\langle s, t\rangle$ under the identity binding, it will be found (due to completeness); otherwise, FINDRULE returns the empty set (due to soundness).

Decomposition. To find a rule in R_{CA} that satisfies Definition 5.22 for p, S, s, and t, we pose the following synthesis query:

$$\exists_{C,A}\, t = [\![R_{CA}]\!](s, I_\beta) \wedge \forall\tau.\mathcal{P}[\![p]\!](\tau, I_\beta) \neq \bot \implies [\![R_{CA}]\!](\tau, I_\beta) = [\![S]\!]\tau \tag{5.1}$$

By semantics of rule programs, $[\![R_{CA}]\!](\tau, I_\beta) = [\![S]\!]\tau$ is equivalent to

$$(\textbf{if } \sigma_\tau \neq \bot \wedge \mathcal{C}[\![C]\!]\sigma_\tau \textbf{ then } \mathcal{A}[\![A]\!](t, \sigma_\tau) \textbf{ else } \bot) = [\![S]\!]\tau \tag{5.2}$$

where $\sigma_\tau = \mathcal{P}[\![p]\!](\tau, I_\beta) = \mathit{bind}(\tau)$, (from line 6). Assuming that $\sigma_\tau \neq \bot$ (as it must in both places of Equation 5.1 where $[\![R_{CA}]\!](\tau, I_\beta) = [\![S]\!]\tau$ appears), Equation 5.2 expands into the following formulas:

$$(\mathcal{C}[\![C]\!]\sigma_\tau \implies [\![S]\!]\tau = \mathcal{A}[\![A]\!](\tau, \sigma_\tau)) \wedge (\neg\mathcal{C}[\![C]\!]\sigma_\tau \implies [\![S]\!]\tau = \bot) \tag{5.3}$$

$$(\mathcal{C}[\![C]\!]\sigma_\tau \implies ([\![S]\!]\tau \neq \bot \wedge [\![S]\!]\tau = \mathcal{A}[\![A]\!](\tau, \sigma_\tau))) \wedge ([\![S]\!]\tau \neq \bot \implies \mathcal{C}[\![C]\!]\sigma_\tau) \tag{5.4}$$

$$(\mathcal{C}[\![C]\!]\sigma_\tau \implies \mathcal{A}[\![A]\!](\tau, \sigma_\tau) = [\![S]\!]\tau) \wedge (\mathcal{C}[\![C]\!]\sigma_\tau \iff [\![S]\!]\tau \neq \bot) \tag{5.5}$$

$$([\![S]\!]\tau \neq \bot \implies \mathcal{A}[\![A]\!](\tau, \sigma_\tau) = [\![S]\!]\tau) \wedge (\mathcal{C}[\![C]\!]\sigma_\tau \iff [\![S]\!]\tau \neq \bot) \tag{5.6}$$

After substituting (5.6) into (5.1), rewriting $t = [\![R_{CA}]\!]s$ into $[\![C]\!]s \wedge t = [\![A]\!]s$, letting $\mathbb{T}$ stand

for $\{t \mid [\![p]\!]t\}$, and simplifying, we get

$$\begin{aligned}\exists_{C,A}\, \mathcal{C}[\![C]\!]\mathit{bind}(s) \wedge (\forall_{\tau\in\mathbb{T}}\, \mathcal{C}[\![C]\!]\mathit{bind}(\tau) \iff [\![S]\!]\tau \neq \bot) \wedge \\ \mathcal{A}[\![A]\!](s, \mathit{bind}(s)) = t \wedge (\forall_{\tau\in\mathbb{T}}\, [\![S]\!]\tau \neq \bot \implies \mathcal{A}[\![A]\!](\tau, \mathit{bind}(\tau)) = [\![S]\!]\tau)\end{aligned} \tag{5.7}$$

Separating Equation 5.7, we get the following separately solvable conjuncts:

$$\exists_{C}(\mathcal{C}[\![C]\!]\mathit{bind}(s) \wedge (\forall_{\tau\in\mathbb{T}}\, \mathcal{C}[\![C]\!]\mathit{bind}(\tau) \iff [\![S]\!]\tau \neq \bot)) \wedge \tag{5.8}$$

$$\exists_{A}(\mathcal{A}[\![A]\!](s, \mathit{bind}(s)) = t \wedge (\forall_{\tau\in\mathbb{T}}\, [\![S]\!]\tau \neq \bot \implies \mathcal{A}[\![A]\!](\tau, \mathit{bind}(\tau)) = [\![S]\!]\tau)) \tag{5.9}$$

These are the formulas solved at lines 10–11 of FindRule.

CEGIS. FindRule uses an off-the-shelf CEGIS engine [133] to solve the synthesis queries at lines 10–11. This engine is sound and complete for finite (loop-free) programs and inputs. Since rule programs are finite by definition, it remains to be shown that the input space $\mathbb{T}$ can be finitized without loss of soundness or completeness. Assuming that RuleSy treats integers as n-bit signed values, denoted by $\mathbb{Z}_n$, we show that $\mathbb{T}$ can be replaced with a finite set $\mathbb{T}_0 \subseteq \mathbb{T}$ in Equations 5.8–5.9 without affecting their satisfiability.[11]

Case 1. p includes no `_` or `etc` tokens. Let v be the number of `var` and `base` patterns in p, and V be a set of v fresh uninterpreted constants. Create a term t_p by and replacing each pattern in p with with a fresh symbolic name x_{idx}, where idx is the index of x_{idx} in t_p. Construct $\mathbb{T}_0$ by letting each x_{idx} in t_p range over $\mathbb{Z}_n$ if $\mathit{scope}(p, idx) =$ `num`, V if $\mathit{scope}(p, idx) =$ `var`, and $\mathbb{Z}_n \cup V$ if $\mathit{scope}(p, idx) =$ `base`. By construction, for each term $t \in \mathbb{T}$, there is a term $t' \in \mathbb{T}_0$ that is isomorphic to t up to a renaming of variables. Since rule programs can only refer to variables via references (not by name), replacing $\mathbb{T}$ with $\mathbb{T}_0$ is sound.

Case 2. p may include `_` but not `etc`. Let $c = (o\ i)$ be a complex term with an arbitrary

[11] This proof covers the constructors/values seen in the algebra domain: namely, variables and integer literals. The assumption (stated in Section 5.6) that the space of values for every constructor must be symbolically representable boils down to being able to modify this part of the proof appropriately. For example, the case for boolean literals from the semantic proof domain is trivial since the space is finite.

operator o from the domain and $i \in \mathbb{Z}_n$. Construct $\mathbb{T}_0$ as for Case 1, additionally letting x_{idx} in t_p range over $\mathbb{Z}_n \cup V \cup \{c\}$ if $scope(p, idx) = _$. This reduction is sound because a well-formed rule treats subterms matched by `_` as opaque: the rule's constraint can only compare them for (in)equality, and the action can only use them as atomic components for making new terms. In particular, **`apply`** can only be used with numeric literal terms.

Case 3. p may include both `_` and **`etc`**. Let k be the number of occurrences of **`etc`** in p. Create a set Q of 2^k patterns by either removing each **`etc`** from p or replacing it with `_`. Construct $\mathbb{T}_0$ for each $q \in Q$ as for Case 2, and take the union of the resulting sets to be $\mathbb{T}_0$ for p. This construction is sound because well-formed rules cannot reference any subterms matched by **`etc`**. As a result, for each occurrence of **`etc`**, $\mathbb{T}_0$ only needs to include enough terms to distinguish between **`etc`** matching no subterms and one subterm. □

Theorem 5.27 (Guarantees of Rule Synthesis)**.** Let S be an execution plan where $[\![S]\!]src = tgt$, and $\bar{k}$ a bound on the size of rule programs. FINDRULES($S, src, tgt, \bar{k}$) returns a set of rules $\mathcal{R}$ with the following properties: (1) every $R \in \mathcal{R}$ best implements S for $\langle src, tgt \rangle$; (2) $\mathcal{R}$ includes a sound rule R of size $\bar{k}$ if one exists; and (3) for every pattern p that refines or is refined by R's pattern, $\mathcal{R}$ includes a sound rule with pattern p and size $\bar{k}$ if one exists.

Proof. Line 2 defines p_0 to be the most refined pattern that matches s, so by Definition 5.25 and Theorem 5.26, p_0 refines the pattern of every rule applicable to s. Since there are finitely many such patterns (Definition 5.25), termination, soundness (1), and completeness (2–3) follow from line 3, Lemma 5.34, and Theorem 5.24. □

Theorem 5.28 (Guarantees of Model Optimization)**.** Let $\mathcal{T}$ be a set of tactics synthesized by RULESY for terms T and axioms $\mathcal{A}$, and let f be a total function from sets of rules and solution graphs to positive real numbers. OPTIMIZE($T, \mathcal{A}, \mathcal{T}, f$) returns a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that can solve each term in T, and for all such $\mathcal{R}' \subseteq \mathcal{A} \cup \mathcal{T}$, $f(\mathcal{R}, \{\text{SOLVE}(t, \mathcal{R}) \mid t \in T\}) \leq f(\mathcal{R}', \{\text{SOLVE}(t, \mathcal{R}') \mid t \in T\})$.

Proof. Let $\mathbb{R}$ be the set of all sets $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ such that SOLVE($t, \mathcal{R}$) terminates for

each $t \in T$. By construction of $\mathcal{T}$ (Theorems 5.20 and 5.27), we can show that for every $\mathcal{R} \in \mathbb{R}$, $\textsc{Solve}(t, \mathcal{R})$ produces a subgraph of the graph $\langle N, E_{\mathcal{A}} \cup E_{\mathcal{T}} \rangle$ defined at lines 4–5. Hence, for each non-$\textsc{Solved}$ term $t \in T$, $\mathcal{G}_{\mathcal{A} \cup \mathcal{T}}$ contains a solution graph G consisting of $\textsc{Solve}(t, \mathcal{R})$ for all $\mathcal{R} \in \mathbb{R}$. As a result, $\textsc{Restrict}(G, \mathcal{R})$ returns $\textsc{Solve}(t, \mathcal{R})$ if $\mathcal{R} \in \mathbb{R}$ and the empty graph otherwise. By line 7, $f_{\emptyset}(\mathcal{R}, \{\textsc{Restrict}(G, \mathcal{R}) \,|\, G \in \mathcal{G}_{\mathcal{A} \cup \mathcal{T}}\})$ is equal to $f(\mathcal{R}, \{\textsc{Solve}(t, \mathcal{R}) \,|\, t \in T\})$ for $\mathcal{R} \in \mathbb{R}$, and it is infinite otherwise. Consequently, the minimization operation at line 8 selects a cheapest set $\mathcal{R} \in \mathbb{R}$. □

5.11 Appendix: Axioms

In this section, we list all of the axioms used for both the algebra (Table 5.1b) and semantic proof (Table 4.1) domains in the evaluation of our system. Due to the formal nature of the problem, we need axioms for operations such as removing unary addition or flattening nested addition. Such axioms are critical for the system to work, but not very useful to teach explicitly.[12] Therefore, as an implementation detail, we distinguish these axioms used to clean up the representation from normal ones, called "hidden" and "normal" axioms respectively. Tactics are labeled with respect to only the normal axioms used within them. The referenced tables show only the normal axioms, while this appendix lists both the normal and hidden axioms.

5.11.1 Algebra Axioms

Normal Axioms

```
def add-identity:
    with (+ a@num _ etc):
    if a = 0:
    do remove a
```

[12] For example, algebra's standard infix-operator notation used by students can't even *represent* unary addition, so many of these "cleaning rules" are necessary as a consequence of modeling equations are trees.

```
def add-cfold:
    with (+ a@num b@num etc):
    if true:
    do replace a with apply(+ a b) and remove b

def mul-identity:
    with (* a@num _ etc):
    if a = 1:
    do remove a

def mul-zero:
    with a@(* b@num etc):
    if b = 0:
    do replace a with 0

def mul-cfold:
    with (* a@num b@num etc):
    if true:
    do replace a with apply(* a b) and remove b

def div-identity:
    with a@(/ b (* c@num)):
    if c = 1:
    do replace a with b

def div-cancel:
    with (/ (* a@base etc) (* b@base etc)):
    if a = b:
    do remove a and remove b
```

```
def frac-combine:
    with (* a (/ b@(* _ etc) _) etc):
    if true:
    do replace b with a :: b and remove a

def factor:
    with (+ (* a@base b) c@(* d@base e) etc):
    if a = d:
    do replace b with (+ b e) and remove c

def distribute:
    with (* a b@(+ c d) etc):
    if true:
    do remove a and
       replace c with (* a c) and
       replace d with (* a d)

def pos-1-coeff:
    with a@var:
    if true:
    do replace a with (* 1 a)

def neg-1-coeff:
    with a@(- b):
    if true:
    do replace a with (* -1 b)
```

```
def inverse-add:
    with (= a@(+ b etc) c):
    if true:
    do replace a with (+ (- b) a) and
       replace c with (+ (- b) c)

def inverse-mul:
    with (= a@(* b etc) c):
    if  b != 0:
    do replace a with (/ a b) and replace c with (/ c b)

def inverse-div:
    with (= a@(/ _ (* b etc)) c):
    if true:
    do replace a with (* b a) and replace c with (* b c)
```

Hidden Axioms

```
def clean-add-unary:
    with a@(+ b):
    if true:
    do replace a with b

def clean-add-flatten:
    with a@(+ b c@(+ _ etc)):
    if true:
    do replace a with b :: c
```

```
def clean-mul-unary:
    with a@(* b):
    if true:
    do replace a with b

def clean-mul-flatten:
    with a@(* b c@(* _ etc)):
    if true:
    do replace a with b :: c

def clean-exp-numerator:
    with (/ a _):
    if true:
    do replace a with (* a)

def clean-exp-denominator:
    with (/ _ a):
    if true:
    do replace a with (* a)

def clean-div-unary:
    with a@(/ b (* ) etc):
    if true:
    do replace a with b

def clean-neg-constant:
    with a@(- b@num):
    if true:
    do replace a with apply(- b)
```

5.11.2 Semantic Proof Axioms

```
def contradiction:
    with a@(facts (I⊨ b) (I⊭ c) etc):
    if b = c:
    do replace a with (I⊨ ⊥)::a

def branch-collapse:
    with a@(branch (facts (I⊨ b@base) etc) c):
    if b = ⊥:
    do replace a with c

def and-1:
    with a@(facts (I⊨ (∧ b _)) etc):
    if true:
    do replace a with (I⊨ b)::a

def and-2:
    with a@(facts (I⊭ (∧ b c)) etc):
    if true:
    do replace a with (branch (I⊭ b)::a (I⊭ c)::a)

def or-1:
    with a@(facts (I⊨ (∨ b c)) etc):
    if true:
    do replace a with (branch (I⊨ b)::a (I⊨ c)::a)
```

```
def or-2:
    with a@(facts (I⊭ (∨ b _)) etc):
    if true:
    do replace a with (I⊭ b)::a

def not-1:
    with a@(facts (I⊨ (¬ b)) etc):
    if true:
    do replace a with (I⊭ b)::a

def not-2:
    with a@(facts (I⊭ (¬ b)) etc):
    if true:
    do replace a with (I⊨ b)::a

def impl-1:
    with a@(facts (I⊭ (⇒ b _)) etc):
    if true:
    do replace a with (I⊨ b)::a

def impl-2:
    with a@(facts (I⊭ (⇒ _ b)) etc):
    if true:
    do replace a with (I⊭ b)::a

def impl-3:
    with a@(facts (I⊨ (⇒ b c)) etc):
    if true:
    do replace a with (branch (I⊭ b)::a (I⊨ c)::a)
```

Chapter 6

LEARNING DOMAIN MODELS FOR PUZZLE GAMES

The second, broadly different, domain for which we instantiated RuleSy is the family of logic puzzles *Nonograms*. This domain shares many common properies of other logic puzzles such as Sudoku, where the goals is to use deduction to fill in parts of a board until a solution state is reached, where the mechanics of the game specify what makes a solution valid. While the RuleSy framework is largely the same from the previous chapter, there are several important differences, some requiring substantial technical contributions. Most critically, we do not have a reasonable way to mine general specifications; our specifications will take the form of a single input/output example. Therefore, our synthesis algorithm needs to ensure it is learning not only sound and concise rules, but general rules that do not overfit to the training examples. As with the previous chapter, we expect portions of the contributions to apply to other domains (in this case, other logic puzzles). Portions of the chapter are reused from or based on prior technical publications [29].

6.1 Background

Automated game analysis is a growing research area that aims to uncover designer-relevant information about games without human testing [97, 119, 102, 132], which can be particularly advantageous in situations where human testing is too expensive or of limited effectiveness [139]. One potential use is automatically understanding game strategies: if we can analyze the mechanics of the game and automatically deduce what the effective player strategies are, we can support a range of intelligent tools for design feedback, content generation, or difficulty estimation. For humans to use these computer-generated strategies, they need the strategies to be both effective in the domain of interest and concisely expressed so a

designer can understand the whole strategy in their mind.

Modeling player interaction is challenging because the game mechanics do not fully capture how human players might approach the game. This is true for all games, but especially for logic puzzle games such as Sudoku or Nonograms. These puzzles are straightforward for a computer to solve mechanically by reduction to SAT or brute-force search, but humans solve them in very different ways. Rather than search, human players use a collection of interconnected strategies that allow them to make progress without guessing. For example, there are dozens of documented strategies for Sudoku[1] [126], and puzzle designers construct puzzles and rank their difficulty based on which of these strategies are used [127]. The strategies take the form of interpretable condition-action rules that specify (1) where a move can be made, (2) what (easy-to-check) conditions must hold to make it, and (3) how to make the move. Human players usually solve puzzles by looking for opportunities to apply these strategies rather than by manual deduction or search in the problem space.

Learning and applying these strategies is the core of human expertise in the game. Understanding these strategies as a designer allows one to effectively build and analyze puzzles and progressions. While many such strategies can be uncovered through user testing and designer introspection, they may not effectively cover the puzzle design space or be the most useful or simple strategies. Designers can benefit from tools that, given a game's mechanics, can help understand its strategy space. While we would prefer finding strategies people use, as a necessary step, we must find strategies we can easily understand and can demonstrate are effective for the problem-solving task.

In this chapter, we investigate automatically learning human-friendly game playing strategies expressed as condition-action rules. We focus on the popular puzzle game Nonograms, also known as Picross, Hanjie, O'Ekaki, or Paint-by-Numbers. A nonogram (see Figure 6.1) is a puzzle in which the player must fill in a grid of cells with either true (black square) or false. Integer *hints* are given for each row and column that specify how many contiguous seg-

[1] `http://www.sudokuwiki.org/Strategy_Families` contains a rather extensive list.

	1	1 1	2 1	1 1	1
0					
3					
1 1 1					
1 1					
1					

	1	1 1	2 1	1 1	1
0	✗	✗	✗	✗	✗
3	✗	■	■	■	✗
1 1 1	■	✗	■	✗	■
1 1	✗	■	✗	■	✗
1	✗	✗	■	✗	✗

Figure 6.1: An example of a Nonograms puzzle, with the start state on the left and completed puzzle on the right. The numbered *hints* describe how many contiguous blocks of *cells* are filled with **true**. We mark cells filled with **true** as a black square and cells filled with **false** as a red X. We use the X to distinguish from unknown cells, which are blank. We describe the mechanics formally in Section 6.3.

ments of filled cells exist in each row or column. Solutions often form interpretable pictures, though this is not necessary. By convention, nonograms have unique solutions and, like other logic puzzles such as Sudoku, can be solved by deducing parts of the answer in any order. Also like many logic puzzles, Nonograms is, for arbitrarily sized puzzles, NP-complete [134], but typical puzzles used in commercial books and games can often be solved with a fixed set of greedy strategies. Even for puzzles that require some tricky moves, greedy strategies can suffice to find a large portion of the solution. It is straightforward to construct a SAT-solver based algorithm to solve any Nonograms puzzle, but humans do not solve the puzzle this way. They usually use a large set of greedy strategies, and the research goal of this chapter is to automatically generate these strategies.

A key challenge in learning these strategies is *interpretability*: the learned strategies need to be expressed in terms of game-specific concepts meaningful to human players, such as, in the case of Nonograms, the numerical hints or state of the board. To address this challenge, we developed a new domain-specific programming language (DSL) for modeling

interpretable condition-action rules. In contrast to previous DSLs designed for modeling games [87, 102, 110, 101, 24], which focus on encoding the rules and representations of the game, our DSL focuses on capturing the strategies that a player can use when puzzle solving. Thus, the constructs of the language are game-specific notions, such as hint values and the current state of the board. In this way, we frame the problem of discovering player strategies for Nonograms as the problem of finding programs in our DSL that represent (logically) sound, interpretable condition-action rules. This soundness is critical and difficult to ensure: rules should be valid moves that respect the laws of the game, and complex constraints must hold for this to be the case. For this reason, we use a constraint solver at the core of our learning mechanism.

Learning condition-action rules for Nonograms involves solving three core technical problems: (1) automatically discovering specifications for potential strategies, (2) finding sound rules that implement those specifications, and (3) ensuring that the learned rules learned are general yet concise. The RuleSy framework for Nonograms takes as input the game mechanics, a set of small training puzzles, a DSL that expresses the concepts available for rules, and a cost function that measures rule conciseness. Given these inputs, it learns an optimal set of sound rules that generalize to large real-world puzzles. The system works in three steps. First, it automatically obtains potential specifications for rules by enumerating over all possible game states up to a small fixed size. Next, it uses an off-the-shelf program synthesis tool [133] powered by an SMT (Satisfiability Modulo Theories) solver to find programs that encode sound rules for these specifications. Finally, it reduces the resulting large set of rules to an optimal subset that strikes a balance between game-state coverage and conciseness according to the given cost function. We evaluate the system by comparing its output to documented strategies for Nonograms drawn from tutorials and guides, finding that it fully recovers many of these control rules and covers nearly all of game states covered by the remaining rules.

While our implementation and evaluation focuses on Nonograms, the system makes relatively weak assumptions (discussed in detail) about the DSL used as input. Variations could

be used, and we expect DSLs representing other logic puzzles could be used in the system. And while many parts of the learning mechanism are specific to logic puzzles, we expect the approach of using program synthesis for learning of human-interpretable strategies to apply more broadly, especially to domains with well-structured problem solving.

Differences from the Algebra Domain

This domain differs from algebra and the other tree-rewrite rule domains in fundamental ways that contribute to these challenges. For example, while we can construct the oracle for sound rules (by reduction to SAT), we cannot easily create general specifications. Thus, it is easy to get individual input/output examples of sound behavior but not to get a function that defines a large class of sound behaviors. Our synthesis algorithm will therefore need to ensure that the rules being synthesized are general while at the same time being concise and sound.

Another way in which this domain differs substantially from tree-rewrite rules in that it is not at all obvious what patterns in the DSL should look like. For trees, it was fairly natural to have a language that pattern-matched on particular shapes of trees. All patterns were based on these "primitive" elements of the state. The equivalent such elements in Nonogarms are the values of the hints and values of the cells. However, when looking at examples of rules that humans use (examples are which are listed in Section 6.11), they are defined in terms of higher-level concepts like "blocks of cells" or "the edge of the board." In order for rules to be concise and for synthesis to be tractable, the DSL's patterns should allow for such high-level concepts to be directly expressed as primitive constructs. But this immediately creates a challenging design problem of which patterns are worth adding. In this chapter, we present a set chosen after much iterative design that produces good results by being expressive enough to admit useful rules while restricted enough for synthesis and verification to be tractable. In the following Chapter 7, we discuss in detail this design problem and how we might go about making it easier for future DSL designers to tackle.

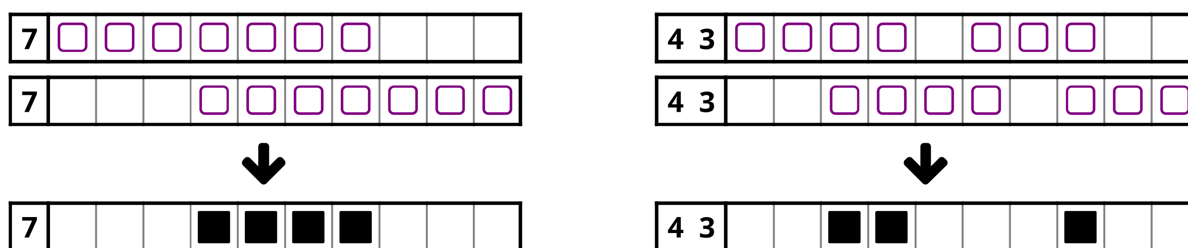
6.2 Overview

This section provides a high-level overview of RuleSy for Nonograms, where we show an example of a greedy strategy that our system can learn. Of course, one can always solve a puzzle by some combination of brute-force search and manual deduction, but human players prefer to use a collection of greedy strategies. Puzzles are designed with these strategies in mind, which take the form of pattern-condition-action rules (as in the algebra domain). This section illustrates the key steps that our system takes to synthesize such condition-action rules.

6.2.1 Pattern-Condition-Action Rules for Nonograms

Nonograms puzzles can be solved in any order: as cells are deduced and filled in, monotonic progress is made towards the final solution. In principle, deductions could be made using information from the entire board. In practice, people focus on some substate of the board. One natural class of substates are the individual rows and columns of the board, which we call *lines*. Since the rules of the game are defined with respect to individual lines, they can be considered in isolation. A solving procedure that uses only lines is to loop over all lines of the board, applying deductions to each. This will reveal more information, allowing more deductions to be applied to crossing lines, until the board is filled. As many puzzle books and games can be completed by only considering one line at a time, that is the scope on which we focus for this paper. That is, our rule programs will map lines to lines rather than full boards to boards.

We define the mechanics formally in Section 6.3, but to summarize here: the numerical hints of a given line describe how many and how long contiguous blocks of true-filled cells exist in a line. For example, a line with the hint sequence $[4, 3]$ means that, when solved, the line must have a total of seven true-filled cells. Four of them must be adjacent to each other, with a gap of of at least false cell to the right, followed by the other three, also adjacent to each other.



(a) An example of the *big hint* rule: for any line with a single, sufficiently large hint, no matter how the hint is placed, some cells in the center will be filled.

(b) An example of the *big hint* rule for multiple hints.

Figure 6.2: The *big hint* rule for one (a) and many (b) hints. This is an example of the kind of sound greedy strategy for which we aim to learn interpretable descriptions.

As an example of a greedy condition-action rule for Nonograms, we consider what we call the *big hint* rule (Figure 6.2), a documented strategy for Nonograms.[2] If a hint value is sufficiently large relative to the size of the line (Figure 6.2a), then, without any further information, we can fill in a portion of the middle of the row. The big hint rule can be generalized to multiple hints (Figure 6.2b): if there is any overlap between the furthest left and furthest right possible positions of a given hint in the row, we can fill in that overlap. Our system aims to discover sound strategies of this kind, and synthesize human-readable explanations of them that are (1) general, so they apply to a wide variety of puzzle states, and (2) concise, so they have as simple an explanation as possible.

6.2.2 System Overview

To synthesize sound, general, and concise descriptions of Nonograms strategies, we use the same broad framework as the algebra domain, illustrated by Figure 4.10. Our system needs the following inputs:

1. The formal mechanics of Nonograms to determine the soundness of learned strategies.

[2] `https://en.wikipedia.org/wiki/Nonogram#Simple_boxes`

2. A domain-specific language (DSL) defining the concepts and features to represent these strategies.

3. A cost function for rules to measure their conciseness.

4. A training set of line states from which to learn rules.

5. A testing set of line states with which to select an optimal subset of rules (that maximize state coverage).

Given these inputs, the system uses the 3-phase algorithmic pipeline to produce an optimal set of rules represented in the DSL: *specification mining*, *rule synthesis*, and *rule set optimization*. We explain each of these phases by illustrating their operation on toy inputs.

Specification Mining

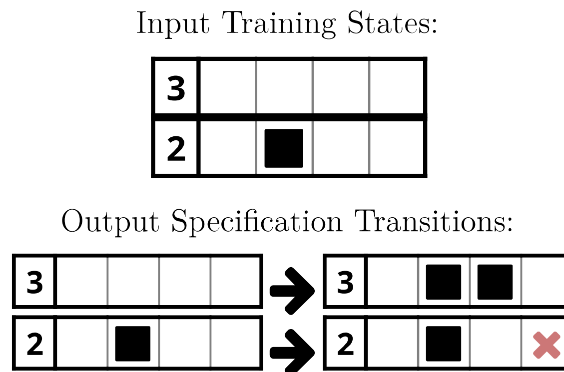


Figure 6.3: Example of the inputs and outputs for specification mining in our toy problem. Given a set of states, we use the rules of Nonograms to calculate deducible transitions for those states. Each transition serves as the specification for a potential rule.

Before synthesizing interpretable strategies, we need specifications of their input/output behavior. Our system mines these specifications from the given training states as illustrated in Figure 6.3. For each training state, we use the rules of Nonograms (and an SMT solver) to calculate all cells that can be filled in, producing a maximally filled target state. The resulting pair of line states—the training state and its filled target state—forms a sound *transition* in

the state of the game. In our system, an individual transition forms the specification for a rule. A single transition is an underspecification of a strategy since many rules may cover that particular transition. We leave it to the rule synthesis phase to find the most general and concise rule for each mined specification.

Rule Synthesis

The rule synthesis phase takes as input a mined transition, the DSL for rules, and the cost function measuring rule complexity. Given these inputs, it uses standard synthesis techniques to find a program in the DSL that both covers the mined transition and is sound with respect to the rules of Nonograms. Figure 6.4 shows the output of the synthesis phase for the first transition in our toy example (Figure 6.3).

```
def big_hint_rule:
    with h = singleton(hint):
    if lowest_end_cell(h) > highest_start_cell(h):
    do fill(true, highest_start_cell(h), lowest_end_cell(h))
```

Figure 6.4: Basic version of the big hint rule. These programs are the output of the rule synthesis phase of the system. The `with`, `if`, and `then` delineate the 3 parts of a rule: the *pattern*, *condition*, and *action*. These are explained in Section 6.4.

The key technical challenge that this phase must solve, beyond finding sound rules, is to ensure the rules are general and concise. Generality is measured by the number of line states to which the rule is applicable, and conciseness is measured by the cost function provided by the designer. We use iterative optimization to maximize each of these. We additionally exploit the structure of the DSL for generality, which we detail in Section 6.6.

Rule Set Optimization

The synthesis phase produces a set of programs in the DSL, one for each mined transition, that represent the interpretable strategies we seek. Because the DSL captures human-relevant

features of Nonograms, the concise programs are human-readable descriptions of the strategies. However, this set of rules can be unmanageably large, so the rule optimization phase prunes it to a subset of the most effective rules. In particular, given a set of rules and a set of testing states on which to measure their quality, this phase selects a subset of the given rules that best covers the states in the testing set. In our implementation, testing states are drawn from solution traces to real-world puzzles. Thus, in our toy example, the big hint rule will be selected for the optimal subset because it is widely applicable in real-world puzzles.

6.3 The Mechanics of Nonograms

In order to simplify the presentations of the DSL and the RuleSy framework, we first separately define the state representation and mechanics of Nonograms formally. For concreteness, the DSL and RuleSy algorithms will be presented in detail on Nonograms. However, both the language and algorithms are defined more abstractly, and could in theory be instantiated for other logic puzzles. We shall point out these differences between Nonograms-specific details versus generic implementation where appropriate.

Problem states are nonograms boards (Definition 6.2), represented as a two-dimensional grid of cells and a set of numerical hints. Cells can take on one of three values: **true**, **false**, or **empty**. Initial nonograms boards start with all cells **empty**, and a solved board has all cells filled with **true** or **false** in accordance with the mechanics.[3] These mechanics are defined with respect to an individual row or column of the board, which we call a *line* (Definition 6.1). We call lines (and boards) *solved* when filled out appropriately in accordance with the mechanics, and *valid* if there is some solution consistent with their current (usually only partially filled) state (Definitions 6.6 and 6.8). Hints apply to a single line, and the validity of a line is self-contained: the mechanics specify what counts as a solved line, and a board is solved if and only if all lines are solved. Most sources of commercial puzzles further

---

[3]There are variants of Nonograms that use other cell values, such as various colors to allow for more complex pictures. We restrict ourselves to the basic mechanics, though much of the framework could be straightforwardly adapted to the variants.

restrict boards to be valid only if there is a unique solution. The relationship between hints and solved lines is formalized with the *line segment* (Definition 6.3), specifically the *true-valued line segment* (Definition 6.4).

Definition 6.1 (Line State)**.** A Nonograms *line state* (or just *line*) s is a tuple $\langle H, V \rangle$, where $H = [h_1, \ldots, h_k]$, $\forall_{i \in \{1,\ldots,k\}} h_i \in \mathbb{N}^+$ is an ordered sequence of hints, and $V = [v_1, \ldots, v_n]$, $\forall_{i \in \{1,\ldots,n\}} v_i \in \{\mathbf{empty}, \mathbf{true}, \mathbf{false}\}$ is an ordered sequence of cells. Hints are known positive integers. Cells can have one of three values: either *unknown* (**empty**) or *filled* with either **true** or **false**. We say $\mathit{hints}(s)$ to mean H and $\mathit{cells}(s)$ to mean V. We use the length of s as shorthand for the length of $\mathit{cells}(s)$.

While the hints are included in the line state for convenience, they cannot be changed by the player; they are not "state" in traditional sense of being mutable.

Definition 6.2 (Board State)**.** A Nonograms *board state* (or just *board*) S is a tuple $\langle \mathcal{R}, C \rangle$ where $\mathcal{R} = [R_1, \ldots, R_M]$ is an ordered sequence of lines of equal size N (the rows and their hints), an ordered sequence $[h_1, \ldots, h_N]$ of sequences of positive integer hints (the column hints). We say S has width N and height M. The *rows* of S (written $\mathit{rows}(S)$) is the sequence $\mathcal{R}$. The *columns* of S (written $\mathit{columns}(S)$) is the sequence of size-M lines $[\langle h_1, V_1 \rangle, \ldots, \langle h_N, V_N \rangle]$ where $V_i = [\mathit{cells}(R_1)_i, \ldots, \mathit{cells}(R_M)_i]$.

Definition 6.3 (Line Segment)**.** A *line segment* (or just *segment*) is a pair of integers $\gamma = \langle b, e \rangle$ where $0 \leq b < e$, representing starting and ending indices for a subsequence of a line. The value b is the 0-indexed, inclusive start index and e the 0-indexed exclusive end index.

For a given line $s = \langle [h_1, \ldots, h_k], [v_1, \ldots, v_n] \rangle$, the segment γ is *valid* for s iff $e \leq n$. We write $s[\gamma] = [v_{b+1}, \ldots, v_e]$ to denote the subsequence of $\mathit{cells}(s)$ specified by γ.

We say b is the *start* of the segment (written $\mathit{start}(\gamma)$), and e is the *end* of the segment (written $\mathit{end}(\gamma)$). We define the *size* of the segment as $\mathit{size}(\gamma) = e - b$.

Definition 6.4 (Valued Line Segment)**.** Given a line s, a segment γ, and a set $\emptyset \subset K \subset \{\mathbf{empty}, \mathbf{true}, \mathbf{false}\}$, we say that γ is a *K-valued segment for s* iff (1) all cells in $s[\gamma]$ are in K and (2) the cells immediately adjacent to the segment are not in K (meaning each end

of the segment either abuts the edge of the line or is next to a cell of a value other than K). That is, $\gamma = \langle b, e\rangle$ is K-valued for $s = \langle [h_1, \dots, h_k], [v_1, \dots, v_n]\rangle$ if:

$$\forall_{i\in\{b,\dots,e-1\}} v_i \in K \wedge (b = 0 \vee v_{b-1} \notin K) \wedge (e = n \vee v_e \notin K)$$

For example, a true-valued line segment contains only **true** cells, and a true/empty-valued line segment contains cells that are either **true** or **empty**.

We write *segments*(s, K) to mean the sequence $[\gamma_1, \dots, \gamma_m]$ of all valid segments of s that are K-valued, ordered by *start*(γ_i). Note that, as a consequence of this definition, these segments are non-overlapping and have at least a one-cell gap between them: that is, $\forall_{i\in\{1,\dots,m\}} \mathit{end}(\gamma_i) < \mathit{start}(\gamma_{i+1})$.

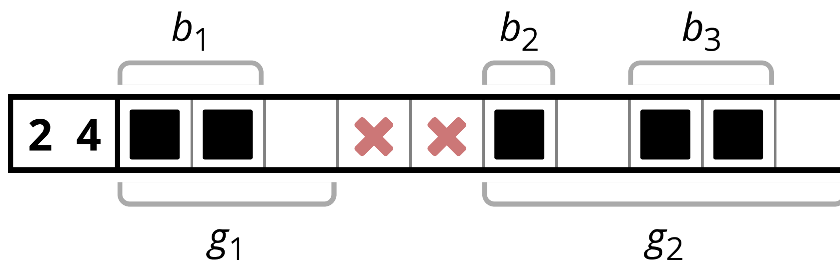


Figure 6.5: Examples of line segments. The top (b_1, b_2, b_3) are the true-valued line segments of this line, and the bottom (g_1, g_2) the true/empty-valued line segments.

Example 6.5 (Valued Line Segments)**.** We illustrate an example of line segments in Figure 6.5. This line has 3 true-valued segments: *segments*$(s, \{\mathbf{true}\}) = [b_1 = \langle 0, 2\rangle, b_2 = \langle 5, 6\rangle, b_3 = \langle 7, 9\rangle]$, and 2 true/empty-valued segments: *segments*$(s, \{\mathbf{empty}, \mathbf{true}\}) = [g_1 = \langle 0, 3\rangle, g_2 = \langle 5, 10\rangle]$. There are 6 possible combinations of valued segments, but true and true/empty are the most interesting and the ones we use as patterns in our DSL.[4] True/empty valued segments are a common concept used in rules since they represent potential places a hint's corresponding segment may be. True-valued segments are important since the me-

[4]More of them are explored in the following chapter on designing the Nonograms DSL.

chanics of Nonograms are defined with them, as we see below.

Definition 6.6 (Solved Lines)**.** A line $s = \langle[h_1, \dots, h_k], [v_1, \dots, v_n]\rangle$ is *solved* iff both (1) all cells in s are filled and (2) $\mathit{segments}(s, \{\mathbf{true}\}) = [\gamma_1, \dots, \gamma_k]$ with $\forall_{i \in \{1,\dots,k\}} \mathit{size}(\gamma_i) = h_i$.

Definition 6.7 (Valid Lines)**.** A line state is *valid* if there is at least one assignment of all unknown cells to **true** or **false** that produce a solved line. That is, there is at least one solution to the line state. Unless otherwise noted, we are implicitly talking about valid line states.

Definition 6.8 (Solved Boards)**.** A board S is *solved* if all lines in $\mathit{rows}(S) \cup \mathit{columns}(S)$ are solved.

Definition 6.9 (Valid Boards)**.** A board is *valid* if there is a unique assignment to all unknown cells in $\mathcal{R}$ that produce a solved board.[5] A Nonograms puzzle (also called a nonogram) is a valid board where all cells in $\mathit{rows}(S) \cup \mathit{columns}(S)$ are unknown.

For the remainder of this chapter, we will be focusing on line states rather than board states. Thus, unless otherwise noted, when we say *state* we mean a line state.

6.4 A Domain-Specific Language for Nonograms Rules

This section presents the RuleSy domain-specific language (DSL) for rules in the nonograms domain. As with algebra, the language is parametric in its definition of problem states, but we present a concrete instantiation of RuleSy that we used in our evaluation.

6.4.1 Specifying Problems and Rules.

As discussed in the previous section, problem states are Nonograms boards (Definition 6.2), but rules are applied to a sub-state of the board, namely lines (Definition 6.1). In principle, the sub-states we choose for rule application could be anything (e.g., multiple lines, the entire

[5] This uniqueness assumption isn't strictly necessary but used by the vast majority of nonograms. With it, a deduction can always be made from any state that fills in only a single cell. RuleSy doesn't care either way: it only deals with specifications for sound deductions. However, even assuming RuleSy learned all theoretically possible rules, those rules may not completely solve puzzles with non-unique solutions.

board). In this implementation we focus on the line: lines are a natural sub-state of boards since the mechanics of Nonograms (Definition 6.6) are defined with respect to individual lines. Moreover, a large portion of puzzles (including all puzzles in the references used for evaluation) can be solved entirely through rules applied to individual lines.[6]

Rules in the Nonograms domain map lines to lines by filling in zero or more cells of the input line. Patterns describe various features of the state that can be matched. Conditions describe additional constraints of the values matched by the pattern. Actions describe filling in a particular set of cells with either **true** or **false**.

6.4.2 Patterns and Bindings in Nonograms Rules

As with algebra, we must consider how rules can apply in multiple ways to the same input state. For example, the big hint rule (Figure 6.2) applies to an arbitrary hint, and potentially multiple hints within the same line. We handle this in a similar way to the algebra domain, using the concept of a *binding* (Definition 6.10). Thus programs will be (partial) functions from lines and bindings to lines. The nature of bindings in this domain is closely tied to the types of patterns available, so we treat them together.

Patterns, loosely speaking, describe various features of the line state. These include the obvious primitive features of the state such as the hints or the cell values, but also include higher-order features such as a valued line segments (Definition 6.4). Figure 6.9 shows the definitions for the four pattern types we chose to use in the implementation evaluated in this chapter: hints, cells, blocks (true-valued segments), and gaps (true/empty-valued segments). RuleSy is designed to allow straightforward addition of new pattern types, so any range of pattern types could be chosen. Particularly for this domain, the problem of which patterns to include is a challenging and subjective design task. We explore this DSL design task

---

[6] Some advanced puzzles do require reasoning with more complex sub-states, such as multiple lines, lines on the edge of a board, or crossing lines. RuleSy could, in principle, consider all such sub-states and find rules for each in parallel. However, since the state representation is different, the DSL's patterns and actions would have to change to support each new sub-state. In essence, RuleSy would have to solve unrelated problems in parallel. For simplicity of presentation, we keep to one state representation.

explicitly in the following Chapter 7.

The common feature of our 4 pattern types is that they represent sequences: the hints are a sequence of integers, the cells are a sequence of true/false/empty cells, and the valued line segments are a sequence of segments. Our pattern language matches (potentially non-deterministically) to a particular element of one of these sequences. Our DSL supports three matching constructs: matching the element of a **singleton** sequence, matching an element of a sequence at a **constant** index, or matching an **arbitrary** element of a non-empty sequence. The meaning of these is formalized in Figure 6.10.

Bindings specify which elements of the state are bound by the various parts of the pattern. We represent this using the 0-indexed location in the corresponding element sequence. Programs in our DSL can contain multiple patterns on which to match, so therefore a binding in our DSL is a sequence of integers corresponding to the matched index for each pattern. Note that both **singleton** and **constant** are deterministic: only **arbitrary** can support matching multiple bindings on the same line.

Definition 6.10 (Bindings). Let $\beta = [\iota_1, \ldots, \iota_k]$ be a sequence of non-negative integers. We say that β is a *binding* for a pattern $p = p_1$ **and** ... **and** p_k if (1) for all i where $p_i =$ **singleton**(T), $\iota_i = 0$, and (2) for all i where $p_i =$ **constant**(T, j), $\iota_i = j$. Any binding is valid for any line state.

To draw a parallelism with the algebra domain, this pattern language shares a critical property with the algebra DSL: the patterns form a semilattice of generality, which we will formalize with the definition of *pattern refinement* in Section 6.6.2. That is, any state/binding matching a **singleton**(T) pattern will also match a **constant**(T,0) pattern, and any state/binding matching a **constant**(T,k) pattern will also match a **arbitrary**(T) pattern. As will become apparent in Section 6.6.2, exploiting this property is critical for efficient synthesis in both the algebra and the Nonograms domains. Figure 6.6 shows an example of exploiting this structure to generalize the big hint rule from Figure 6.4; the new rule uses a more general pattern and thus is applicable to a strict superset of states.

```
def big_hint_rule_general:
    with h = arbitrary(hint):
    if lowest_end_cell(h) > highest_start_cell(h):
    do fill(true, highest_start_cell(h), lowest_end_cell(h))
```

Figure 6.6: General version of the big hint rule, which uses an **arbitrary** pattern instead of a **singleton** pattern. This rule applies in strictly more situations than the one in Figure 6.4 and is therefore more general.

6.4.3 Semantics of Patterns, Conditions, and Actions

Rule programs (Figure 6.7), denoting partial functions from lines and bindings to lines, have semantics that instantiate of the general RuleSy semantics (Figure 4.8). In this section we give domain-specific meaning to contexts, patterns, conditions, and actions. Figure 6.8 shows the types for the various semantic functions and the definition of those functions is shown in Figures 6.10 and 6.9. A complete example of the semantics of rule application is shown in Example 6.12.

Patterns

The semantic functions have a parallel meaning to the ones from the algebra DSL. The function $\mathcal{P}$ gives meaning to patterns as partial functions from states and bindings to contexts. For Nonograms, patterns match on lists of state features, e.g., a pattern might match on an arbitrary hint. Other types of patterns include matching on individual cells or sections of same-valued cells. For a given type, patterns can match using one of the **arbitrary**, **constant**, or **singleton** constructs. These correspond to matching to an arbitrary element, an element at a fixed index, and the only element of a singleton list, respectively. For Nonograms, a context maps a variable x of the pattern to a tuple $\langle e, L\rangle$, where e is the matched element (e.g., some hint, some cell) and L is the entire sequence of elements of the matched type (e.g., the sequence of all hints). The value L is required for constructs such as **maximal**. The values in these sequences are specific to the pattern type, defined by the *elements* function and given meaning by the *method* function.

rule	R	::=	**with** P: **if** e: **do** A
patterns	P	::=	d {**and** d}*
pattern declaration	d	::=	x = p
pattern expression	p	::=	**singleton**(t)
		|	**constant**(t, k)
		|	**arbitrary**(t)
pattern type	t	::=	***hint*** | ***cell*** | ***block*** | ***gap***
action	A	::=	**fill**(B, e, e)
expression	e	::=	(e)
		|	**true**
		|	e **and** e
		|	e o e
		|	x **is unique where** b
		|	**maximal**(M,x)
		|	**minimal**(M,x)
		|	M(x)
operator	o	::=	+ | - | = | >= | >
method	M	::=	*value*
		|	*index*
		|	*line_size*
		|	*lowest_end_cell*
		|	*highest_start_cell*
		|	**true**?
		|	**false**?
		|	empty?
identifier	x	::=	identifier
integer	k	::=	integer literal
boolean	B	::=	**true** | **false**

Figure 6.7: Syntax for the Nonograms DSL. The notation {*form*}* means zero or more repetitions of the given form.

The semantics of $\mathcal{P}$ are defined by the semantic functions $\mathcal{M}$ and $\mathcal{B}$, which has meaning similar to algebra. The function $\mathcal{M}$ (for "match") gives meaning to the pattern, checking that a subpattern matches the state. The function $\mathcal{B}$ (for "bind") gives meaning to the variables in the pattern, extracting the sequences of pattern types.

$$
\begin{aligned}
\text{binding} &\coloneqq \mathbb{Z}^{+} \text{ sequence} \\
\text{context} &\coloneqq \text{identifier} \to \text{pattern element} \times (\text{pattern element sequence}) \\
\mathcal{P} &: \text{pattern} \times \text{state} \times \text{binding} \to (\text{context} \cup \{\bot\}) \\
\mathcal{C} &: \text{condition} \times \text{context} \to \text{value} \\
\mathcal{A} &: \text{action} \times \text{state} \times \text{context} \to \text{state} \\
\mathcal{M} &: \text{pattern expression} \times \text{state} \times \mathbb{Z}^{+} \to \text{boolean} \\
\mathcal{B} &: \text{pattern expression} \times \text{state} \to \text{pattern element sequence} \\
\mathit{elements} &: \text{pattern type} \times \text{state} \to \text{pattern element sequence} \\
\mathit{method} &: \text{method} \times \text{pattern element} \to \text{value}
\end{aligned}
$$

Figure 6.8: Types for the Nonograms DSL's inputs/outputs and semantic functions. A binding β is a sequence of non-negative integers (see Definition 6.10). Pattern elements are values specific to each pattern type (e.g., ***hint***), given meaning by the *elements* function (Figure 6.9). Values are either booleans or integers; the DSL has a simple type system that is omitted for presentation clarity.

Conditions

The function $\mathcal{C}$ gives meaning to conditions and subexpressions of conditions and actions as a function from contexts for (type-appropriate) values. Expressions include boolean and integer arithmetic as well as pattern-specific expressions. For well-formed programs, the top-level expression in a condition evaluates to a boolean.

Semantically equivalent conditions are sometimes expressible in qualitatively distinct ways. Our system uses a designer-provided cost function to choose the best one. For example, Figure 6.11 shows another program for the basic big hint rule that uses an arithmetic condition rather than the geometric one used in Figure 6.4. Which rule is selected by our system depends on whether the cost function assigns lower values to geometric or arithmetic operations—a decision left to the designers using the system, enabling them to explore the

$$
\begin{aligned}
elements[\![\texttt{cell}]\!]\langle H, V\rangle &= (\langle i, V_i\rangle)_{i=1}^{|V|} \\
method[\![\texttt{index}]\!]\langle idx, v\rangle &= idx \\
method[\![\texttt{true?}]\!]\langle idx, v\rangle &= v = \textbf{true} \\
method[\![\texttt{false?}]\!]\langle idx, v\rangle &= v = \textbf{false} \\
method[\![\texttt{empty?}]\!]\langle idx, v\rangle &= v = \textbf{empty} \\
\\
elements[\![\texttt{hint}]\!]\langle H, V\rangle &= (\langle H, V, i\rangle)_{i=1}^{|H|} \\
method[\![\texttt{value}]\!]\langle H, V, idx\rangle &= H_{idx} \\
method[\![\texttt{line_size}]\!]\langle H, V, idx\rangle &= |V| \\
method[\![\texttt{lowest_end_cell}]\!]\langle H, V, idx\rangle &= idx - 1 + \textstyle\sum_{j=1}^{idx} H_j \\
method[\![\texttt{highest_start_cell}]\!]\langle H, V, idx\rangle &= |V| - |H| + idx - \textstyle\sum_{j=idx}^{|H|} H_j \\
\\
elements[\![\texttt{block}]\!]s &= segments(s, \{\textbf{true}\}) \\
elements[\![\texttt{gap}]\!]s &= segments(s, \{\textbf{true}, \textbf{empty}\}) \\
method[\![\texttt{start}]\!]seg &= start(seg) \\
method[\![\texttt{end}]\!]seg &= end(seg) \\
method[\![\texttt{size}]\!]seg &= size(seg)
\end{aligned}
$$

Figure 6.9: Semantics for patterns in the Nonograms DSL, grouped by pattern type. New patterns can be added by adding further cases to these semantic functions, which Chapter 7 explores in detail. Some notation is described in Section 4.2. For example, the first group describes the `cell` pattern type. Cell elements are represented as pairs of an index and a value. The *elements* semantic function for cells maps a line state to the sequence of these pairs. Methods on a particular cell element include one that evaluates to the cell's index, and three predicates that evaluate depending on whether the cell's value is **true**, **false**, or **empty**.

$$
\begin{aligned}
&[\![\texttt{with } p\texttt{: if } c\texttt{: do } a]\!](s,\beta) &&= \textbf{if } \sigma \neq \bot \wedge \mathcal{C}[\![c]\!]\sigma \textbf{ then } \mathcal{A}[\![a]\!](s,\sigma) \textbf{ else } \bot \\
& && \quad \textbf{where } \sigma = \mathcal{P}[\![p]\!](s,\beta) \\
&\mathcal{P}[\![x_1 = p_1 \texttt{ and } \ldots\ x_n = p_n]\!](s,\beta) &&= \textbf{if } \beta = [\iota_1, \ldots, \iota_n] \wedge \forall_{i \in \{1,\ldots,n\}} \mathcal{M}[\![p_i]\!](s,\iota_i) \\
& && \quad \textbf{then } \{x_i \mapsto \langle L_{\iota[i]+1}, L\rangle \mid L = \mathcal{B}[\![p_i]\!]s,\ i \in \{1,\ldots,n\}\} \\
& && \quad \textbf{else } \bot \\
&\mathcal{M}[\![\texttt{singleton(}T\texttt{)}]\!](s,\iota) &&= \iota = 0 \wedge |L| = 1 \ \ \textbf{where } L = \mathit{elements}[\![T]\!]s \\
&\mathcal{M}[\![\texttt{constant(}T\texttt{, }k\texttt{)}]\!](s,\iota) &&= \iota = k \wedge 0 \leq \iota < |L| \ \ \textbf{where } L = \mathit{elements}[\![T]\!]s \\
&\mathcal{M}[\![\texttt{arbitrary(}T\texttt{)}]\!](s,\iota) &&= 0 \leq \iota < |L| \ \ \textbf{where } L = \mathit{elements}[\![T]\!]s \\
&\mathcal{B}[\![\texttt{singleton(}T\texttt{)}]\!]s &&= \mathit{elements}[\![T]\!]s \\
&\mathcal{B}[\![\texttt{constant(}T\texttt{, }k\texttt{)}]\!]s &&= \mathit{elements}[\![T]\!]s \\
&\mathcal{B}[\![\texttt{arbitrary(}T\texttt{)}]\!]s &&= \mathit{elements}[\![T]\!]s \\
&\mathcal{C}[\![\texttt{true}]\!]\sigma &&= \text{true} \\
&\mathcal{C}[\![\texttt{false}]\!]\sigma &&= \text{false} \\
&\mathcal{C}[\![b_1 \texttt{ and } b_2]\!]\sigma &&= \mathcal{C}[\![b_1]\!]\sigma \wedge \mathcal{C}[\![b_2]\!]\sigma \\
&\mathcal{C}[\![r \texttt{ + } e]\!]\sigma &&= \mathcal{E}[\![r]\!]\sigma + \mathcal{E}[\![e]\!]\sigma \\
&\mathcal{C}[\![r \texttt{ - } e]\!]\sigma &&= \mathcal{E}[\![r]\!]\sigma - \mathcal{E}[\![e]\!]\sigma \\
&\mathcal{C}[\![r \texttt{ = } e]\!]\sigma &&= \mathcal{E}[\![r]\!]\sigma = \mathcal{E}[\![e]\!]\sigma \\
&\mathcal{C}[\![r \texttt{ > } e]\!]\sigma &&= \mathcal{E}[\![r]\!]\sigma > \mathcal{E}[\![e]\!]\sigma \\
&\mathcal{C}[\![r \texttt{ >= } e]\!]\sigma &&= \mathcal{E}[\![r]\!]\sigma \geq \mathcal{E}[\![e]\!]\sigma \\
&\mathcal{C}[\![\texttt{maximal(}m\texttt{, }x\texttt{)}]\!]\sigma &&= u = \max\{\mathit{method}[\![m]\!]v \mid v \in L\} \\
& && \quad \textbf{where } \langle u, L\rangle = \sigma[x] \\
&\mathcal{C}[\![\texttt{minimal(}m\texttt{, }x\texttt{)}]\!]\sigma &&= u = \min\{\mathit{method}[\![m]\!]v \mid v \in L\} \\
& && \quad \textbf{where } \langle u, L\rangle = \sigma[x] \\
&\mathcal{C}[\![x \texttt{ is unique where } b]\!]\sigma &&= \mathcal{C}[\![b]\!]\sigma \wedge \forall_{v \in L \setminus u} \neg\mathcal{C}[\![b]\!](\sigma \cup x \mapsto v) \\
& && \quad \textbf{where } \langle u, L\rangle = \sigma[x] \\
&\mathcal{C}[\![m\texttt{(}x\texttt{)}]\!]\sigma &&= \mathit{method}[\![m]\!]u \ \ \textbf{where } \langle u, L\rangle = \sigma[x] \\
&\mathcal{A}[\![\texttt{fill(}B\texttt{,}e_1\texttt{,}e_2\texttt{)}]\!](s,\sigma) &&= \mathit{fill}(s, \mathcal{C}[\![B]\!]\sigma, \mathcal{C}[\![e_1]\!]\sigma, \mathcal{C}[\![e_2]\!]\sigma) \\
&\mathit{fill}(\langle H, V\rangle, c, s, e) &&= \textbf{if } s \geq 0 \wedge e < |V| \wedge \forall_{i \in \{s+1,\ldots,e\}} (V_i = \textbf{empty} \vee V_i = c) \\
& && \quad \textbf{then } \langle H, (\textbf{if } s < i \leq e \textbf{ then } c \textbf{ else } V_i)_{i=1}^{|V|}\rangle \\
& && \quad \textbf{else } \bot
\end{aligned}
$$

Figure 6.10: Semantics for most of the Nonograms DSL (Figure 6.7). Semantics for the pattern-specific *elements* and *method* functions are shown in Figure 6.9.

ramifications of various assumptions about player behavior.

```
def big_hint_rule_arithmetic:
    with h = singleton(hint):
    if 2 * h > line_size:
    do fill(true, line_size - h, h)
```

Figure 6.11: Variant of the basic big hint rule (Figure 6.4) but using arithmetic constructs instead of geometric ones. The designer-provided cost metric for rules is used to measure their relative complexity and choose the more concise.

Actions

The function $\mathcal{A}$ gives meaning to actions as functions from contexts and states to states. In this DSL, we limit actions to filling in a single contiguous chunk of cells with a single value. Actions expressions are of the form **fill**(b,s,e), which says that the board state may be modified by filling in the cells in the range $[s, e)$ (which must both be nonnegative integer expressions with $b < e$ and e at most the length of the input line) with the value b (either **true** or **false**). These limited actions are sufficient to express common line-based strategies for Nonograms (see Section 6.7), but it would be straightforward to support more complex actions since our algorithms are agnostic to the choice of action semantics. In any case, limiting actions does not limit expressiveness: a set of rules with the same pattern and condition but different actions can express any rule that fills in arbitrary cells.

Well-formed Rule Programs

The meaning of rules is defined only for *well-formed programs* (Definition 6.11), in which the binding is valid for the pattern (Definition 6.10). Furthermore, the Nonograms DSL has a simple type system and well-formed programs are well-typed. Due to the simplicity of the type system (e.g., there are no subtypes or other complexities), we omit describing the type system for simplicity of presentation. Every construct in the language evaluates to a fixed

type and expects fixed types for its subexpressions; for example, conditions must evaluate to a boolean value, and pattern-type-specific methods expect an argument of the appropriate pattern element.[7]

Definition 6.11 (Well-Formed Programs). Let $R =$ `with` p`:` `if` c`:` `do` `fill(`b`,` s`,` e`)` be a program in the DSL (Figure 6.7). We say that R is *well-formed* if the following hold:

- The pattern p contains no duplicate variable names.
- For all references r contained in c, s, and e, that variable is defined by p.
- For all expressions in c, s, and e, the arguments are of the right kind and arity.
- The condition c must be a boolean expression, and s and e must be integer expressions.

Example 6.12 (A Complete Example Rule Application). To illustrate the semantics of rules, we look at the general big hint rule from Figure 6.6 (which we'll call R with pattern p, condition c, and action a) on the input from Figure 6.2b. Our input is $s = \langle [4, 3], V \rangle$ where V is a 10-length sequence of **empty** cells. We will considering the binding $\beta = [1]$ (R also applies to s with binding $[0]$).

Looking first at the pattern semantics, we have that:

$$elements[\![\texttt{hint}]\!]s = [\langle [4, 3], V, 0\rangle, \langle [4, 3], V, 1\rangle]$$

Therefore, we have:

$$\begin{aligned} \mathcal{M}[\![\texttt{arbitrary(}\textit{hint}\texttt{)}]\!](s, 1) &= 0 \leq 1 < |L| \;\; \textbf{where } L = elements[\![T]\!]s \\ &= 0 \leq 1 < 2 \\ &= \text{true} \end{aligned}$$

Thus, s matches p under β. Next, $\mathcal{B}$ of any pattern is simply *elements* of that pattern, so:

[7] Having an appropriate type system is somewhat important for efficiency; the more restrictive the type system, the smaller the space of programs that need to be searched during synthesis. For example, in our implementation, we distinguish various types of integers, such as "an index of a cell" or "a length of a span of cells." The expression `start(`e`)` has the former type while `size(`e`)` has the latter.

$$\mathcal{B}[\![\texttt{arbitrary(}\textit{\texttt{hint}}\texttt{)}]\!]s = elements[\![\textit{\texttt{hint}}]\!]s = [\langle[4,3],V,1\rangle,\langle[4,3],V,2\rangle]$$

Putting the pattern together, we have:

$$\begin{aligned}
\mathcal{P}[\![p]\!](s,\beta) &= \mathcal{P}[\![h\ \texttt{=}\ \texttt{arbitrary(}\textit{\texttt{hint}}\texttt{)}]\!](s,\beta) \\
&= \textbf{if } \beta = [\iota_1] \wedge \mathcal{M}[\![\texttt{arbitrary(}\textit{\texttt{hint}}\texttt{)}]\!](s,\iota_1) \\
&\quad\ \textbf{then } \{h \mapsto \langle L_{\iota[1]+1}, L\rangle \mid L = \mathcal{B}[\![\texttt{arbitrary(}\textit{\texttt{hint}}\texttt{)}]\!]s\} \\
&\quad\ \textbf{else } \bot \\
&= \textbf{if } \mathcal{M}[\![\texttt{arbitrary(}\textit{\texttt{hint}}\texttt{)}]\!](s,1) \\
&\quad\ \textbf{then } \{h \mapsto \langle L_2, L\rangle \mid L = elements[\![\textit{\texttt{hint}}]\!]s\} \\
&\quad\ \textbf{else } \bot \\
&= \textbf{if } \text{true} \\
&\quad\ \textbf{then } \{h \mapsto \langle L_2, L\rangle \mid L = [\langle[4,3],V,1\rangle,\langle[4,3],V,2\rangle]\} \\
&\quad\ \textbf{else } \bot \\
&= \{h \mapsto \langle\langle[4,3],V,2\rangle,[\langle[4,3],V,1\rangle,\langle[4,3],V,2\rangle]\rangle\}
\end{aligned}$$

That is, h is the only variable in the pattern and it points to the second hint element (the one with value 3). This is represented by a pair of $(elements[\![\textit{\texttt{hint}}]\!]s)_2$ and $elements[\![\textit{\texttt{hint}}]\!]s$ (i.e., the second element of and the entire sequence of hints, respectively). Let σ be this context.

Looking at the common subexpressions of the condition and action we have:

$$\begin{aligned}
\mathcal{C}[\![\textit{\texttt{lowest_end_cell}}(h)]\!](s,\sigma) &= method[\![\textit{\texttt{lowest_end_cell}}]\!]\langle[4,3],V,2\rangle \\
&= 2-1+\textstyle\sum_{j=1}^{2}[4,3]_j \\
&= 2-1+7 \\
&= 8
\end{aligned}$$

And for the other:

$$\begin{aligned}
\mathcal{C}[\![\textit{\texttt{highest_start_cell}}(h)]\!](s,\sigma) &= method[\![\textit{\texttt{highest_start_cell}}]\!]\langle[4,3],V,2\rangle \\
&= |V| - |[4,3]| + 2 - \textstyle\sum_{j=2}^{|[4,3]|}[4,3]_j \\
&= 10-2+2-3 \\
&= 7
\end{aligned}$$

Now looking at the condition, we have:

$$\begin{aligned}\mathcal{C}[\![c]\!](s,\sigma) &= \mathcal{C}[\![\texttt{lowest_end_cell(}h\texttt{) > highest_start_cell(}h\texttt{)}]\!](s,\sigma)\\ &= \mathcal{C}[\![\texttt{lowest_end_cell(}h\texttt{)}]\!](s,\sigma) > \mathcal{C}[\![\texttt{highest_start_cell(}h\texttt{)}]\!](s,\sigma)\\ &= 8 > 7\\ &= \text{true}\end{aligned}$$

Thus, the condition holds and the rule is applicable. Applying the action, we have:

$$\begin{aligned}\mathcal{A}[\![a]\!](s,\sigma) &= \mathcal{A}[\![\texttt{fill(true, highest_start_cell(}h\texttt{), lowest_end_cell(}h\texttt{))}]\!](s,\sigma)\\ &= \mathit{fill}(s,\\ &\quad\quad \mathcal{C}[\![\texttt{true}]\!](s,\sigma),\\ &\quad\quad \mathcal{C}[\![\texttt{highest_start_cell(}h\texttt{)}]\!](s,\sigma),\\ &\quad\quad \mathcal{C}[\![\texttt{lowest_end_cell(}h\texttt{)}]\!](s,\sigma))\\ &= \mathit{fill}(s, \text{true}, 7, 8)\end{aligned}$$

Putting everything together, the output of applying the general big hint rule to $\langle [4,3], V\rangle$ with binding [1] is the line $\langle [4,3], V\rangle$ where the 7th (0-indexed) cell is replaced with **true**, illustrated in Figure 6.12.

Figure 6.12: The output for Example 6.12.

6.4.4 Creating DSLs for other Domains

While the detailed constructs of our DSL are domain-specific, the structure is general to other logic puzzles. Our system assumes the DSL has the basic structure of patterns, conditions, and actions, but the rest of it can be varied. Our DSL for Nonograms is one of many plausible DSLs with this structure, and similar DSLs could be crafted for games such as Sudoku.

6.5 Characterizing Sound and Useful Rules

Before describing our algorithm for finding sound and useful rules, we must precisely describe what we mean by sound and useful. In this section we formalize both of these notions, highlighting the key challenge (not present in the algebra domain) of finding usefully general rules. As stated previously, we focus on lines (Definition 6.1) as the context for applying strategies, and thus that will be the context for defining soundness and usefulness. Usefulness is primarily defined by a rule's generality: that is, how many states on which it applies and how many cells it fills in on those states.

While our definitions for the notions of soundness and usefulness are specific to Nonograms, analogous notions exists for any puzzle game (e.g., Sudoku) in which the player monotonically makes progress towards the solution through deduction. Our problem formulation assumes the domain has partially ordered (Definition 6.13) states and transitions, but our algorithm is agnostic to the details.

Definition 6.13 (Partial Ordering of States)**.** Given any two states s and t, s is *weaker* than t ($s \preceq t$) iff s and t share the same hints, the same number of cells, and, filled cells in s are a subset of the filled cells in t. In particular, $s \preceq t$ if t is the result of filling zero or more of unknown cells in s. That is, given $s = \langle H_s, V_s \rangle$ and $t = \langle H_t, V_t \rangle$, $s \preceq t$ iff: (1) $H_s = H_t$, (2) $|V_s| = |V_t|$, and (3) $\forall_{i=1}^{|V_s|}$, $(V_s)_i = \textbf{empty} \vee (V_s)_i = (V_t)_i$. Being strictly weaker ($s \prec t$) is being weaker and unequal.

The remaining definitions in this section are with respect to any abstract state space where states support the notions of validity, being solved, and this weakness partial ordering. The key definition for soundness is the *line transition* (Definition 6.14), a pair of states representing a line and the line resulting from filling in zero or more empty cells.

Definition 6.14 (State Transition)**.** Given a state space, a *transition* is a pair of states $\langle s, t \rangle$ where $s \preceq t$. A transition is *productive* iff $s \prec t$.

6.5.1 Soundness of Rules

The notion of soundness is most naturally thought of in terms of relations: some abstract relation on transitions is sound if all pairs represent valid deductions according to the mechanics. Rule programs are partial functions from states and bindings to states, but we can view them as a relation of transitions by projecting over bindings.

We define the soundness of rules with respect to the formal game mechanics, and specifically transitions. Transitions are *valid* if this deduction is sound according to the game mechanics (Definition 6.15). We use an SMT solver to encode the game's mechanics, which gives us an oracle relation that (up to some finite bound) describes all valid transitions. We can straightforwardly characterize soundness with respect to this relation (Definition 6.16). Though we can characterize soundness for arbitrary transitions, in practice we can only verify a finite set of transitions. Therefore, we define soundness with respect to a given finite set of states $\mathbb{S}$. In our implementation, we chose a bound larger than the largest row or column that appeared in any of the puzzles used for training or testing, as described in Section 6.7.

Definition 6.15 (Valid State Transition)**.** A transition $\langle s, t \rangle$ is *valid* iff: (1) both states are valid, and (2) the transition represents a necessary, sound deduction; that is, for any solved state z, if $s \preceq z$, then $t \preceq z$. As we are only concerned with valid states and sound deductions, unless otherwise mentioned, we are implicitly talking about valid transitions.

Definition 6.16 (Soundness of Rules)**.** A program R is sound with respect to a set of states $\mathbb{S}$ iff all pairs of states engendered by applications of R on states in $\mathbb{S}$ are valid transitions. That is, for all $s \in \mathbb{S}, \beta$ where $[\![P]\!](s, \beta) \neq \bot$, $\langle s, [\![P]\!](s, \beta) \rangle$ is a valid transition.

6.5.2 Useful Rules

For this dissertation, the usefulness of rules is primarily defined by how *general* they are.[8] Intuitively, generality is measured by the number of situations in which a rule applies. In a

[8] There are other reasonable ways to define usefulness, some of which are directly counter to generality. For instance, perhaps an educator might be looking for the more specific and concise possible strategy for a given situation in order to help a novice that is stuck.

logic puzzle like Nonograms, this includes not just how many states a rule can be applied to but also how many cells are filled in. Because a rule might apply to the same state in multiple ways (e.g., the big hint rule in Figure 6.2), we must consider each possible binding of a rule. Thus, when comparing two rules for generality, we do so on a per-application basis: a rule R is as general as another rule Q if, for every possible application of Q, there is a corresponding application of R that fills in at least as many cells.

How we formalize this notion depends on the context. First we consider synthesis, described in more detail in Section 6.6.2. As with the algebra domain, RuleSy attempts to find the most general rule for a given specfication with respect to a fixed pattern. Thus in this context, we are interested in finding a maximally general rule with a particular pattern. We define generality for synthesis (Definition 6.17) only for rules with the same pattern. As with soundness, we can (in principle) define this notion with respect to any set of states, but in practice we can only measure generality with respect to a finite set of states.

Definition 6.17 (Generality of Rules)**.** Given rule programs R and Q with the same pattern, we say that R is as or more *general* than Q with respect to a state set $\mathbb{S}$ if, for all $s \in \mathbb{S}$ and bindings β, R applies to all inputs Q does and fills in at least as many cells. We say R is strictly more application-wise general than Q is there exists $s \in \mathbb{S}$ and binding β to which R applies but either Q does not or fills in fewer cells. That is, define $\mathit{apply}(P, s, \beta) = \text{ if } [\![P]\!](s, \beta) \neq \bot \text{ then } [\![P]\!](s, \beta) \text{ else } s$. A program R is as general as a program Q if $\forall_{s \in E, \beta}\, \mathit{apply}(Q, s, \beta) \preceq \mathit{apply}(R, s, \beta)$. The program R is strictly more general than Q if, in addition, $\exists_{s \in E, \beta}\, \mathit{apply}(Q, s, \beta) \prec \mathit{apply}(R, s, \beta)$.

Next consider the context of the domain model optimization phase. Rather than comparing one rule against another, RuleSy needs to compare sets of rules with each other. Generality in this context is further defined specifically over a set of test problems. mention wherever we detail this

6.6 Rule Mining, Synthesis, and Optimization

This section presents the technical details of our system for synthesizing sound, general, and concise condition-action rules for Nonograms. As with the algebra domain, the framework has the same structure as presented in Figure 4.10. We describe our algorithms for each stage and state their guarantees. Proofs can be found in Section 6.10.

6.6.1 Specification Mining

As with algebra, specification mining takes as input a set of examples and a domain definition, and produces a set of specifications for tactic rules. Here, the domain definition takes the form of the formal game mechanics described in Section 6.3. The nature of the domain, particularly the lack of obvious general specifications, means that the specification mining phase is relatively simpler than that for algebra, but the synthesis algorithm is substantially more complicated. We describe the key challenge in specifying Nonograms rules and show how we address it.

Specifying Sound and Useful Rule Behavior

As before, our output specifications must describe sound and useful rule behavior. In algebra, we target learning tactics that were explicit combinations of axiom rules. In Nonograms, we have no such rules to use as the basis for specifications. What we do have, as discussed in Section 6.5.1, is an oracle that defines *all* valid transitions, i.e., a universal rule.

We also have some very specific rules: individual transitions. Every individual transition defines a single maximally specific rule; it applies only to one state. Our approach will be to use this individual input-output examples as specifications. It will be the job of the program synthesis phase of the framework to generalize beyond this single example, finding maximally general rules (with constraints to remain concise) that cover the specification.

As discussed in Section 6.5.1, in practice we can only verify rule behavior over a finite space of inputs. So our specification includes this finite space, defined by the notion of a

transition space (Definition 6.18). This space defines the range of states over which RuleSy will guarantee soundness. In practice, as long as this space is bigger than the space of potential puzzles, rules will be sound for all practical purposes. In our implementation, we select the transition space to be all states up to length 7.

Definition 6.18 (Transition Space)**.** A *transition space* $S = \langle E, \mathcal{T} \rangle$ is a pair of two sets, where E is a non-empty set of states and $\mathcal{T}$ the set of all valid transitions within those states. That is, for all $\langle s, t \rangle \in T$, $s \in E$ and $\langle s, t \rangle$ is a valid transition. As S is determined by the E, we say that S is the transition space on E.

Our full specification will therefore take the form of a triple $\langle S, src, tgt \rangle$, where S is a transition space and $\langle src, tgt \rangle$ is a valid transition. The reader may notice a parallelism between specifications for Nonograms rules and those for algebra rules. They both are defined by a *plan* and a *canonical input-output example.* Though the details are different, each piece serves the same purpose in both domains. The plan defines a sound over approximation of the rule behavior, used both for verification and finding general rules. In the case of Nonograms, the plan is the set of all possible valid transitions, while in algebra it specified a particular way of composing axioms. The canonical example is used both to constrain the rule behavior, but, more importantly, to aid in efficient synthesis. As we'll describe in the following subsection and exactly as we did in algebra, we use this canonical example to define a space of potential patterns to enumerate over. This is important for making synthesis tractable.

Computing Plans

As illustrated in Figure 6.3, specification mining takes as input a set of line states (Definition 6.1) and produces a set of maximal transitions (Definition 6.14), one for each given state, that serve as specifications for the rule synthesis phase. In particular, for every training state s, we use an SMT solver to calculate a target state t such that $\langle s, t \rangle$ is a valid transition, and t is stronger than any state u (i.e., $u \preceq t$) for which $\langle s, u \rangle$ is a valid transition. This

```
1: function FindSpecs(T: set of states, E: set of states)
2:   𝒯 ← {}
3:   for all src ∈ T do            ▷ enumerate all valid transitions using an SMT solver
4:     𝒯 ← 𝒯 ∪ {⟨src, tgt⟩ | ⟨src, tgt⟩ is a valid transition}
5:   𝒮 ← {}
6:   for all src ∈ T do
7:     t ← maximalTransition(src)          ▷ the maximal transition possible from src
8:     for all tgt ∈ decomposeActions(src, t) do     ▷ break t into expressible actions
9:       𝒮 ← 𝒮 ∪ {⟨⟨E, 𝒯⟩, src, tgt⟩}
10:  return 𝒮
```

Figure 6.13: FindSpecs takes as input a training set T and a verification set E. Given these inputs, it returns a set of transition-space/transition pairs specifying nonograms rules.

calculation is straightforward: for each unknown cell in s, we ask the SMT solver whether that cell is necessarily **true** or **false** according to the rules of Nonograms, and fill it (or not) accordingly. The resulting transitions, and therefore rule specifications, represent the strongest possible deductions that a player can make for the given training states.

The effectiveness of our mining process depends critically on the choice of the training set. If the training set is too small or haphazardly chosen, the resulting specifications are unlikely to lead to useful rules. We want a variety of states, so enumerating states up to a given size is a reasonable choice. But for lines of size n, there are between 2^n and 4^n possible transitions, so we choose a relatively small value (in our evaluation, a random subset of lines of size $n \leq 7$), relying on the rule synthesis phase to generalize these rules so they apply on the larger states in the testing set.

One important consideration is the mismatch between the space of possible transitions and the space of transitions expressible by actions in the DSL. A transition $\langle s, t \rangle$ may fill in an arbitrary set of empty cells of s with **true** or **false** or both, but the actions in our DSL can only fill in contiguous runs of cells with a single value. Therefore, a transition may represent multiple rules that we could learn, so we need to decompose a transition into pieces whose actions can actually be expressed in the DSL. In order to be complete, RuleSy would need to try all possible actions compatible with a transition. In our practical

implementation, we greedily decompose a transition into maximally sized chunks.This may cause us to miss some rules but shortens synthesis time substantially while still finding many useful rules. Note that, even if the DSL had a more expressive action language, we would still need this: any given example might represent two qualitatively distinct rules being applied simultaneously, so if we want RuleSy to find such rules, we must try various decompositions of the transition. example of decomposing transitions some note about how, in practice, we don't use *all* states but only those that lead to canonical transitions

Definition 6.19 (Maximal State Transition)**.** A valid transition $\langle s, t\rangle$ is *maximal* iff, for all valid transitions $\langle s, u\rangle$, $u \preceq t$.

Theorem 6.20 (Guarantees of Specification Mining)**.** Let $T \subseteq E$ be sets of Nonograms line states. The function FindSpecs(T, E) terminates and produces a set $\mathcal{S}$ of transition space and transition triples with the following properties: (1) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, $S = \langle E, \mathcal{T}\rangle$ is the transition space for E (2) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, $\langle \mathit{src}, \mathit{tgt}\rangle$ is a valid transition in $\mathcal{T}$ that can be expressed by some action expression, and (3) for every maximal transition $\langle \mathit{src}, \mathit{tgt}\rangle$ where $\mathit{src} \in T$, there exist tuples $\langle S, \mathit{src}, \mathit{tgt}_1\rangle, \ldots, \langle S, \mathit{src}, \mathit{tgt}_k\rangle$ in $\mathcal{S}$ such that $\mathit{fillset}(\langle \mathit{src}, \mathit{tgt}\rangle) \subseteq \bigcup_{i\in\{1,\ldots,k\}} \mathit{fillset}(\langle \mathit{src}, \mathit{tgt}_i\rangle)$.

6.6.2 Rule Synthesis

Given specifications of the form $\langle S, \mathit{src}, \mathit{tgt}\rangle$, RuleSy synthesizes tactics by searching for well-formed, sound programs that satisfy them. As in the algebra domain, this search is a form of syntax-guided synthesis [1], using an automatic verifier to check if a chosen candidate program from a given syntactic space satisfies the specification. We must solve the same challenges as the previous domain: RuleSy needs (1) an automatic verifier for its DSL, and (2) a method for pruning the candidate space without omitting any correct implementations. We address both challenges by exploiting the structure of well-formed rule programs and specifications produced by FindSpecs. In this section, we illustrate the challenges of classic syntax-guided synthesis for rule programs, show how we address then them, and present the

FindRules algorithm for sound, complete, and efficient solving of this query.

Basic Synthesis for Rule Programs

In this setting, every specification $\langle S, \mathit{src}, \mathit{tgt} \rangle$ has the same transition space S. Thus, obviously, the classical synthesis query where the program matches S on all inputs is overly strict; S represents the universal rule of all possible transitions (up to a finite bound). We want to find some rule that covers a subset of the transition space while specifically including the transition $\langle \mathit{src}, \mathit{tgt} \rangle$. Thus, the basic synthesis query in our setting takes the following form, where $S = \langle E, \mathcal{T} \rangle$:

$$\exists_R \, (\exists_\beta \, [\![R]\!](\mathit{src}, \beta) = \mathit{tgt}) \wedge (\forall_{s \in E, \beta} \, [\![R]\!](s, \beta) \neq \bot \implies \langle s, [\![R]\!](s, \beta) \rangle \in \mathcal{T}) \tag{6.1}$$

Existing off-the-shelf synthesis tools [1, 124, 133] can solve this query since the input space is of finite size and programs are loop free (assuming we search over a finite space of programs). However, these tools can fail to find useful rules because this query fails to specify that a rule should be maximally general, which is the challenge RuleSy must address. For example, a valid answer to this query is a rule that admits only the transition $\langle \mathit{src}, \mathit{tgt} \rangle$, which is obviously not particularly useful. We describe our solution below.

The Best-Implements Synthesis Query

Our solution is to reformulate the synthesis query as we did in algebra: we would like to find as many rules as possible that fire on *src* to produce *tgt* and capture a locally maximal subset of behaviors, which we capture in the *best-implements* search query (Definition 6.22) and search for with the FindRules algorithm (Figure 6.17). In practice, there can be many different maximally general rules: that is, multiple rules that cover $\langle \mathit{src}, \mathit{tgt} \rangle$, are each locally maximally general, but which cover non-overlapping subsets of the transition space. As in the tree rewrite domain, two program R and Q can differ in this way because they have different patterns. But, as illustrated in Example 6.21, R and Q could have the same pattern

and both be maximally general but still have different behavior from different conditions.

Example 6.21.

As a practical implementation matter, we cannot feasibly search for every single maximally general rule that matches a given specification $\langle S, src, tgt \rangle$. But it is important to explore multiple rules: to see why, consider the example illustrated in Figure 6.14. This rule is strictly more general than the big hint rule from Figure 6.6, but is substantially more complicated (i.e., less concise). Figure 6.15 illustrates an input showing that it is more general. We want RuleSy to find rules like that in Figure 6.6 that balance generality with conciseness. To achieve this, RuleSy has a compromise: for a single synthesis query, RuleSy will greedily find only one maximally general rule. However, RuleSy will run multiple queries for each specification with various constraints on program size and behavior in order to find rules or various levels of conciseness.[9]

```
def big_hint_rule_complex:
    with h = arbitrary(hint) and
         g = constant(gap, 0):
    if start(g) > highest_start_cell(h) - lowest_end_cell(h):
    do fill(true,
            highest_start_cell(h),
            lowest_end_cell(h) + start(g))
```

Figure 6.14: A more general yet more complex version of the big hint rule than the one from Figure 6.6.

For one, RuleSy will enumerate over all possible patterns. This allows RuleSy to find both the big hint rules illustrated in Figures 6.6 and 6.14. Beyond enumerating over patterns, RuleSy enumerates over various bounds on program shapes. This allows RuleSy to find multiple maximally general rules within a single pattern. Thus, our best-implements

[9]This process can be thought of as being akin to *regularlization* from statistical machine learning: RuleSy constrains parts of the program to prevent over-fitting to a particular example. By trying many different queries, RuleSy can greedily optimize only generality while finding a variety of rules to balance generality versus conciseness.

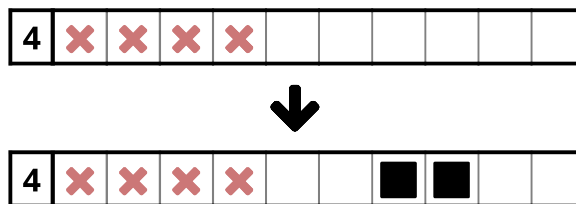


Figure 6.15: An example transition to which the basic big hint rule (Figure 6.4) does not apply but the more complex one (Figure 6.14) does.

synthesis query will be defined with respect to both a given fixed pattern and given bound on program size.[10]

Definition 6.22 (Best Implementation)**.** Let $\langle S, \mathit{src}, \mathit{tgt} \rangle$ be a specification, p be a pattern, and $\bar{k}$ be a bound on program size. A well-formed program R with pattern p *best implements* $\langle S, \mathit{src}, \mathit{tgt} \rangle$ under bounds $\bar{k}$ if:

1. R has size at most $\bar{k}$,
2. R satisfies the basic synthesis query for $\langle S, \mathit{src}, \mathit{tgt} \rangle$ (Equation 6.1), meaning it is sound under S and fires on *src* to produce *tgt*.
3. R is maximally general with respect to S (Definition 6.17) over all programs with pattern p and size at most $\bar{k}$.

Sound and Efficient Search

As the best implementation is defined with respect to both a particular pattern and a bound on program size, RuleSy must enumerate over all patterns and a (user-specified) range of potential program sizes. For patterns, we exploit the exact same trick as the algebra domain: using *pattern refinement* (Definition 6.23). Any best implementation R of $\langle S, \mathit{src}, \mathit{tgt} \rangle$ must fire on *src* to produce *tgt*, and therefore the pattern of R must *refine* the unique most refined

[10]Conveniently, restricting the pattern and program shapes makes the synthesis task substantially more tractable by dramatically reducing the space of potential programs to search.

pattern for *src*. There are finitely many such patterns and thus they can be enumerated (Line 3 of Figure 6.17), starting with the most refined such pattern (Line 2).

Definition 6.23 (Pattern Refinement). A condition pattern p_1 *refines* a pattern p_2 if $p_1 \sqsubseteq p_2$, where $\sqsubseteq$ is defined as follows:

- $p \sqsubseteq p$
- **singleton**(t) $\sqsubseteq$ **constant**(t, 0)
- **constant**(t, k) $\sqsubseteq$ **arbitrary**(t)
- p_1 **and** ... **and** $p_n \sqsubseteq p_1$ **and** ... p_{i-1} **and** p_{i+1} ... **and** p_n for all $0 < i \leq n$
- p_1 **and** ... **and** p_i **and** ... **and** $p_n \sqsubseteq p_1$ **and** ... **and** p'_i **and** ... **and** p_n if $p_i \sqsubseteq p'_i$

Note that, by definition, any given pattern refines finitely many other patterns.

Example 6.24 (Pattern Refinement). To illustrate the concept of the most refined pattern for a given state, consider the line state in Figure 6.16. The most refined pattern for this line is:

```
constant(hint,0) and constant(hint,1) and
singleton(block) and
constant(gap,0) and constant(gap,1) and
constant(cell,0) and constant(cell,1) and
constant(cell,2) and constant(cell,3) and
constant(cell,4)
```

In practice, useful general rules use relatively few bound elements (every variation of the big hint rule we've discussed uses only one or two, for example). We can significantly improve the performance of pattern generalization by searching for programs with fewer patterns first. Referencing Example 6.24, rather than finding rules with all 10 pattern elements, we would search for programs that use small subsets of them, in increasing order. Our implementation stops after a fixed upper bound on size but in principle could enumerate over all of them.

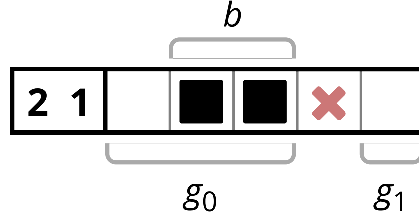


Figure 6.16: An example line state whose most refined pattern is described in Example 6.24. The **`singleton`** block and **`constant`** gaps are labeled as b and g_1, g_2, respectively.

Theorem 6.25 (Partial Ordering of Pattern Refinement)**.** Let p_1, p_2 be patterns, c be a condition, a be an action, and R_1, R_2 be the well-formed programs $R_1 =$ **`with`** p_1**`:`** **`if`** c**`:`** **`then`** a and $R_2 =$ **`with`** p_2**`:`** **`if`** c**`:`** **`then`** a. If p_1 refines p_2 ($p_1 \sqsubseteq p_2$), then R_2 is as general as R_1 and has matching behavior for applicable inputs to R_1. That is, $\forall t, \beta. [\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$.

In order to find rules that best implement the specification (i.e., are maximally general), FindRule uses a greedy iterative optimization. FindRule first finds any sound rule that fires from *src* to *tgt* (Line 11). The function then attempts to find a new sound rule that strictly more general than the previous best (Lines 12–14). Upon failure to synthesize a new rule, the function is guaranteed to have a (locally) maximally general rule.[11] As discussed, this technique is greedy; we will find some arbitrary locally most general rule. We rely on enumerating over many synthesis queries to find a variety of rules. We could straightforwardly extend the system to produce more maximally general by restarting the search after hitting a local optimum and adding constraints to cover states not covered by *any* of the previous rules, but such a strategy would substantially increase synthesis time.

Theorem 6.26 (Guarantees of Rule Synthesis)**.** Let $\langle S, \mathit{src}, \mathit{tgt}\rangle$ be a specification and $\bar{k}$ a bound on the size of rule programs. FindRules$(S, \mathit{src}, \mathit{tgt}, \bar{k})$ returns a set of rules $\mathcal{R}$ with the following properties: (1) every $R \in \mathcal{R}$ best implements S for $\langle \mathit{src}, \mathit{tgt}\rangle$ under $\bar{k}$; (2) $\mathcal{R}$

[11] In both this domain and the algebra domain, we also use this type of loop-until-failure iteration to find maximally concise rules. In the Nonograms domain, we first maximize generality, then minimize the program size while preserving generality.

```
1:  function FindRules(S: transition space, src, tgt: states, k̄: ints, n: int)
2:    p0 ← termToPattern(s)                              ▷ Most refined pattern that matches s
3:    R ← ⋃_{p0 ⊑ p} FindRule(p, S, src, tgt, k̄, n)
4:    return R                                   ▷ Rules that best implement S for ⟨src, tgt⟩

5:  function FindRule(p: pattern, ⟨E, T⟩: transition space, s, t: states, k̄: ints, n: int)
6:    C ← WellFormedConditionHole(p, k̄)                                  ▷ Condition sketch
7:    A ← WellFormedActionHole(p, k̄)                                        ▷ Action sketch
8:    R ← with p: if C: do A
9:    𝕊 ← {σ | M⟦p⟧σ ∧ σ ∈ E}                   ▷ Symbolic representation of states satisfying p
10:   best ← ⊥
11:   p ← CEGIS(Sound(R, T, 𝕊) ∧ Matches(R, src, tgt))
12:   while p ≠ ⊥ do                                   ▷ greedily generalize rule until failure
13:     best ← p
14:     p ← CEGIS(Sound(R, T, 𝕊) ∧ Matches(R, src, tgt) ∧ MoreGeneral(R, best, 𝕊))
15:   return {best | best ≠ ⊥}

16: function Sound(R: program, T: transition set, 𝕊: symbolic state set)
17:   return ∀_{σ∈𝕊,β} ⟦R⟧(σ, β) ≠ ⊥ ⟹ ⟨σ, ⟦R⟧(σ, β)⟩ ∈ T

18: function Matches(R: program, src, tgt: states)
19:   return ∃β. ⟦R⟧(src, β) = tgt

20: function MoreGeneral(R: program, Q: program, 𝕊: symbolic state set)
21:   apply ← λ(P, s, β). if ⟦P⟧(s, β) ≠ ⊥ then ⟦P⟧(s, β) else s
22:   return (∀_{s∈𝕊,β} apply(Q, s, β) ⪯ apply(R, s, β)) ∧ (∃_{s∈𝕊,β} apply(Q, s, β) ≺ apply(R, s, β))
```

Figure 6.17: FindRules takes as input a specification $\langle S, src, tgt\rangle$ and a bound $\bar{k}$ on program size. Given these inputs, it returns a set of rules best implementing S for $\langle src, tgt\rangle$ under $\bar{k}$.

includes a sound rule R of size at most $\bar{k}$ if one exists; and (3) for every pattern p that refines or is refined by R's pattern, $\mathcal{R}$ includes a sound rule with pattern p and size $\bar{k}$ if one exists.

6.6.3 Rule Set Optimization

After synthesizing a set of rules $\mathcal{R}$ from the training examples E, RuleSy applies discrete optimization to find a subset of $\mathcal{R}$ that minimizes a given objective function f, defined over a set of testing examples F. As with algebra, the example objective function we consider is one that evaluates a set of rules based on the balance of conciseness and how well the rules can be used to solve problems. The primary challenge in this phase is characterizing what it means for a set of rules to be useful to solve problems, and solving this challenge is our primary contribution in the phase.

Basic Rule Set Optimization Algorithm

Given a set of rule programs $\mathcal{R}$ and a testing set of examples F, the rule set optimization algorithm selects a subset of $\mathcal{R}$ that is the most general over F (as per Definition 6.17). For F, our evaluation uses a set of states drawn from solution traces of real-world puzzles. To choose a subset of rules with the best coverage of the testing set, we set up a discrete optimization problem with the following objective: select k rules (for some fixed k) that cover the greatest proportion of (maximal) transitions from the testing set. For this optimization, we measure coverage by the total number of cells filled. That is, the coverage of a test item can be partial; the objective function awards a score for each cell filled. Greedy methods suffice for this optimization.

Using an oracle for decomposition.

The natural way to characterize generality on the testing set is the same way we do during synthesis: treat each example in F as a potential input to each rule in $\mathcal{R}$ and measure how many cells are filled in. However, evaluating rules in this way is not reflective on how people

use rules to solve problems: human players use the entire set of rules to make progress on particular lines. Thus, we measure the generality of a set of rules $\mathcal{R}$ by applying all rules to fixed point on each state in F.

When human players apply greedy strategies, they do so by considering both states, such as lines, and substates, such as parts of a line. If a player can deduce that certain hints must be constrained to certain cell ranges (as illustrated in Figure 6.18), then the player can focus on the identified substate (essentially, a smaller line), which might make new rules available, or at least make case-based-reasoning simpler. This form of problem decomposition is often required for applying strategies, especially on very large boards. We do not model this strategy in our system, but want to account for it when evaluating rules.[12]

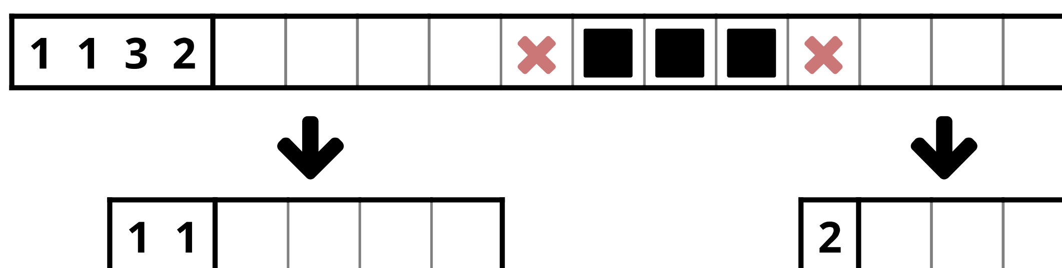


Figure 6.18: An example of state decomposition. Because hints are ordered, if we know where one hint lies (in this case, hint 3), then we can consider sub-lines in isolation. This allows us to apply the *big hint* rule (Figure 6.4) to the right sub-line.

In order to account for this player behavior when evaluating our objective function, we use an SMT solver as an oracle for decomposition.[13] That is, to measure how much of a transition can be solved with a given set of rules, we apply both the rules and the decomposition oracle to a fixed point. This allows us to measure the impact of the rules in a more realistic way than

[12]However, future work could create a new DSL to decide when it's safe to decompose and use RuleSy to learn rules for it!

[13] The *Picross* series of videogames for the Nintendo DS, 3DS, and Switch actually provide a limited version of this oracle to the player. For each partially-filled line, the game tells the player which (if any) of the hints are satisfied. The reasoning is based only on the partial state, not the solution; it uses some deductive/search-based procedure.

under the assumption that greedy strategies are used on their own, without decomposition.

6.7 Evaluation

To evaluate our system, we compared its output to a set of documented strategies from Nonograms guides and tutorials. Unlike Sudoku, there is no comprehensive database of strategies for Nonograms, so we recovered these *control rules* from various sources: the introduction to a puzzle book [92], the tutorial of a commercial digital game [64], and the Wikipedia entry for Nonograms.[14] These sources demonstrate rules through examples and natural language, so, to encode them in our DSL, some amount of interpretation is necessary. In particular, while these sources often explain the reasoning behind a strategy, the strategy is demonstrated on a simple case, letting the reader infer the general version. We encoded the most general (when possible, multiple) variants of the demonstrated rules in our DSL, for a total of 14 control rules. The implementation source code and evaluation data are available online.[15]

We evaluated our system by asking the following questions:

1. Can the system recover known strategies by learning rules in the control set?

2. How does the learned set compare to the control set when measuring coverage of the testing data?

Training Data. For this evaluation, we trained the system using a random subset of maximal transitions of lines up to length 7. There were 295 such states. Note that, excepting tutorial puzzles, no commercial puzzles are on boards this small, so none of the testing examples are this small.

[14] `https://en.wikipedia.org/wiki/Nonogram#Solution_techniques`

[15] `https://github.com/edbutler/nonograms-rule-synthesis`

Testing Data. Our testing data is drawn from commercial Nonograms puzzle books and digital games *Original O'Ekaki* [98], *The Essential Book of Hanjie and How to Solve It* [92], and the *Pircoss e* series [63, 65, 64]. We randomly selected 17 puzzles from these sources. The lines range in size from 10 to 30. All puzzles are solvable by considering one line at a time. To get test data from these boards, we created solution traces with random rollouts by selecting lines, using our oracle to fill the maximum possible cells, and repeating until the puzzle was solved. We took each intermediate state from these solution traces as testing states. This resulted in 2805 testing states.

Learned Rules. From our training examples, the first two phases of the system (specification mining and rule synthesis) learned 1394 semantically distinct rules as determined by behavior on the test set. Two learned rules are shown in Figures 6.19 and 6.20. We measure the quality of the learned rules by comparing them to the control rule set, both on whether the learned set includes the control rules and how well the learned rules perform on the testing data.

6.7.1 Can the system automatically recover the control rules?

Our system recovered 9 of the 14 control rules, either exactly, or as a more general rule. Figure 6.19 shows an example of a rule from the control set for which our system learned a syntactically identical rule. While the training set included example states for all control rules, our greedy generalization algorithm choose a different generalization than the one represented by the missed control rules. As discussed in Section 6.6.2, we could extend the system to explore multiple generalizations. Given sufficient training time, such a system would find these particular rules as well. In some cases where our system did not match a control rule, it did find a qualitatively similar rule that covered many of the same states (as in Figure 6.20).

```
# crossing out the cell next to a satisfied hint,
# which can be determined because it's (one of)
# the biggest hints.
def punctuate_maximum:
    # for any arbitrary hint and block
    with h = arbitrary(hint) and
         b = arbitrary(block):
    # if the hint is maximal,
    # the block and hint have the same size,
    # and the block is strictly right of the left edge,
    if maximal(h) and size(b) = size(h) and start(b) > 0:
    # then cross out the cell to the left of the block
    do fill(false, start(b) - 1, start(b))
```

Figure 6.19: An example of a control rule that our system recovers exactly, annotated with comments. This is a top-10 rule as determined by the rule set optimization.

6.7.2 How does an optimal subset of rules compare on coverage?

In order to quantitatively compare the coverage of our learned set to the control set, we measured the proportion of the maximal transitions of the testing examples that each rule set covered. As described in Section 6.6.3, we measure this by the proportion of transitions covered; for a set of rules $\mathcal{R}$, the coverage $C(\mathcal{R})$ is the total number of cells over all testing examples covered by applying rules in $\mathcal{R}$ and the decomposition oracle to a fixed point.

Coverage of learned rules First, we compared the entire rule sets. On our test examples, the 14 control rules $\mathcal{R}_0$ have a coverage $C(\mathcal{R}_0)$ of 4652. Our trained rule set $\mathcal{R}_t$ has a coverage $C(\mathcal{R}_t)$ of 7558. These sets are incomparable; the control rules cover some items that the learned do not and vice-versa. Looking at the union of the two sets, they have a total coverage $C(\mathcal{R}_t \cup \mathcal{R}_0)$ of 7673. That is, the learned set alone covers over 98% of the transitions covered by the learned and control sets together. This means that, even though we do not recover the control rules exactly, the learned rules cover nearly all test cases covered by the missed control rules.

[16] `https://en.wikipedia.org/wiki/Nonogram#Mercury`

```
# crossing out the left side of the line if a block
# is more than hint-value distance from the edge.
def mercury_variant:
    # for singleton hint and arbitrary block
    with h = singleton(hint) and
         b = arbitrary(block):
    # if the right side of the block is
    # greater than the value of the hint
    if start(b) + size(b) > size(h):
    # then cross out cells from 0 up through the
    # one that is hint-value many cells away from
    # the right edge of the block.
    do fill(false, 0, start(b) + size(b) - size(h))
```

Figure 6.20: An example top-10 rule learned by our system that is *not* in the control set, annotated with comments. This rule is similar to what we call the *mercury* control rule, which is not recovered exactly. But the learned rule covers a large portion of the same states. While slightly less general, it is significantly more concise than the control rule, using one fewer pattern, one fewer condition, and less complex arithmetic. The learned rule is also a reasonable interpretation of the description on which the control rule is based.[16]

Coverage of a limited set of rules We would expect that the very small control set would have less coverage than the large set of learned rules. For a more equitable comparison, we measure the top-10 rules from each set, using the rule set optimization phase of our system. Choosing the top 10 rules, the top 10 control rules have a coverage of 4652 (unchanged from the full 14), and the top 10 learned rules have a coverage of 6039. The learned rules, when limited to the same number as the control rules, still outperform the control on the testing examples. Figure 6.20 shows an example of a learned rule in the top-10 set that was not in the control set. Comparing the complexity of these rules with the cost function, the top-10 control rules have a mean cost of 31.7 while the top-10 from the learned set have a mean cost of 33.7. Though our learning algorithm minimizes individual rule cost, the optimization greedily maximizes coverage while ignoring cost.

These results suggest that our system can both recover rules for known strategies and cover more states from real-world puzzles.

6.8 Related Work

Automated Game Analysis Automated game analysis is a growing research area that aims to uncover designer-relevant information about games without human testing [97, 119, 102, 132], which is needed in situations where human testing is too expensive or of limited effectiveness [139]. Researchers have investigated estimating game balance [60, 139] and using constraint solving to ensure design constraints are satisfied [118]. These approaches typically reason at the level of game mechanics or available player actions. We contend that for many domains, such as logic puzzles, the mechanics do not capture the domain-specific features players use in their strategies, necessitating representations that contain such features.

General Game AI [103] is a related area in automatically understanding game strategies. However, it tackles the problem of getting a computer to play a game while we tackle the different problem of finding human-interpretable strategies for playing a game.

Prior research has also looked at analyzing the difficulty of logic puzzles. Browne presents an algorithm called *deductive search* designed to emulate the limits and process of human solvers [23]. Batenburg and Kosters estimate difficulty of Nonograms by counting the number of steps that can be solved one row/column at a time [13]. This line of work relies on general difficulty metrics, while our work models the solving process with detailed features captured by our DSL for rules.

Interpretable Learning Finding interpretable models has become a broader topic of research because there are many domains where it is important, such as policy decisions. Researchers have looked at making, e.g., machine learning models more explainable [72, 112]. Many of these techniques focus on either (1) learning an accurate model and then trying to find an interpretable approximation (e.g., [57]), or (2) restricting the space of models to only those that are interpretable. We take the latter approach, targeting a domain not addressed by other work with a unique approach of program synthesis.

Modeling Games with Programming Languages Game description languages are a class of formal representations of games, many of which were proposed and designed to support automated analysis. Examples include languages for turn-based competitive games [86] and adversarial board games [101, 24]. Osborn et al. [102] proposed the use of such a language for computational critics. The Video Game Description Language (VGDL) [110] was developed to support general game playing. Operational logics [89] deal with how humans understand the game, and Ceptre [87] is a language motivated by gameplay. We share goals here, but are arguing to build a new DSL for each game. All of these prior languages model the rules or representation of the game, while our language models concepts meaningful to players in order to capture strategies at a fine-grained level, which necessitates the inclusion of domain-specific constructs.

Program Synthesis Program synthesis, the task of automatically finding programs that implement given specifications [48], is well-studied and has been used for a variety of applications. Notably, program synthesis has been used for several applications in problem-solving domains, from solution generation [51] to problem generation [3] to feedback generation [113].

One challenging feature of our program synthesis problem is its underspecification. Some methods address this challenge with an interactive loop with the user to refine specifications [48], while others rank possible programs and select a single best one [104]. Our method ranks programs using the metrics of generality and conciseness, and differs from prior work in that we are choosing a set of programs that best implement a set of specifications, rather than a single program for a single specification.

6.9 Conclusion

This chapter presented an instantiation of RuleSy for the puzzle game Nonograms. Our system takes as input the rules of Nonograms, a domain-specific language (DSL) for expressing strategies, a set of training examples, and a cost function for measuring rule complexity. Given these inputs, it learns general, concise programs in the DSL that represent

effective strategies for making deductions on Nonograms puzzles. The DSL defines the domain-specific constructs and features of Nonograms, ensuring that our learned strategies are human-interpretable. Since the DSL is taken as input, the designer can use it to encode domain-specific considerations and bias. When compared with documented strategies recovered from guides and tutorials, the rules our system learns recover many existing strategies exactly and outperform them when measured by coverage on states from real-world puzzles.

This work focused on automatic synthesis of human-interpretable strategies. But the eventual goal is to find strategies that humans are likely to use. Avenues of future work thus include using player solution traces for cost estimation. For example, rather than using a designer-provided cost function to estimate rule complexity, we can attempt to learn a cost function from play traces.

Our work enables a range of applications, such as game-design feedback, difficulty estimation, and puzzle and progression generation. For example, previous puzzle game generation research relied on a designer-authored space of concepts and solution features to generate a progression of puzzles [25]. Tools like the one presented in this paper can serve as input for such systems.

While we applied our system to Nonograms, we expect it to be applicable to other puzzle games as well. The system assumes that the input DSL has the basic structure of patterns, conditions, and actions, but is agnostic to the detailed constructs. The presented system is designed for logic puzzles of this structure, but we believe that program synthesis can be used to learn human-interpretable strategies in a wider range of games and problem-solving domains.

Acknowledgements. This research was supported in part by NSF CCF-1651225, 1639576, and 1546510; and Oak Foundation 16-644.

6.10 Appendix: Proofs of Theorems

In this section, we restate and provide proofs for all theorems stated earlier in the chapter.

Theorem 6.20 (Guarantees of Specification Mining)**.** Let $T \subseteq E$ be sets of Nonograms line states. The function FINDSPECS(T, E) terminates and produces a set $\mathcal{S}$ of transition space and transition triples with the following properties: (1) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, $S = \langle E, \mathcal{T}\rangle$ is the transition space for E (2) for every $\langle S, \mathit{src}, \mathit{tgt}\rangle \in \mathcal{S}$, $\langle \mathit{src}, \mathit{tgt}\rangle$ is a valid transition in $\mathcal{T}$ that can be expressed by some action expression, and (3) for every maximal transition $\langle \mathit{src}, \mathit{tgt}\rangle$ where $\mathit{src} \in T$, there exist tuples $\langle S, \mathit{src}, \mathit{tgt}_1\rangle, \ldots, \langle S, \mathit{src}, \mathit{tgt}_k\rangle$ in $\mathcal{S}$ such that $\mathit{fillset}(\langle \mathit{src}, \mathit{tgt}\rangle) \subseteq \bigcup_{i \in \{1,\ldots,k\}} \mathit{fillset}(\langle \mathit{src}, \mathit{tgt}_i\rangle)$.

Proof. Termination follows from the fact that all loops in FINDSPECS iterate over finite structures. Corectness (1) follows from lines 3–4, line 9, and Definition 6.18 and further (2) follows from the assumption that $T \subseteq E$, lines 6–9, and the fact that decomposing a valid transition results in valid transitions. Completeness (3) follows from line 8. □

Theorem 6.25 (Partial Ordering of Pattern Refinement)**.** Let p_1, p_2 be patterns, c be a condition, a be an action, and R_1, R_2 be the well-formed programs $R_1 =$ `with` p_1`:` `if` c`:` `then` a and $R_2 =$ `with` p_2`:` `if` c`:` `then` a. If p_1 refines p_2 ($p_1 \sqsubseteq p_2$), then R_2 is as general as R_1 and has matching behavior for applicable inputs to R_1. That is, $\forall t, \beta.\, [\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$.

Proof. The proof follows from the semantics of patterns, inducting on the cases in Definition 6.23. For any term t and binding β, we have that both $\mathcal{M}[\![p_2]\!](t, \beta) \implies \mathcal{M}[\![p_1]\!](t, \beta)$ and $\mathcal{B}[\![p_2]\!](t, \beta) = \mathcal{B}[\![p_1]\!](t, \beta)$ (or, in the case where $p_1 = a_1$ `and` ... a_m and $p_2 = b_1$ `and` ... b_n, $\forall_{i \in \{1,\ldots,n\}} \mathcal{M}[\![b_i]\!](t, \beta) \implies \forall_{i \in \{1,\ldots,m\}} \mathcal{M}[\![a_i]\!](t, \beta)$ and the resulting context for p_2 is a subset or that for p_1). From this and the assumptions that R_1 and R_2 are well-formed and share a condition and action, it follows that $[\![R_1]\!](t, \beta) \neq \bot \implies [\![R_1]\!](t, \beta) = [\![R_2]\!](t, \beta)$. □

Lemma 6.27. Let p be a pattern, S be a transition space, and $\langle s, t\rangle$ be a valid transition such that $\mathcal{P}[\![p]\!]s \neq \bot$. Also let `with` p`:` `if` C`:` `do` A be a sketch with well-formed holes of size $\bar{k}$. If this sketch includes a rule that best implements S for $\langle s, t\rangle$ under $\bar{k}$, FINDRULE$(p, S, s, t, \bar{k})$ returns a singleton set containing such a rule; otherwise, it returns $\emptyset$.

Proof. The proof consists of two parts. First, we show that the synthesis queries satisfy the definition of best implementation. Then, we show that the CEGIS engine [133] is sound and complete for these queries.

Best Implementation. We first show the queries guarantee the rule returned (if any) best implements S for $\langle src, tgt \rangle$ under $\bar{k}$, if any such rule exists. Looking at the queries used in FINDRULE, the sketch created on lines 6–8 is up to bound $\bar{k}$ (criteria (1) for Definition 6.22). The functions SOUND and MATCHES used in both queries (lines 11 and 14) ensures any rule assigned to p and thus returned matches criteria (2); furthermore, because the query on line 11 is the minimal constraint for (2), such a rule will be found if it exists. Finally, the while loop (lines 12–14) will terminate only when either (a) no sound rule exists or (b) *best* contains a maximally general rule, meeting criteria (3).

CEGIS. FINDRULE uses an off-the-shelf CEGIS engine [133] to solve the synthesis queries at lines 11 and 14. This engine is sound and complete for finite (loop-free) programs and inputs, which is the case for our queries. □

Theorem 6.26 (Guarantees of Rule Synthesis)**.** Let $\langle S, src, tgt \rangle$ be a specification and $\bar{k}$ a bound on the size of rule programs. FINDRULES$(S, src, tgt, \bar{k})$ returns a set of rules $\mathcal{R}$ with the following properties: (1) every $R \in \mathcal{R}$ best implements S for $\langle src, tgt \rangle$ under $\bar{k}$; (2) $\mathcal{R}$ includes a sound rule R of size at most $\bar{k}$ if one exists; and (3) for every pattern p that refines or is refined by R's pattern, $\mathcal{R}$ includes a sound rule with pattern p and size $\bar{k}$ if one exists.

Proof. Line 2 defines p_0 to be the most refined pattern that matches s, so by Definition 6.23 and Theorem 6.25, p_0 refines the pattern of every rule applicable to s. Since there are finitely many such patterns (Definition 6.23), termination, soundness (1), and completeness (2–3) follow from line 3 and Lemma 6.27. □

6.11 Appendix: Control Rules

In this section, we list all of the control rules used in the evaluation. Note that, unlike the algebra domain, these are not axioms. These are rules collected from various published Nonograms media used as comparison for RuleSy's synthesized rules.

```
def rule-big-hint-basic:
    with a = arbitrary(hint-basic):
    if value(a) + value(a) > linelen(a):
    then fill(true, (linelen(a) - value(a)), value(a))

def rule-big-hint:
    with h = arbitrary(hint):
    if lec(h) > hsc(h):
    then fill(true, hsc(h), lec(h))

def rule-edge-fill:
    with h = constant(hint, 0) and
         b = arbitrary(block):
    if lec(h) > end(b):
    then fill(true, end(b), lec(h))

def rule-edge-fill/cell:
    with h = constant(hint, 0) and
         c = arbitrary(cell):
    if (lec(h) - 1) > index(c) and
       true?(c):
    then fill(true, index(c) + 1, lec(h))
```

```
def rule-edge-fill/cell/hint-basic:
    with a = constant(hint-basic, 0) and
         c = arbitrary(cell):
    if (value(a) - 1) > (index(c) - 0) and
       true?(c):
    then fill(true, index(c) + 1, 0 + value(a))


def rule-edge-fill/true-cell:
    with h = constant(hint, 0) and
         t = constant(cell-t, 0):
    if (lec(h) - 1) > index(t):
    then fill(true, index(t) + 1, lec(h))


def rule-full-edge-gap:
    with h = constant(hint, 0) and
         g = constant(gap, 0) and
         b = constant(block, 0):
    if start(g) = start(b) and
       value(h) > 1:
    then fill(true, start(g) + 1, start(g) + value(h))


def rule-cross-small-gap-singleton:
    with h = singleton(hint) and
         g = arbitrary(gap):
    if value(h) > size(g):
    then fill(false, start(g), end(g))
```

```
def rule-cross-small-gap-min:
    with h = arbitrary(hint) and
         g = arbitrary(gap):
    if minimal(value(h)) and
       value(h) > size(g):
    then fill(false, start(g) + 0, start(g) + size(g))

def rule-cross-small-gap-left:
    with h = constant(hint, 0) and
         g = constant(gap, 0):
    if value(h) > size(g):
    then fill(false, start(g), end(g))

def rule-punctuate-max:
    with h = arbitrary(hint) and
         b = arbitrary(block):
    if maximal(value(h)) and
       value(h) = size(b) and
       start(b) > 0:
    then fill(false, start(b) + -1, start(b) + 0)

def rule-punctuate-hint0:
    with h = constant(hint, 0) and
         b = constant(block, 0):
    if lec(h) = start(b) and
       size(b) > 1:
    then fill(false, 1, size(b))
```

```
def rule-mercury:
    with h = constant(hint, 0) and
         b = constant(block, 0):
    if end(b) > 0 + value(h) and
       value(h) >= (start(b) - 0):
    then fill(false, 0, (end(b) - value(h)))

def rule-force-max:
    with h = arbitrary(hint) and
         g = arbitrary(gap):
    if (g is unique where size(g) >= value(h)) and
       value(h) + value(h) > size(g):
    then fill(true,
              start(g) + size(g) - value(h),
              start(g) + value(h))
```

Chapter 7

A CASE STUDY ON DESIGNING PROGRAMMING LANGUAGES FOR SYNTHESIS APPLICATIONS

The RuleSy framework relies heavily on the choice of input DSL. This chapter is concerned with the process of how that DSL is designed. We can view this problem through two different lenses: the practical question of whether end-users could successfully design DSLs for use in RuleSy, and the broader question of how we might go about automating the DSL design process itself.

The first is the practical consideration of whether it is reasonable to expect someone to be able to successfully apply RuleSy (or something like it) to other domains. If we are to suggest RuleSy might help in the domain model design process, we must understand whether others can use it. A great deal of effort and time was spent by the authors developing the languages used in the implementations of RuleSy for algebra and Nonograms. How well could others do this task? What are the ways we might make this task easier? This is a general challenge for all program synthesis systems: many other synthesis applications such as the PROSE framework [104] or DeepCoder [10] rely on application-specific DSLs to be successful. Very little has been written about the process of effective DSL design for synthesis applications; papers present the final version of such DSLs without discussing the extensive iterative development process that was likely behind them. What are the challenges and barriers people face in DSL design for synthesis? How might we develop methods and tools to make this process more accessible?

The second lens is a more abstract one. A recurring theme in this thesis has been the (semi) automation of design tasks: automatically generating problems, generating progressions, generating domain models. Choosing the domain-specific language to use with

RuleSy (or many other synthesis applications) is, itself, a design task. How we might begin automating that task? The first step in answering such a question is to understand the existing design process. What are the biggest challenges? What are the effective approaches?

In this chapter, we present a formative case study on DSL design for the Nonograms RuleSy instantiation, aiming to understand the challenges and barriers to successful DSL design for RuleSy. This study is done in the context of a tool intended to aid in the DSL design process. Eventually, we want to build tools and methods to make this process approachable and easy to do. In this chapter, we focus first on questions of what people do when asked to solve this task. Can people successfully recreate the results presented in the previous chapter? What challenges and barriers do people encounter? When given a tool that aims to help with this process, which parts of the tool do people use, and how does that shape their process? We begin to answer these questions with a formative user evaluation in which participants attempted to create the Nonograms DSL for use in RuleSy. By necessity, this case study used a simplified subtask of the full language design process, but the results both (a) suggest that RuleSy is understandable to end-users and they can have some success in (albeit contrived) usage of RuleSy, and (b) point us toward useful future research questions in how to make the DSL design process easier.

7.1 Background

DSL design is a critical component of many synthesis-based systems and applications [52]. While some applications have a natural predefined language (e.g., superoptimization necessarily targets the language it is optimizing), many others must choose an application-specific language. This is fundamentally a design task: while parts of the DSL may be obviously determined by technical constraints, other parts are subjective. However, these choices are critically important for the application's success.

For example, in the applications presented in this thesis, the task of choosing the DSL is literally one of selecting the problem domain concepts on which rules are defined. What these concepts should be and which are worthy of inclusion is a design problem that must

be tackled by a domain expert. As another example, consider the PROSE framework [104], used for programming-by-example applications in, e.g., Microsoft Excel macros. Choosing particular constructs for the DSL amounts to deciding which kinds of programs can be expressed concisely, or at all. What programs can be concisely expressed in the DSL has a large impact on which programs the system synthesizes and suggests to users. Both program synthesis researchers and software engineers building industrial synthesis applications must be able to effectively design these DSLs.

Despite the challenging nature of this design problem and how crucial it is to the success of (some) synthesis applications, very little has been written in the academic literature on the DSL design process. Anecdotally, the development of the languages for RuleSy took multiple years and many rounds of iteration. While we want to tackle the research problem of how to make this process easier, we must first address the problem of understanding what that process is and which parts are hard. Below, we document parts the language design process and methods used by the authors in the creation of RuleSy.

The Design Process for the RuleSy *Languages*

To answer our research questions, we built a DSL editing tool for RuleSy to support a limited DSL design task that could feasibly be completed in a couple hours. The structure of this task and the affordances of the tool were based on the authors' experience developing the original DSLs for RuleSy (detailed in Sections 5.3 and 6.4). We describe this process here, both to motivate our task/tool design and to document parts of the design process used in the development of RuleSy.

Methods for Feedback. While the evaluation metrics described in Sections 5.7 and 6.7 were used to compare the quality of output from RuleSy, global measures such as coverage on a test set or counts of rules synthesized do not lend themselves to figuring out how that output might be improved or pinpointing particular issues in the current design. For feedback, we relied heavily on investigation of individual input/output examples. For instance,

when developing the DSL for Nonograms, the first thing we typically checked to evaluate a new variant of the DSL was how well it performed in synthesizing the control rules (Section 6.11). That is, for each control rule, we would manually write (usually informally) how this rule would be expressed in the current DSL, ran RuleSy on a single training example that we would expect to result in that rule, and compare the two. New features were frequently chosen specifically because they allowed a particular control rule to be expressible. These particular examples were gathered from real-world sources such as textbooks or online tutorials.

Debugging Synthesis Failure. When RuleSy fails to synthesize a rule for a particular specification, such an outcome is typically caused by one of two issues: a problem with the DSL (e.g., no sound program can be expressed that meets the specification), or a problem with the synthesis algorithm (e.g., bugs in the implementation or practical limits such as timeouts). In order to distinguish between these, our typical approach was to hand-write a program that we believed meets the specification. By adding constraints to force the system to use this program, we could pinpoint the mistake: if the program turned out to be unsound, there was a bug in our DSL design caused by a mismatch in our mental model of rules and the actual DSL. It is important to note that failure can be caused by *too* expressive of a DSL: if the program space is too large, the synthesis algorithm will fail to find a program before timing out even if one could be found.

7.2 System and Design Task

For this thesis, we ran a case study, asking users to design a Nonograms DSL for RuleSy. In practice, the DSL design process takes place over a long period of time and involves substantial engineering, making studying this process challenging. Thus, to make a case study feasible, we must model a reasonable facsimile of the DSL design process. This involves reducing the design space, removing the need for engineering, and building an interface to facilitate the task. In this section, we describe the task and the tool built for the task.

7.2.1 The DSL Design Task

The first way we shrink the design task to one feasible for a case study is by shrinking the language design space. Rather than have users design the entire language from scratch, the system provides a foundation on which the users will build. Beyond being necessary for making the task easier to complete in limited time, limiting the DSL design space is necessitated by technological constraints as well: RuleSy expects its DSLs to be structured in certain ways (detailed in Chapter 6) for soundness and performance. Furthermore, substantial DSL-specific engineering effort is required to make synthesis tractable. Fixing parts of the language allows us to ensure these conditions are met regardless of the participants' designs.

What parts of the language should be fixed, and which left for the user to design? Programs in the RuleSy DSL represent procedural knowledge for solving problems in the domain. This means the DSL itself in some sense captures the concepts of the domain. As discussed in previous chapters, this is primarily defined by the patterns: the elements on which the patterns match represent conceptual units of the problem on which rules are defined. Therefore, defining the language of patterns in the primary design task of DSL design, so that is what we have the participants design.

The remainder of the language (conditions, actions, and the overall structure) are fixed. This is a reasonable reduction (from the sense of language design) because these parts of the language are obviously determined. For example, actions are fixed by the problem domain. Conditions contain primarily basic elements such as boolean and integer arithmetic. Many parts of the language design are fixed by the structure of RuleSy, as described in previous chapters. Thus, while the original design process obviously involved iterating on all areas of the language design, focusing on designing only the patterns while keeping the overal structure fixed is somewhat reflective of the real-world process of applying RuleSy to new domains. The primary challenge in the DSL-design portion of applying RuleSy to any new domain will likely be choosing the patterns/concepts to use in the DSL.

The DSL Design Space

Designing the patterns for the Nonograms DSL entails adding new *pattern types*, as defined in Section 6.4. Specifically, this involves adding new pattern types for the syntax (Figure 6.7), and new corresponding cases to the *elements* and *method* semantic functions (Figure 6.9). The structure of patterns and the available matching functions remain the same. That is, all patterns represent ordered sequences of state features. To define a new pattern type, one needs to define both a mapping from states to a sequence of elements of that type (the *elements* semantic function) as well as methods to access various boolean and integer properties of those elements (the *method* semantic function).

Starting Language

Rather than starting users from a blank slate with no patterns, we prime the system with the basic hint and cell patterns already available. This is done for two reasons: first, hints and cells are the fundamental units of the Nonograms state, and thus is it reasonable to assume that any designer would (at some point and probably quite early) include such features in their DSL. Second, having a starting set of patterns enables us to have a starting set of rule programs. Because we need to quickly teach users about the task and the parts of the language being provided for them, such example programs are valuable for illustrating those concepts.

The methods provided with these starting concepts are the minimal ones necessary to define the state. In particular, the ***`hint`*** feature used in the implementation in Chapter 6 is more complex than the one provided for the case study. The geometric variations of the big hint rule (e.g., Figure 6.4) require methods not provided; only the arithmetic variation (Figure 6.11) can be expressed. Its semantics are shown in Figure 7.2.

Precomputing Results

Another barrier to studying the DSL design process is the iteration time. In practice, computing results for RuleSy takes hours or days. This makes studying multiple iterations in a short study session impossible. To overcome this, we used RuleSy to precompute results for a variety of language designs. Using both intuition from the development of the original framework and information gathered from preliminary usability studies of the DSL design tool, we selected a set of pre-defined patterns we expected users to explore and used this to offer precomputed results.

Precomputation of results is possible due to the way RuleSy finds rules. Let L_1 and L_2 be DSLs that are identical except that the patterns of L_1 are a subset of the patterns of L_2. Because RuleSy enumerates over patterns during specification mining, for any given mined specification, program synthesis is independent of the pattern language. RuleSy will explore a subset of specifications for L_1 compared to L_2, meaning the synthesized rules for L_1 will be a subset of those synthesized for L_2. Thus, given results from RuleSy, run on a DSL L, we can efficiently calculate results from a hypothetical run of RuleSy on a language that is a subset L' of L by filtering the results for those using features from L'.

During the study, if the user stayed within the space of precomputed results, the system could provide those results within seconds. If the user chose a feature outside of this space, however, the system would have to be run normally. The study was performed in multiple sessions (see Section 7.3.2), so such a computation could be run between sessions.

Other System Inputs

RuleSy requires several inputs beyond the DSL, such as a domain description and training examples. For the purposes of this case study, these other inputs were fixed. The domain description was already defined, so fixing it is natural. Fixing the training set and objective function is less desirable, but necessary for the sake of this case study. For instance, precomputing results is only possible with a fixed training and testing set. We selected a training

set of 8 input/output examples. Each of these training examples demonstrates one of the control rules (Section 6.11) from the original evaluation of RuleSy(though not all control rules were covered by these examples).

7.2.2 The DSL Design Tool

To support designing a DSL quickly in a user-study setting, we built a computer interface that allows for both modifying the DSL and analyzing the quality of the DSL. This interface displays several metrics that measure the quality of RuleSy's output (and thus the quality of the DSL) on the testing set. It further allows for the user to add/remove pattern types from the DSL and recompute RuleSy's output. We describe the interface in detail below. The elements and affordances of this interface were based on the authors' personal experience developing the RuleSy DSLs. One of the questions we aim to answer with this case study is which of the elements of the interface participants actually use or find helpful.

Metrics

For the purposes of the study, the DSL is evaluated against three metrics: coverage, complexity, and synthesis time. The first two have been discussed extensively in this thesis: they represent the trade-off between rules that allow for solving many problems and rules that are concise. The metrics are simplified proxies of those used when evaluating the systems in previous chapters, in order to make them simple to explain and efficient to evaluate in a live setting. Rather than telling participants how to balance the trade-off, we ask them to maximize coverage while keeping complexity and time low, allowing them to decide which are more important.

Coverage. Coverage is measured in a similar manner to the evaluation of RuleSy in Section 6.7. Using the same testing set as that evaluation, the tool first synthesizes rules using RuleSy on the training set. Then, the tool applies every rule to every state in the testing set, counting how many cells are filled in. The complexity score is the ratio of cells

filled in by the synthesized rules over the total possible number of cells to fill in (and thus ranges from 0 to 1). For the purposes of tool responsiveness, rules are evaluated purely on the starting states in the test set; we do not use state decomposition nor applying rules to a fixed point.

For a given pattern feature F, we show two feature-specific complexity measures. First, we show a ratio of coverage: the coverage score of rules only using F over the total coverage score. Second, we show a ratio of coverage for which F is necessary: the coverage score of rules using F, but not counting any cells filled in by rules that do not use F, all over the total coverage score. The former is a measure of raw coverage: how many cells do rules using F fill in? The latter is a measure of redundancy: how many cells are filled in *only* by rules using F?

Complexity. The complexity metric is a composite metric that tries to approximate the relative complexity of the best rules used in synthesis. This metric captures two ideas: given two rules of equal generality: more concise is better, and given a set of rules of equal conciseness, using fewer to cover the same set of states is better. The metric calculates a score based on the covered testing states. Every rule R is assigned a complexity cost $C(R)$ that is essentially the size of the program's syntax. Then, for every cell c covered by at least one synthesized rule, we sum up the cost of the cheapest rule that covers c, plus an additional (small) constant for every additional rule that covers c. For a given feature F, the per-feature complexity score is the complexity of the rules using F (as calculated above) over the complexity of all rules.

Synthesis Time. The final metric shown is the total processor-hours required for synthesis. This includes successes, failures, and timeouts. Notably, individual work items typically do not result in direct failure (UNSAT); successes (SAT) take much less time to find than failures, so the system typically hits the timeout before failing. This is also shown on a per-feature basis by showing the ratio of processor-hours spent synthesizing rules whose patterns

include a particular feature.

Interface

The software tool (shown in Figure 7.1) supports three primary actions: (1) modifying the DSL, (2) viewing various metrics of the output of running RULESY for the current DSL, and (3) verifying human-specified programs. Both actions 1 and 3 use a Wizard-of-Oz (WOZ) style interface. We first discuss the system model and then discuss each action in turn.

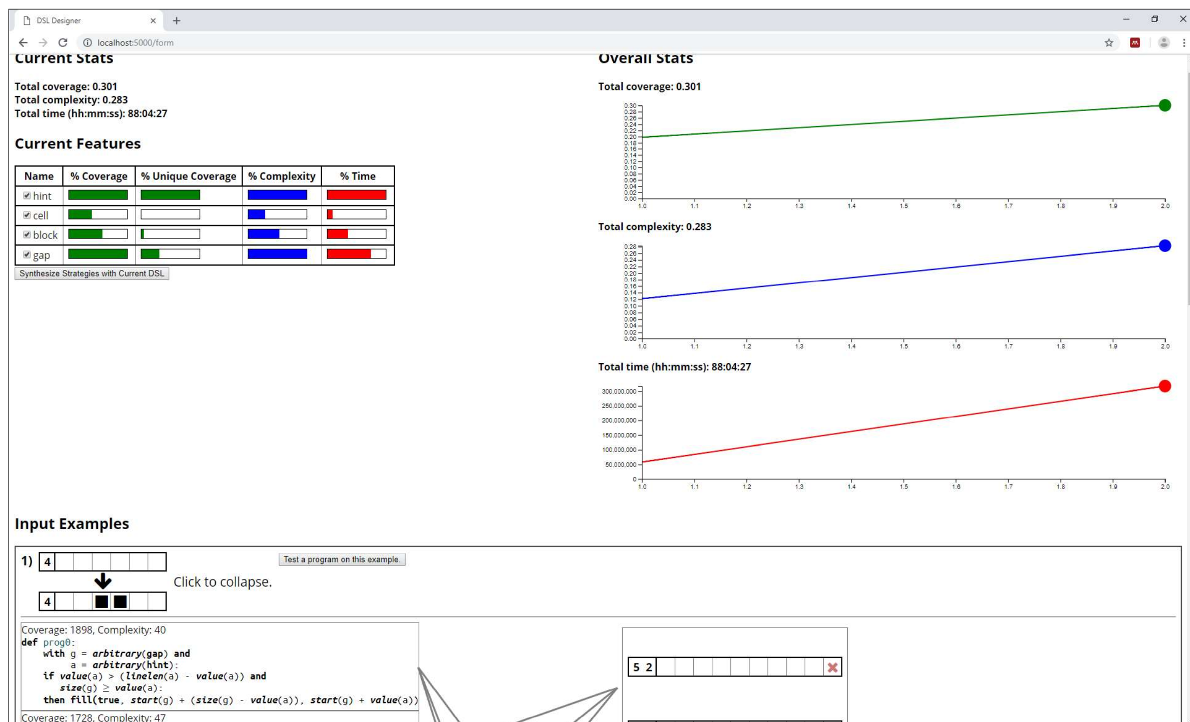


Figure 7.1: A screenshot of the DSL editing tool. The current pattern types are shown in the top left, along with the metrics. At the bottom of the screen, the training examples are shown, along with the best rules synthesized for each example (additional training examples are off the bottom of the screen).

Model. The tool models the DSL's design, representing the DSL as a set of pattern features. It models both currently active features as well as features that were used but are currently disabled. The tool tracks the history of this design as the user makes edits, used

for displaying how the metrics change over time. Participants could name features (including precomputed ones) whatever they wished; our predetermined names were not exposed.

Defining the DSL. The tool shows the current set of active and inactive features as a list of checkboxes. The user can toggle any of these features on or off. The list begins with only the basic hint and cell features. Adding new features to this list is done via a WOZ interface. The user describes the feature to the researcher using any of examples, pseudocode, mathematical notation, or informal description. To determine precisely what feature the user wishes to add, the researcher asks disambiguating questions via example inputs.

A WOZ-style interface was chosen due to lack of reasonable alternatives. Asking participants to write code themselves would require a substantial amount of training and practice with the system; informal methods of defining the DSL are necessary to allow users to define features not already in the tool. The alternative—using some in-software method to explore a pre-determined set of features—is no better. In early usability tests, users would reliably attempt to enumerate all of the features baked into the system by exploiting the interface. Such behavior is far removed from the real-world setting we hope to understand.

Due to the fact that results were precomputed, some leeway was given when participants defined features that very nearly matched a precomputed feature. If they desgined a feature with *elements* semantics that exactly matched a precomputed feature but whose *method* semantics differed only slightly, we made the precomputed feature available to them. By this we mean things such as methods having different names or missing methods that are combinations of others. For example, every participant designed a feature using segments (Definition 6.3), but often chose different names for the `start` and `size` functions or omitted the `end` function. We still provided them with the precomputed features since the `end` function is expressible by adding `start` and `size`.

Viewing Results. After selecting a language, the user could press a button to view RuleSy's output on that language. Metrics for the current DSL as displayed, both global and feature-specific. The tool also shows a timeline of the history of each of the global metrics from each iteration of the DSL. Below a list of the training examples was shown. For example, the tool displayed (a) whether any rule was found that covered that example, and (b) a list of the best programs (those with maximal coverage) for each feature combination covering that feature. Along with the programs, the system showed a set of testing examples to distinguish program behavior. The user could interactively click on the programs and see how their behaviors on the testing program differed.

Verifying Rules. To support the activity of interrogating unexpected synthesis failures, the tool allows (through WOZ) the user to verify programs. Given a program, the tool runs the verifier, showing both a counter-example to soundness (if one exists) and the program's behavior on the training inputs. So that users do not have to become proficient in the concrete syntax of the DSL, we use a WOZ-style interface in which the user describes to the researcher (via pseudocode or natural language) the program they wish to verify, and the researcher handles producing the correct syntax.

7.3 Methodology

To investigate how people would approach designing languages for RuleSy, we ran a case study using our DSL-design tool on the task described above. Sessions were approximately two hours per participant, and participants were people with some prior programming synthesis experience.

7.3.1 Recruitment

We recruited participants from researchers and graduate students familiar with program synthesis. This experience ranged from taking a class or using synthesis as part of work to being a synthesis researcher. We gave participants $50 in gift cards for their time.

7.3.2 Procedure

Case study sessions were run one participant at a time. Each participant had two sessions of 60 minutes each. Sessions began with obtaining consent and an introductory explanation of the research and study goals.

Participants were then introduced to the Nonograms problem domain by playing through a tutorial (specifically, "Tutorial 1" of "Playing Picross") of the *Picross S2* game for the Nintendo Switch.[1] During this tutorial, the researcher interjected to draw attention to the strategies the tutorial was presenting and encouraged participants to think about such strategies and how they would be defined.

Following the introduction to Nonograms, the researcher described the task in detail. The structure of the RuleSy system was described in basic detail. Two example rule programs (one of which is shown in Figure 6.11) were presented in complete detail: participants were provided with a handout showing the concrete syntax of the these programs. The researcher then described the parts of the RuleSy DSL that were fixed. Finally, the participant was instructed in how to define new parts of the DSL, using the starting "hint" and "cell" features as illustrative examples.

After being instructed on the task, participants were introduced to the software tool. The researcher described each part of the interface and the metrics and information presented. Participants were informed about the metrics on which their output would be measured. This procedure from start up to this point typically took approximately 40 minutes.

After this, participants were told to solve the task using the software tool while thinking aloud. They were permitted to ask questions of the researcher at any point. We collected data of both the tool usage and notes from the participants' think-alouds. Participants were additionally asked a handful of questions at the conclusion of the final session relating to their previous experience, which parts of the interface they found useful, and which parts of the task they found challenging.

---

[1] This game is from the same series from which we drew example rules and problems in Chapter 6.

7.3.3 Data Analysis

Our analysis examined the DSLs designed by participants and notes from the participants thinking aloud while performing the task. Due to the simplified and constrained nature of the task, limited number of participants, and potential interference from frequently interacting with the researcher due to the Wizard-of-Oz style interface, we limit ourselves to broad descriptive observations rather than detailed statistical analysis. Beyond a thesis-specific question of whether participants could successfully develop languages for RuleSy, the purpose of the study was to gain a better understanding of future directions for research. We do not have expectations of how the observations would generalize to other settings but are drawing conclusions about what things might be worthwhile to investigate in other settings, nor do we expect to be able to draw useful explanatory observations from the data. In particular, we looked at: (1) what DSL features participants designed and how they differed from the original RuleSy implementation, (2) whether participants were able to understand the task and make progress on it, (3) which parts of the interface participants claimed were useful for the task, (4) which parts of the task participants claimed were challenging, and (5) broad observations about how participants approached the task.

7.4 Results

We recruited 4 participants in the manner described. When asked about their experience with program synthesis, two had taken classes on program synthesis or formal methods, one had some experience using synthesis for software engineering, and one was a full-time synthesis researcher. We will refer to them as P1 through P4 when discussing results.

7.4.1 Designed Languages

Table 7.1 lists all of the features designed by participants, with the original features shown for comparison. None of the participants designed the same DSL as our original one presented in Chapter 6, though every feature from our DSL was also designed by at least one participant.

The semantics for these features is shown in Figure 7.2. All 4 participants managed to design DSLs in which a program was synthesized for each of the 8 training examples.

Feature	Original	P1	P2	P3	P4
`hint`	✗				✗
`block`	✗	✗	✗	✗	
`gap`	✗		✗		
`segment-e`		✗	✗		
`seg-e/x-border`		✗	✗	✗	✗
`segments`		✗			
`hint/sum`			✗		
`hint/pairs`			✗		
`block/max-sat`					✗

Table 7.1: Table listing the various features that each participant designed, not including the two (`hint/basic` and `cell`) given at the start of the task. The original features used in the implementation from Chapter 6 are shown for comparison.

7.4.2 Question Responses

After the task, participants were asked which parts of the user interface they relied on or found useful for the task, and which parts of the task they found challenging.

Regarding parts of the interface that were useful, all 4 participants reported relying on the training examples. However, P1, P2, and P3 specifically noted that they only used these examples in a binary way: if an example was not covered at all, they chose a feature to cover it. After a training example was covered, they were not able to get useful information to improve the metrics, especially coverage. In particular, they said that the example programs and example inputs shown with each training example did not help them. P2 noted that it was difficult to know what to do once an example was covered, and P1 said they were unable to figure out, looking at the synthesized programs, whether features were good as measured by coverage or not. All participants focused on the metrics, particular when choosing features to remove. Regarding feature-specific metrics, P1 and P2 reported using

$$
\begin{array}{lcl}
\mathit{elements}[\![\texttt{hint/basic}]\!]\langle H, V\rangle & = & \langle H_i, V\rangle \\
\mathit{method}[\![\texttt{bl_value}]\!]\langle h, V\rangle & = & h \\
\mathit{method}[\![\texttt{bl_linelen}]\!]\langle h, V\rangle & = & |V| \\
\mathit{elements}[\![\texttt{segment-e}]\!]s & = & \mathit{segments}(s, \{\textbf{empty}\}) \\
\mathit{elements}[\![\texttt{seg-e/x-border}]\!]s & = & \textbf{empty} \text{ segments of } s \text{ not bordered by } \textbf{true} \text{ cells} \\
\mathit{elements}[\![\texttt{segments}]\!]s & = & \text{all single-valued segments of } s \text{ paired with their value} \\
\mathit{method}[\![\texttt{seg_true?}]\!]\langle \mathit{seg}, v\rangle & = & v = \textbf{true} \\
\mathit{method}[\![\texttt{seg_false?}]\!]\langle \mathit{seg}, v\rangle & = & v = \textbf{false} \\
\mathit{method}[\![\texttt{seg_empty?}]\!]\langle \mathit{seg}, v\rangle & = & v = \textbf{empty} \\
\mathit{elements}[\![\texttt{hint/sum}]\!]s & = & \text{same semantics as hint (plus the below method)} \\
\mathit{method}[\![\texttt{numhints}]\!]\langle H, V, \mathit{idx}\rangle & = & |H| \\
\mathit{elements}[\![\texttt{hint/pairs}]\!]s & = & \text{consecutive pairs of hints} \\
\mathit{elements}[\![\texttt{block/max-sat}]\!]s & = & [\arg\max_{s\in B} \mathit{size}(s)] \textbf{ if } |B| > 0, \\
 & & \max_{s\in B} \mathit{size}(s) = \text{max hint value in } s, \text{ and} \\
 & & \text{max segment size in } B \text{ is unique } \textbf{else } [] \\
 & & \ \textbf{where } \ B = \mathit{segments}(s, \{\textbf{true}\})
\end{array}
$$

Figure 7.2: Semantics for patterns added by participants, grouped by pattern type. Some notation is described in Section 4.2, and some of these semantics reference semantics from the DSL defined in Section 6.4. For instance, all of the segment types (e.g., `segment-e`) use the same methods as the standard segment types (e.g., `block`).

the “unique coverage” metric much more than the others, though P2 asked why the process of using these metrics to remove features could not be done automatically by the system.

Regarding challenges, both P2 and P4 said that the task was initially difficult to adapt to, with P2 specifically saying, “This is not a normal way of thinking." They were more inclined to write rules directly. P3 also mentioned that it was difficult to figure out where to start on the task. However, both P2 and P4 also noted that after working on the task, they were able to understand what was going on and what they were supposed to do. P1 said that they did not have a hard time coming up with many new features, but did have a difficult time whittling them back down to reduce complexity, particularly when features were redundant in some way. P2 noted that it was difficult to think beyond the training examples; during the task, both P2 and P4 tried to add additional training examples to the tool, which was not possible in this study due to results being precomputed. P2 also said they would likely have a very hard time in a real-world setting where the iteration time is on the order of hours of days: it would be difficult for them to remember what features they had designed or what they meant when coming back to them. Indeed, between two session that were a couple days apart, P2 misremembered one of their designed features.

7.5 Discussion

That every participant was able to design a language that at the very least covered the training set suggests that participants were able to understand RuleSy and the task. The languages designed differed substantially from each other and the original. Notably, one of the features, `seg-e/x-border`, performed better than the original’s `gap` on the complexity metric while only being slightly worse on the coverage metric, suggesting that participants in some ways eclipsed the quality of the original language. While participants noted that working with RuleSy was unnatural at first, all of them were able to succeed after working on the task. Overall, these results provide existential evidence that users could successfully design (at least part of) the input languages for RuleSy or similar systems, at various levels of expertise with program synthesis.

While it is difficult to draw generalizable conclusions from the data, we can, from observations of the participants as well as their answers to the questions, determine hypotheses to be investigated in future studies, both for RuleSy specifically and program synthesis development more generally. Every participant focused initially on ensuring the initial training examples were satisfied, and multiple tried to add new training examples or used them to help design new features. After satisfying all of the training examples, participants struggled figuring out what to do next or how to improve the metrics. Because the user interface of the DSL design tool strongly afforded focusing on the training examples, it is difficult to disentangle the influence of the UI and particular task from the language design approaches of the participants. For instance, though participants reported not getting much utility out of showing the synthesized programs, that may be due to the interface or to the fact that the study task forced them to learn a new concrete syntax very quickly. Nevertheless, future research could investigate how concrete input/output examples could be used to make this process easier. Particularly because the participants only used training examples as binary indicators of success rather than deeply investigating, it seems that more training examples could be beneficial for prompting designers to create new features or modify existing features that improve coverage. Would some kind of training example management interface aid designers? Are there domain-agnostic heuristics we could add to the tool to automatically show failing testing examples to the designer that could plausibly allow them to make progress? Would showing results via examples rather than concrete programs help?

When deciding which features to remove from the language to lower complexity, participants reported primarily relying on the "unique coverage" metric. Features with little to no unique coverage were considered for removal. This makes sense, because the unique coverage metric directly measure redundancy, which makes the complexity metric rise. This suggests that the unique coverage metric may be more useful than the others and is worth further investigation. As P2 noted, however, these decisions probably could be made automatically by the system. A computer obviously cannot easily create new features from nothing, but it *can* mechanically explore subsets of existing features to choose the best according to the

metrics. Future Research might explore such automation in DSL design tools.

7.6 Limitations

Our research was a small case study with 4 participants. The biggest limitation are the contrivances necessary to make a task doable in a couple hours. Because our primary goal was to demonstrate that users can develop reasonable DSLs for RuleSy, we had participants work on that particular task. However, studying and observing the process of DSL development for program synthesis on a researcher-given task is challenging without the kinds of sacrifices we needed to make. In order to gain a better understanding of general program synthesis language development, alternative methodologies should be pursued, such as interviewing practitioners or studying existing tasks in real-world contexts.

The limitations of the tool necessary to allow participants to work in a short time period prevented them from taking actions that they were inclined to do. For example, multiple participants wanted to add other training examples to explore particular rules and features they had in mind. This is a reasonable approach, but participants were unable to do this due to the fact that we needed to precompute all results.

The study gave participants a user interface for working with the DSL. This was a necessary part of making the task approachable enough to be feasibly done in a short user study, but the tools affordances likely influenced participant behavior. It is hard to untangle whether the challenges encountered were caused by the interface or by the nature of the task.

Lastly, this study was on a specific synthesis application. While we think some of the results suggest hypotheses for general future work in making program synthesis more approachable, it is possible that little of the observed behavior would be seen in applications beyond the one studied.

7.7 Related Work

Though programming languages and formal methods systems do not typically have user evaluations, researchers have investigated human factors questions for programming languages.

In the broader field of programming language design, decades of research exist in understanding the usability of programming languages, for professionals and students [96]. Specifically within the formal methods community, researchers have recently begun to investigate the usability of formal methods tools [36, 18], which traditionally have been evaluated only for their guarantees and performance.

Traditional synthesis frameworks require a significant amount of bespoke effort to create [48], leading to recent efforts in creating frameworks to reduce the authorial burden in creating synthesis tools. SKETCH is a tool that takes, as input, a partial program in a C-like language, with holes for certain subexpressions to be filled in through synthesis [124]. ROSETTE [133] is a framework for synthesis and verification of DSL programs that uses symbolic execution to automatically convert queries to SMT. These DSLs are implemented in a subset of Racket, a variant of Lisp. Smten [135] is another framework that automatically creates SMT, except Smten uses a Haskell-inspired custom input language. Our framework has so far used ROSETTE for synthesis. Many learning domains (especially those in mathematics) can be expressed in SMT theories, and ROSETTE's affordance of describing the DSL semantics in Racket allows for freedom to quickly define DSLs for synthesis.

While syntactic restrictions are a good way to inject domain-specific knowledge to make synthesis tractable, applications often still require hand-turning the synthesis algorithm with domain-specific insights to be sufficiently performant. Recent efforts have investigated methods to separating this domain-specific insight from the search algorithm, allowing program synthesis frameworks to be instantiated for new domains without as much effort or synthesis expertise as previous approaches. PROSE (a.k.a. FlashMeta) [104] is a synthesis framework that allows for DSL designer to inject domain-specific knowledge in what the authors call *witness functions*, which (roughly) describe the inverse semantics of a DSL's operators. These witness functions are used to deductively decompose the specification into smaller subproblems, on which a generic enumerative search synthesis algorithm is used to find programs. SYNAPSE [19] is an optimal synthesis framework (built on ROSETTE) for specifying a high-level, domain-specific search strategy using a series of sketches (as opposed to the

typical single sketch). This enables syntactic constraints not otherwise expressible, such as restricting programs to single static assignment (SSA) form. This ability to inject domain knowledge without having to modify the synthesis algorithm is highly desirable. Our framework is designed in this way. However, both our framework and these frameworks require an appropriately designed DSL to work effectively, which itself involves critical domain insights, synthesis expertise, and likely much iteration [50].

7.8 Conclusion

This chapter presented a small case study in which participants designed DSLs for use in RuleSy. Our study focused on (1) understanding whether users could successfully design languages for RuleSy, and (2) investigating the challenges of designing DSLs for program synthesis to motivate future research. Participants of various backgrounds in program synthesis were able to understand RuleSy and design DSLs of comparable quality to the one developed by the authors. The limited nature of the study and contrived nature of the task prevent us from drawing firm or conclusions about the language design process, but the cast study indicates that focusing on individual training examples is an approach worth investigating further. Future work should investigate this process with other methodologies (e.g., interviews with practitioners) to get a broader understanding of the process and its challenges. However, the results show existential evidence that people are able to understand RuleSy enough to design effective DSLs to be used with RuleSy.

Chapter 8

CONCLUSION

This thesis has presented RuleSy, a framework for automatically synthesizing domain models for educational domains by framing the program as one of program synthesis. Unlike prior frameworks for learning domain models which rely on examples, RuleSy is the first framework to automatically learn strategies using only a domain description to guarantee sound rules. Thanks to its novel algorithms, RuleSy is able to synthesize sound and useful rules that generalize beyond given training examples. We instantiated RuleSy for multiple domains, and our evaluation has shown that RuleSy is capable of generating rules of comparable quality to human-created rules for those domains. The framework aims to ease the process of effective domain model design. This is motivated, in addition to cited work, by two novel applications presented in this thesis. The first proposed a novel mixed-initiative interface for designing progressions in games, and the second proposed and evaluated an algorithm for fully generated progressions. A core feature of RuleSy is that its user provides the framework with a domain-specific language in which to represent rules, allowing a domain expert to bias RuleSy and the rules it synthesizes. To evaluate the feasibility of using RuleSy and specifically designing input DSLs, we ran a small case study in which participants were successful in creating good DSLs for a modified version of RuleSy. This thesis represents a step towards technology that enables designers to quickly and easily craft formal models of human expertise and solving strategies, enabling a range of applications in education, game design, and beyond.

Evaluation of RuleSy Output. Evaluating the quality of any educational content is challenging, both methodologically and logistically. In this thesis, our evaluation of RuleSy's

output was limited to broad comparisons to existing materials such as textbooks or document puzzle game strategies. The structure of the rules learned by framework was based on current understanding on human cognition in similar domains. However, before using RuleSy in a practical setting, a much more rigorous evaluation of its output is warranted, particularly with respect to educational utility. For example, one might evaluating rules synthesized by RuleSy on students in a classroom setting to investigate, for instance, whether they can be understood or whether they are useful for problem solving. Related questions include further exploration of tools and applications that can be built on top of the domain models generated by framework.

Utility of RuleSy. This thesis proposes a particular solution for making the domain model design process easier and more effective. In particular, we argue that synthesis can benefit designers in ways such as ensuring sound rules are generated or exploring parts of the design space that a human might miss. While these are real benefits for these design problems and while our framework design was motivated by documented techniques for domain model development [67], further investigation is required. The practices for domain model development across the various industries are not well documented; investigations such as interviews with practicing designers should be undertaken to ensure tools are solving real-world problems. Specifically, it is undeniable that this system offloads one challenging design problem from a person by giving them a qualitatively different but still challenging design problem of designing the inputs for RuleSy. We performed a case study to evaluate the feasibility of (and uncover the challenges involved in) using RuleSy, particularly in design the DSL, but future work should perform longer-term studies on less simplified tasks to get a better understanding of the potential benefits of tools like RuleSy. The answers to these questions likely varies wildly between industries and applications; beyond education, one could explore more deeply in applications such as game development and design.

Broader Applications to Programming Languages. There is additional future work in research questions more generally applicable to the program synthesis community. For example, in Chapter 5, we suggested that tree-rewrite rules may have applications for traditional programming languages application such as compilers. Many compilers use rewrite rules, either for optimization or as a more fundamental and necessary part of the compilation process. As another example, proof assistants like Coq also function through use of rewrite rules; in this case, proof macros for trying to apply various proof strategies. Not only is it challenging to write sound rewrite rules by hand, but there is (as with educational domains) a challenging design problem of deciding which combination of rules should be included and in what order they should be used. Future work might investigate both how to build such rewrite rules more easily or building more effective systems than can be done by hand. Can they be composed in the way we compose algebra axioms for create more useful rules? In some ways, such applications of rule synthesis are easier to investigate than the educational applications in that there are reasonable objective criteria on which to evaluate output (e.g., do the synthesized rules enable compilers to create more optimized programs more efficiently than without?).

Designing DSLs for Program Synthesis. The research question posed in Chapter 7 has only begun to be explored: how can people develop effective languages for program synthesis applications? Answering this question is crucial to the practical success of program synthesis, especially in applications where the language design is not obvious, such as programming-by-example. Our study uncovered some of the challenges encountered by language designers in a simplified task using a single synthesis tool, so a lot more investigative work remains to be done. One could, for instance, interview practicing developers of synthesis systems or run user studies on different applications. Once we have a better understanding on how people approach this design problem and which parts are challenging, the next questions are in how to build tools and methods for effective design. Our case study uncovered some potential such designs. For example, participants relied heavily on concrete training examples

to drive their language design, and struggled to improve the language once the individual input/output examples were satisfied. How can we build tools centered around concrete examples to help users understand their language designs? Could tools automatically suggest plausible additional input/output examples that are not covered to drive the human language designer's attention in a fruitful way?

Looking Forward. Each of these research directions, when realized, empower users to create things not otherwise possible. Domain models power existing important educational technology such as computer tutors, so making it easier to design them effectively improves existing applications. Beyond this, automatic generation of domain models enables exciting unrealized applications such as advanced game design tools which are otherwise not feasible due to the prohibitive cost of designing domain models by hand. Effective domain model generation applies beyond technology: it could be used for the design of classroom or textbook curricula, impacting education for students everywhere. Applying these ideas beyond education, there are many applications without programming languages that could benefit greatly from automated rule learning, from easier-to-construct compilers to better automated proof assistants. Finally, future work in methods and tools for DSL design enables more people to more easily create program synthesis systems, helping advance research and realize applications. The common thread is empowering users: empowering designers and domain experts, empowering builders of programming systems, and empowering novices of all domains to be able to create artifacts not otherwise within reach. This thesis represents a step towards realizing these goals.

BIBLIOGRAPHY

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[2] Chris Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Synthesis of geometry proof problems. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 245–252. AAAI Press, 2014.

[3] Erik Andersen, Sumit Gulwani, and Zoran Popović. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI*, 2013.

[4] Erik Andersen, Yun-En Liu, Rich Snider, Roy Szeto, and Zoran Popović. Placing a value on aesthetics in online casual games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1275–1278. ACM, 2011.

[5] Erik Andersen, Eleanor O'Rourke, Yun-En Liu, Rich Snider, Jeff Lowdermilk, David Truong, Seth Cooper, and Zoran Popovic. The impact of tutorials on games of varying complexity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 59–68. ACM, 2012.

[6] 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.

[7] John R Anderson and Christian Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.

[8] Anna Anthropy and Naomi Clark. *A Game Design Vocabulary*. Addison-Wesley, 2014.

[9] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, December 2001.

[10] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations (ICLR)*, 2017.

[11] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.

[12] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. Synthesis of programs in computational logic. In *PROGRAM DEVELOPMENT IN COMPUTATIONAL LOGIC*, pages 30–65. Springer, 2004.

[13] K Joost Batenburg and Walter A Kosters. On the difficulty of nonograms. *ICGA Journal*, 35(4):195–205, 2012.

[14] Christian Bauckhage, Kristian Kersting, Rafet Sifa, Christian Thurau, Anders Drachen, and Alessandro Canossa. How players lose interest in playing a game: An empirical study based on distributions of total playing times. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 139–146. IEEE, 2012.

[15] Joseph Beck, Mia Stern, and Beverly Park Woolf. Using the student model to control problem difficulty. *Courses and Lectures-International Centre for Mechanical Sciences*, pages 277–288, 1997.

[16] C. Bell and D.S. McNamara. Integrating iSTART into a high school curriculum. In *Proceedings of the 29th Annual Meeting of the Cognitive Science Society*, Austin, TX, USA, 2007. Cognitive Science Society.

[17] Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, Dec 1998.

[18] James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. *Proc. ACM Program. Lang.*, 2(OOPSLA):149:1–149:26, October 2018.

[19] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 775–788, New York, NY, USA, 2016. ACM.

[20] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[21] John D Bransford, Ann L Brown, Rodney R Cocking, et al. *How people learn*. Washington, DC: National Academy Press, 2000.

[22] John L Bresina, Ari K Jónsson, Paul H Morris, and Kanna Rajan. Mixed-initiative planning in mapgen: Capabilities and shortcomings. In *Proceedings of the ICAPS-05 Workshop on Mixed-initiative Planning and Scheduling, Monterey, CA*, pages 54–61. Citeseer, 2005.

[23] Cameron Browne. Deductive search for logic puzzles. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013.

[24] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:1–16, 2010.

[25] Eric Butler, Erik Andersen, Adam M. Smith, Sumit Gulwani, and Zoran Popović. Automatic game progression design through analysis of solution features. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 2407–2416, New York, NY, USA, 2015. ACM.

[26] Eric Butler and Rahul Banerjee. Visualizing progressions for education and game design. 2014, Unpublished.

[27] Eric Butler, Adam M. Smith, Yun-En Liu, and Zoran Popović. 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.

[28] Eric Butler, Emina Torlak, and Zoran Popović. A framework for parameterized design of rule systems applied to algebra. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems - Volume 9684*, ITS 2016, pages 320–326, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[29] Eric Butler, Emina Torlak, and Zoran Popović. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 10:1–10:10, New York, NY, USA, 2017. ACM.

[30] Eric Butler, Emina Torlak, and Zoran Popović. A framework for computer-aided design of educational domain models. In *Procedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2018.

[31] Randall I. Charles, Basia Hall, Dan Kennedy, Allan E. Bellman, Sadie Chavis Bragg, William G. Handlin, Stuart J. Murphy, and Grant Wiggins. *Algebra 1: Common Core*. Pearson Education, Inc., 2012.

[32] Jenova Chen. Flow in games (and everything else). *Communications of the ACM*, 50(4):31–34, 2007.

[33] Daniel Cook. The chemistry of game design. *Gamasutra*, July 2007.

[34] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[35] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper & Row Publishers, Inc., 1990.

[36] Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J. Dougherty. User studies of principled model finder output. In Alessandro Cimatti and Marjan Sirjani, editors, *Software Engineering and Formal Methods*, pages 168–184, Cham, 2017. Springer International Publishing.

[37] Jennifer Demski. This time it's personal: True student-centered learning has a lot of support from education leaders, but it can't really happen without all the right technology infrastructure to drive it. and the technology just may be ready to deliver on its promise. *The Journal (Technological Horizons In Education)*, 39(1):32, 2012.

[38] Nachum Dershowitz and Jean-Pierre Jouannaud. Chapter 6 - rewrite systems. In Jan van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 243–320. Elsevier, Amsterdam, 1990.

[39] Michel C Desmarais and Ryan SJ d Baker. A review of recent advances in learner and skill modeling in intelligent learning environments. *User Modeling and User-Adapted Interaction*, 22(1-2):9–38, 2012.

[40] Michel C Desmarais, Peyman Meshkinfam, and Michel Gagnon. Learned student models with item to item knowledge structures. *User Modeling and User-Adapted Interaction*, 16(5):403–434, 2006.

[41] Jean-Paul Doignon and Jean-Claude Falmagne. Spaces for the assessment of knowledge. *International journal of man-machine studies*, 23(2):175–196, 1985.

[42] Matthew W Easterday, Vincent Aleven, Richard Scheines, and Sharon M Carver. Using tutors to improve educational games. In *Artificial Intelligence in Education*, pages 63–71. Springer, 2011.

[43] Molly Q. Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. Automatic diagnosis of students' misconceptions in k-8 mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 264:1–264:12. ACM, 2018.

[44] 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.

[45] Mary L Gick. Problem-solving strategies. *Educational psychologist*, 21(1-2):99–120, 1986.

[46] Arthur C Graesser, Patrick Chipman, Brian C Haynes, and Andrew Olney. Autotutor: An intelligent tutoring system with mixed-initiative dialogue. *Education, IEEE Transactions on*, 48(4):612–618, 2005.

[47] Barbara J. Grosz, Luke Hunsberger, and Sarit Kraus. Planning and acting together. *AI Magazine*, 20:23–34, 1999.

[48] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10. ACM, 2010.

[49] Sumit Gulwani. Example-based learning in computer-aided stem education. *Communications of the ACM*, 57(8):70–80, 2014.

[50] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, October 2015.

[51] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. ACM, 2011.

[52] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[53] Guershon Harel and Larry Sowder. Toward comprehensive perspectives on the learning and teaching of proof. *Second handbook of research on mathematics teaching and learning*, 2:805–842, 2007.

[54] 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*, pages 79–88. ACM, 2013.

[55] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, UIST '06, pages 299–308. ACM, 2006.

[56] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 91–100. ACM, 2008.

[57] Lisa Anne Hendricks, Zeynep Akata, Marcus Rohrbach, Jeff Donahue, Bernt Schiele, and Trevor Darrell. Generating visual explanations. In *European Conference on Computer Vision*, pages 3–19. Springer, 2016.

[58] Katherine Isbister, Mary Flanagan, and Chelsea Hash. Designing games for learning: insights from conversations with designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2041–2044. ACM, 2010.

[59] G Tanner Jackson, Kyle B Dempsey, and Danielle S McNamara. Short and long term benefits of enjoyment and learning within a serious game. In *Artificial Intelligence in Education*, pages 139–146. Springer, 2011.

[60] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popović. Evaluating competitive game balance with restricted play. In *AIIDE*, 2012.

[61] Matthew P Jarvis, Goss Nuzzo-Jones, and Neil T Heffernan. Applying machine learning techniques to rule generation in intelligent tutoring systems. In *Intelligent Tutoring Systems*, pages 541–553. Springer, 2004.

[62] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: A model for dynamic level generation. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[63] Jupiter. Picross e6, 2015.

[64] Jupiter. Pokémon picross, 2015.

[65] Jupiter. Picross e7, 2016.

[66] Scott R. Klemmer, Anoop K. Sinha, Jack Chen, James A. Landay, Nadeem Aboobaker, and Annie Wang. Suede: a wizard of oz prototyping tool for speech user interfaces. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, UIST '00, pages 1–10. ACM, 2000.

[67] Kenneth R Koedinger, Emma Brunskill, Ryan SJd Baker, Elizabeth A McLaughlin, and John Stamper. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.

[68] K.R. Koedinger, A.T. Corbett, et al. Cognitive tutors: Technology bringing learning science to the classroom. *The Cambridge handbook of the learning sciences*, pages 61–78, 2006.

[69] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35 – 77, 1985.

[70] Joseph B Kruskal and Myron Wish. *Multidimensional scaling*. Sage, 1978.

[71] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1 – 64, 1987.

[72] Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, New York, NY, USA, 2016. ACM.

[73] James A. Landay. Silk: sketching interfaces like krazy. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 398–399. ACM, 1996.

[74] Pat Langley, John E Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, 2009.

[75] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In *Intelligent Tutoring Systems*, pages 306–311. Springer, 2014.

[76] Colleen Lee. DeduceIt: a tool for representing and evaluating student derivations. Stanford Digital Repository: http://purl.stanford.edu/bg823wn2892, 2012.

[77] Nan Li, William Cohen, Kenneth R Koedinger, and Noboru Matsuda. A machine learning approach for automatic student model discovery. In *Educational Data Mining 2011*, 2010.

[78] Nan Li, Abraham J Schreiber, WW Cohen, and KR Koedinger. Efficient complex skill acquisition through representation learning. *Advances in Cognitive Systems*, 2, 2012.

[79] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[80] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Procedings of the 8th International Conference on the Foundations of Digital Games*, pages 213–220, 2013.

[81] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. Denim: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '00, pages 510–517. ACM, 2000.

[82] 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*, pages 1979–1988. ACM, 2011.

[83] Chao-Lin Liu. A simulation-based experience in learning structures of bayesian networks to represent how students learn composite concepts. *International Journal of Artificial Intelligence in Education*, 18(3):237–285, 2008.

[84] Yun-En Liu, Christy Ballweber, Eleanor O'rourke, Eric Butler, Phonraphee Thummaphan, and Zoran Popović. Large-scale educational campaigns. *ACM Trans. Comput.-Hum. Interact.*, 22(2):8:1–8:24, March 2015.

[85] Derek Lomas, Kishan Patel, Jodi L Forlizzi, and Kenneth R Koedinger. Optimizing challenge in an educational game using large-scale design experiments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 89–98. ACM, 2013.

[86] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, 2008.

[87] Chris Martens. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[88] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126. IEEE Computer Society Press, 1987.

[89] Michael Mateas and Noah Wardrip-Fruin. Defining operational logics. In *DiGRA*, 2009.

[90] Noboru Matsuda, William W Cohen, and Kenneth R Koedinger. Applying programming by demonstration in an intelligent authoring tool for cognitive tutors. In *Aaai workshop on human comprehensible machine learning (technical report ws-05-04)*, pages 1–8, 2005.

[91] Antonija Mitrovic. An intelligent sql tutor on the web. *International Journal of Artificial Intelligence in Education*, 13(2-4):173–197, 2003.

[92] Gareth Moore. *The Essential Book of Hanjie and How to Solve It.* Atria Books, 2006.

[93] Leonardo Moura and Nikolaj Bjørner. *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, chapter Z3: An Efficient SMT Solver, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[94] Tom Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10, 1999.

[95] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The amulet environment: new models for effective user interface software development. *Software Engineering, IEEE Transactions on*, 23(6):347–365, 1997.

[96] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, September 2004.

[97] Mark J Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Artificial Intelligence in the Game Design Process*, 2011.

[98] Tetsuya Nishio. *Original O'Ekaki: Intelligent Designs from Its Creator*. Vertical, 2008.

[99] Derek C Oppen. Reasoning about recursively defined data structures. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 151–157. ACM, 1978.

[100] Eleanor O'Rourke, Erik Andersen, Sumit Gulwani, and Zoran Popović. A framework for automatically generating interactive instructional scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1545–1554, New York, NY, USA, 2015. ACM.

[101] John Orwant. Eggg: Automated programming for game generation. *IBM Systems Journal*, 39(3.4):782–794, 2000.

[102] Joseph Carter Osborn, April Grow, and Michael Mateas. Modular computational critics for games. In *AIIDE*, 2013.

[103] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[104] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Inter. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126. ACM, 2015.

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

[106] Sam Redding. Getting personal: The promise of personalized learning. *Handbook on innovations in learning*, pages 113–130, 2013.

[107] Charles Rich and CandaceL. Sidner. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3-4):315–350, 1998.

[108] Steven Ritter, John R Anderson, Kenneth R Koedinger, and Albert Corbett. Cognitive tutor: Applied research in mathematics education. *Psychonomic bulletin & review*, 14(2):249–255, 2007.

[109] Florian Schanda and Martin Brain. Diorama. `http://warzone2100.org.uk/`, April 2013.

[110] Tom Schaul. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence in Games*, 2013.

[111] Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3):237–248, 2011.

[112] Zhangzhang Si and Song-Chun Zhu. Learning and-or templates for object recognition and detection. *IEEE transactions on pattern analysis and machine intelligence*, 35(9):2189–2205, 2013.

[113] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[114] Derek Sleeman, Pat Langley, and Tom M Mitchell. Learning from solution paths: An approach to the credit assignment problem. *AI Magazine*, 3(2):48, 1982.

[115] Ruben Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 2:1–2:8. ACM, 2010.

[116] Adam Smith, Eric Butler, and Zoran Popović. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Procedings of the 8th International Conference on the Foundations of Digital Games*, 2013.

[117] 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 *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 156–163. ACM, 2012.

[118] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[119] Adam M Smith, Mark J Nelson, and Michael Mateas. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 91–98. IEEE, 2010.

[120] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *In procedings of the 4th International Conference on the Foundations of Digital Games*, 2009.

[121] Gillian. Smith, James Whitehead, and Michael Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):201–215, 2011.

[122] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. Launchpad: A rhythm-based level generator for 2-d platformers. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(1):1–16, 2011.

[123] Stephen Smith, Ora Lassila, and Marcel Becker. Configurable, mixed-initiative systems for planning and scheduling. In *Advanced Planning*, pages 235–241. AAAI Press, 1996.

[124] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2006.

[125] Mia K Stern and Beverly Park Woolf. Curriculum sequencing in a web-based tutor. In *Intelligent Tutoring Systems*, pages 574–583. Springer, 1998.

[126] Andrew C Stuart. *The Logic of Sudoku*. Michael Mepham Publishing, 2007.

[127] Andrew C Stuart. Sudoku creation and grading, January 2012. [Online, accessed 8 Mar 2017].

[128] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. *Acm Sigplan Notices*, 45(1):199–210, 2010.

[129] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[130] Michael Terry and Elizabeth D. Mynatt. Side views: persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, UIST '02, pages 71–80. ACM, 2002.

[131] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Constructing coding duels in Pex4Fun and Code Hunt. In *ISSTA*, pages 445–448. ACM, 2014.

[132] J. Togelius, G.N. Yannakakis, K.O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[133] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[134] Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. *Technical Report*, 1996.

[135] Richard Uhler and Nirav Dave. *Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries*, pages 678–683. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[136] Kurt VanLehn. *Mind bugs: The origins of procedural misconceptions*. MIT press, 1990.

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

[138] Georgios N Yannakakis and Julian Togelius. Experience-driven procedural content generation. *Affective Computing, IEEE Transactions on*, 2(3):147–161, 2011.

[139] Alexander Zook, Brent Harrison, and Mark O Riedl. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th Conference on the Foundations of Digital Games*, 2015.

[140] Alexander Zook, Stephen Lee-Urban, Mark O. Riedl, Heather K. Holden, Robert A. Sottilare, and Keith W. Brawner. Automated scenario generation: toward tailored and optimized military training in virtual environments. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 164–171. ACM, 2012.