Synthesizing Optimal Domain Models for Educational Applications

ERIC BUTLER, University of Washington, edbutler@cs.washington.edu
EMINA TORLAK, University of Washington, emina@cs.washington.edu
ZORAN POPOVIĆ, University of Washington, zoran@cs.washington.edu

Many applications in educational technology, from student modeling to problem generation, are built on a formal model of the operational knowledge for a given domain. These *domain models* consist of 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. In practice, domain models are handcrafted at great expense, so applications rely on a single generic model. But many models can capture the operational knowledge for a domain, and they differ in how well they meet specific educational objectives (such as maximizing problem-solving efficiency for advanced users or minimizing cognitive load for novices). Rapid creation of custom domain models is thus a key challenge in the development of personalized educational tools that adapt to their user's needs.

This paper presents RuleSy, a new framework for computer-aided authoring of optimal domain models for educational applications. RuleSy takes as input a set of example problems (e.g., $x + 1 = 2$), a set of basic *axiom* rules for solving these problems (e.g., factoring), and a function expressing the desired educational objective. Given these inputs, it first synthesizes a set of *tactic* rules (e.g., combining like terms) that integrate multiple axioms into advanced problem-solving strategies. The axioms and tactics are then searched for a domain model that optimizes the objective. RuleSy is based on new algorithms for mining tactic specifications from examples and axioms, synthesizing tactic rules from these specifications, and selecting an optimal domain model from the axioms and tactics.

We evaluate RuleSy on the domain of K-12 algebra, finding that it recovers textbook tactics and domain models, discovers new tactics and models, and outperforms a prior tool for this domain by orders of magnitude. But RuleSy generalizes beyond K-12 algebra: we also use it to (re)discover proof tactics for propositional logic, demonstrating its potential to aid in the development of custom models for a variety of educational domains.

Additional Key Words and Phrases: Program Synthesis

1 INTRODUCTION

Background. A key challenge in the design of educational applications is modeling the operational knowledge that captures the expertise of a given domain. This knowledge takes the form of a *domain model*, which consists of rules that experts apply to solve problems in that domain. For example, factoring, $ax + bx \rightarrow (a + b)x$, is an expert problem-solving rule for K-12 algebra. Educational applications rely on domain models to automate tasks such as problem and progression generation [2], hint and feedback generation [29], student modeling [3], and misconception detection [39].

Domain modeling is a pressing problem in education technology for two reasons. First, domain models are currently created by hand, taking hundreds of hours of development time to model a single hour of instructional material [27]. Second, many different domain models can capture the operational knowledge for a domain, and basing an application on a single model limits its usefulness. Recent research [24] shows that to achieve mastery by over 95% of students, some students need over six times more content than others. To best serve a broad population of students, applications therefore need multiple models that optimize different educational objectives, such as maximizing solving efficiency by including many specialized rules versus minimizing memorization burden by including as few general rules as possible. For both of these reasons—the high cost of building a

single model and the need for many custom models—domain modeling is a major technical roadblock in ongoing efforts to build applications that deliver effective personalized education [9, 31].

Problem and Scope. This paper proposes a new approach to rapid development of optimal domain models that is based on program synthesis. We realize this approach in RuleSy, a framework that assists developers with both creating problem-solving rules and assembling them into domain models that optimize desired educational objectives. The RuleSy framework was motivated by practical experience: some of the authors work for an educational technology company building adaptive K12 applications that need custom domain models. These applications are the intended use case for our work, and our company is already adopting key ideas from RuleSy (e.g., the DSL from Section 3). Here, we focus on presenting and evaluating the novel technical aspects of RuleSy, with the future goal of integrating the framework fully into our company's workflow and evaluating its impact on the productivity of the company's developers.

Approach. RuleSy is a framework for modeling domains that express operational knowledge with *condition-action rules*, such as $ax + bx \longrightarrow (a + b)x$. A rule of this form describes one step that an expert would take to solve a problem. The rule's condition recognizes problem states that trigger rule application, and its action specifies how to transition to the next problem state from the current one. For example, the factoring rule $ax + bx \longrightarrow (a + b)x$ rewrites algebra terms of the form $ax + bx$ to $(a + b)x$. These rules are easy to explain and align with existing instructional design [40]. But crafting a set of such rules (i.e., a domain model) that best meets an educational objective is difficult, and RuleSy helps automate this task.

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 and constant folding, $c_0 + c_1 \longrightarrow c_2$ 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, $c_0x + c_1x \longrightarrow c_2x$, which composes factoring and constant folding? While such compound rules, or *tactics*, are not necessary for problem solving, some are always included in textbook domain models (e.g., Charles et al. [8]) because they enable more efficient problem solving with less cognitive load [36]. Yet there is a limit to how many rules students can remember, so the optimal mix of axioms and tactics for a domain model depends on the desired tradeoff between maximizing solving efficiency and minimizing the memorization burden.

RuleSy helps navigate such tradeoffs by generating an optimal domain model given only a set of axioms for the domain, a set of representative example problems, and an objective function that expresses the desired optimization criteria in terms of rule and solution costs. Using the axioms and the examples, RuleSy first synthesizes an exhaustive set of tactics that combine multiple axioms into advanced problem-solving strategies. Each of these tactics shortens the solution to at least one example problem compared to using the axioms alone. Having generated the tactics, RuleSy then applies discrete optimization to both the axioms and the tactics, producing a domain model that solves the example problems and optimizes the provided objective.

To enable tactic synthesis, RuleSy treats condition-action rules as programs in a high-level domain-specific language (DSL). These programs consume and produce problem states expressed as trees; a program consists of a pattern to match against a tree, and a set of tree editing actions to apply to the matched subtrees. A condition-action rule (and, hence, a program in our DSL) can *fire* on any part of the problem state to transform it to another state. For example, given the problem $z + 0 = 0 + y$, the additive identity rule $x + 0 \longrightarrow x$ can fire on either $z + 0$ or $0 + y$ (due to the commutativity of addition), giving rise to two possible output states, $z = 0 + y$ or $z + 0 = y$. This non-determinism makes rule programs a challenging target for synthesis—to our knowledge, existing tools can neither synthesize such programs directly, nor scale to support indirect encodings that enumerate all possible applications of a rule.

RuleSy tackles the rule synthesis challenge with a novel reduction to a set of classic syntax-guided synthesis [1] queries. The reduction employs two stages: *specification mining* and *rule synthesis*. Specifications mining uses the axioms and the examples to produce a set of deterministic functions that define sound and useful tactics. In particular, each mined function captures a subset of the input-output behaviors allowed by a sequence of axiom applications, including behaviors that shorten the solution to at least one example problem. The rule synthesis stage uses these functions as specifications, thus side-stepping the need to reason about the semantics of rule firing. To scale to practical rules, RuleSy employs a novel formulation of the classic syntax-guided synthesis query, exploiting the structure of its DSL and specifications to reduce the synthesis search space asymptotically.

Once it synthesizes the tactics for a given set of examples and axioms, RuleSy needs to explore the resulting design space to find a domain model that solves the examples and optimizes the desired objective. Finding such a model is undecidable in general, since an arbitrary set of rules (i.e., a candidate model) in our DSL may not be terminating [11]. We therefore constrain this optimization task to be decidable and develop a new algorithm for discharging it. In particular, RuleSy finds a domain model that solves the examples while minimizing the objective over the model's rules and the shortest (rather than all) solutions obtainable with those rules.

Our work builds on a short paper by Butler et al. [7], published in the education technology literature. That paper argued for computer-aided development of custom domain models, and presented a prototype tool for creating such models for the domain of introductory K-12 algebra. The prior tool employs a procedure for synthesizing tactics that is neither sound nor complete, and it uses heuristics to try and tackle the undecidability of the domain model optimization query. Unlike that tool, RuleSy generalizes beyond K-12 algebra, and it is based on new algorithms for sound specification mining, sound and complete rule synthesis, and decidable optimization.

Evaluation. To evaluate RuleSy, we applied it to the domain of introductory K-12 algebra, seeking to assess the quality of its output and the performance of its algorithms. We use a standard algebra textbook [8] as the baseline for evaluating the quality of the synthesized tactics and domain models, and the prior tool by Butler et al. [7] as the baseline for evaluating runtime performance. Applying RuleSy to examples and axioms from the textbook, we find that it recovers standard algebra tactics (such as combining like terms) presented in the book, while also discovering additional advanced tactics. We also find that RuleSy can not only recover the textbook's domain model when given a specific objective, but that it can also discover different variants of this model that optimize different objectives. Finally, we show that RuleSy significantly outperforms the prior tool by Butler et al. [7], both in terms of output quality and runtime of its algorithms.

While our evaluation focuses on the domain of K-12 algebra, our approach is not specific to this domain. The approach assumes that problem states are represented as trees and that rule programs transform trees. To show the framework's generality, we instantiate it a with a DSL for proving theorems in propositional logic, and use it to synthesize proof tactics from textbook [6] axioms and exercises. We find that RuleSy synthesizes both new and standard tactics for this domain (e.g., modus ponens), just as it did for K-12 algebra.

Summary. In summary, this paper makes the following contributions:

- A formal development of the RuleSy framework for computer-aided development of optimal domain models, expressed as sets of condition-action rules.
- New algorithms for mining tactic specifications from a set of examples and axioms; for synthesizing rule programs that implement those specifications; and for selecting a subset of the axioms and tactics that solves the examples while optimizing a given objective.

- An extended case study demonstrating the effectiveness of RuleSy in the domain of introductory algebra, and a brief case study demonstrating the framework's applicability to other domains, such as logic proofs.

The rest of the paper is organized as follows. Section 2 presents an overview of RuleSy and its application to a toy algebra domain. Section 3 describes the DSL for condition-action rules that RuleSy targets for synthesis. Section 4 describes our algorithms for specification mining, rule synthesis, and rule set optimization. Section 5 presents two case studies that apply RuleSy to the domains of K-12 algebra and propositional logic. We conclude with a discussion of related work in Section 6 and a summary of contributions in Section 7.

2 OVERVIEW

This section illustrates RuleSy's functionality on the problem of creating optimal domain models for a toy algebra domain. The domain focuses on solving linear equations that involve addition of variables and constants, e.g., $x + 2 = 3$. People solve an equation of this kind by repeatedly applying rewrite rules until the unknown (x) is isolated on one side of the equality symbol ($x = 1$). Many sets of rules (i.e., domain models) can be used to accomplish this task depending on the educational objective—e.g., minimizing memorization for novice students or maximizing solving efficiency for advanced students. Our goal is to generate domain models that optimize such objectives.

To use RuleSy, the domain model developer provides a set of example problems; a set of initial rules, or *axioms*, sufficient to define the domain and solve the example problems; and an educational objective expressed as a function of rule and solution costs. RuleSy then synthesizes a set of shortcut, or *tactic*, rules, and finds a subset of the axioms and tactics that optimizes the objective. This section shows sample inputs for our toy algebra domain and uses them to illustrate RuleSy's algorithmic pipeline.

2.1 Examples, Axioms, and Objectives

Examples. Figure 1 shows the example problems we will use for the toy algebra domain. The problems (b) are represented as s-expressions (c). 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. Such a problem is solved by reducing it to an equation of the form $x = c$.

Axioms. To solve a toy algebra equation, we use a set of *condition-action rules* to repeatedly rewrite the problem until it takes the reduced form. Figure 1a shows three such rules that are sufficient to solve our example problems (as well as all problems expressed in the syntax from Figure 1c). A rule consists of a *condition*, which matches a syntax tree with a specific shape, and an *action*, which creates a new tree by applying editing operations (such as adding or removing nodes) to the matched tree. For example, rule **A** matches trees of the form $(+\ 0\ e\ \ldots)$, where the order of subtrees is ignored, and it rewrites such trees by removing the constant 0 to produce $(+\ e\ \ldots)$. These rules will serve as our axioms for solving toy algebra problems; e.g., we can solve p_1 in two steps, by applying the rules $\mathbf{B} \circ \mathbf{A}$ 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 3) that can solve the example problems in fewer steps.

Objective. By design, axioms are more general than tactics but lead to longer solutions, while tactics lead to more efficient solutions but apply less generally. The best mix of axioms and tactics for a domain model therefore depends on the desired educational objective. RuleSy expresses these objectives as functions of rule and solution costs. Figure 1d shows an example objective function, given as a weighted sum of two components: rule set complexity and solution efficiency, which are proxy measures for the difficulty of learning and applying knowledge encoded in the given

```
; Additive identity: (+ 0 e ...) → (+ e ...)
(define A
  (Rule
    (Condition
      (Pattern (Term + (ConstTerm) _ etc))
      (Constraint (Eq? (Ref 1) 0)))
    (Action (Remove (Ref 1)))))

; Constant folding: (+ c1 c2 ...) → (+ c ...),
;     c = c1 + c2
(define B
  (Rule
    (Condition
      (Pattern
        (Term + (ConstTerm) (ConstTerm) etc))
      (Constraint true))
    (Action
      (Replace (Ref 1)
               (Apply + (Ref 1) (Ref 2)))
      (Remove (Ref 2)))))

; Adding the negation of a term to both sides:
; (= (+ e0 ...) e1) → (= (+ (- e0) e0 ...) (+ e1 (- e0)))
(define C
  (Rule
    (Condition
      (Pattern (Term = (Term + _ etc) _))
      (Constraint true))
    (Action
      (Replace
        (Ref 1)
        (Cons (Make - (Ref 1 1)) (Ref 1)))
      (Replace
        (Ref 2)
        (Make + (Ref 2)
                (Make - (Ref 1 1)))))))
```

(a) Axioms as production rules in our DSL.

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 for the toy algebra domain.

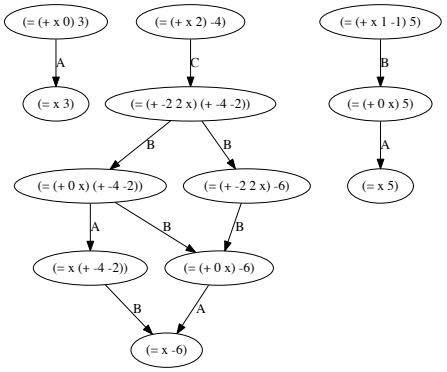
p : $(= \ e \ c)$
e : $id \mid (+ \ id \ c^+)$
id : identifier
c : integer literal

(c) Syntax for example problems.

```
; Computes α · R(Rules) + (1 − α) · S(Solutions), where
; α ∈ [0, 1] is a weighting term, R is the sum of
; the costs of the Rules, and S is the average
; cost of the Solutions.
(define (objective Rules Solutions alpha)
  (+ (* alpha
        (apply + (map rule-cost Rules)))
     (* (- 1 alpha)
        (apply
          mean
          (map sol-cost Solutions)))))
```

(d) A sample objective function.

Fig. 1. The inputs to our system for the toy algebra domain consist of a set of axioms (a), example problems (b-c), and an objective function (d). The axioms (a) are expressed in the production rule DSL described in Section 3. The rule condition consists of a pattern to match against the input's abstract syntax tree (AST) and a constraint that the matched AST must satisfy; the action specifies a set of functional edits to this AST. A reference expression (**Ref** $i \ldots$) identifies a nested subexpression in the matched AST, with child indexing starting at 1. Rule application ignores the order of arguments to commutative operators. The example problems (b) are expressed in a simple S-expression syntax (c). The objective function (d), used for choosing an optimal subset of learned rules, is a weighted sum of the cost of a candidate rule set and the cost of applying it to solve the input problems.



(a) All shortest solutions for the toy problems in Figure 1b, using the axioms in Figure 1a.

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

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

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

(c) Using the plan in (b) to solve the problem p_1 (Figure 1b).

Fig. 2. Mining a tactic specification from the problem p_1 (Figure 1b) using the toy algebra axioms (Figure 1a). RuleSy first finds all shortest solution paths for each problem using the axioms (a). A specification takes the form of a plan (b) for applying a sequence of axioms to a given AST (c). Each step in the plan specifies a permutation of its input that satisfies the axiom's condition, given as a binding from the references (**Ref** $i \ldots$) to the indices they match in the input AST.

domain model [36]. Rule set complexity is taken to be the sum of individual rule costs, where the cost of a rule is a function of its syntactic structure. Solution efficiency measures the average cost of solving all example problems using a given rule set. We take the cost of solving a problem to be the sum of the costs of the individual rewrite steps, measured by the number of syntax tree manipulations performed during the application of a rule. RuleSy finds the mix of axioms and tactics that optimizes the desired trade-off between rule set complexity and solution efficiency.

2.2 Specification Mining

To find an optimal domain model, RuleSy first needs to synthesize a set of tactic rules that can help 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" rule that composes the axioms **B** and **A**. In contrast, creating a rule that composes **C** with itself would not help solve any of the examples in fewer steps. RuleSy determines which rules to synthesize, and how those rule should behave, by mining a set of *tactic specifications* from the input problems and axioms.

A tactic specification defines a *plan* for applying a sequence of axioms, as shown in Figure 2b. A plan describes which axioms to apply, in what order, and how. Since an axiom may be applied to an algebraic expression in multiple ways, a plan associates each axiom with an application index and a binding for the axiom's pattern. The application index identifies a subtree in the expression's abstract syntax tree (AST), and the binding specifies a mapping from the index space of the axiom's pattern to the index space of the subtree. The plan in Figure 2b solves p_1 in one step by reducing the expression $(+\, x\, 1\, {-1})$ to x. In essence, plans represent functional specifications of tactic rules that can help solve the example problems in fewer steps. This removal of non-determinism from tactic specifications is key to making synthesis tractable, since we need to be able to reduce tactic semantics to efficiently-solvable SMT formulas in order to verify that a candidate rule implements a given specification.

RuleSy computes tactic specifications in two stages. The first stage uses the axioms to find all shortest solutions to each example problem. Figure 2a shows the shortest solutions for our toy examples and axioms; we represent the solutions to a given problem as a directed acyclic multigraph with a single source and at least one sink. The second stage (Figures 2b and 2c) enumerates a set of plans that describe all paths between every pair of nodes in the resulting graphs. These plans capture the behaviors of all solution-shortening tactics that can be inferred from the given example problems and axioms.

2.3 Rule Synthesis

Given a set of tactic specifications, RuleSy synthesizes a set of tactic rules, expressed in the same DSL (Section 3) as the input axioms. Our synthesis algorithm builds on an existing tool [38] for syntax-guided synthesis [1, 34], which takes as input a specification and a *sketch* of the implementation. A sketch is a program with missing expressions, called "holes," to be filled by the synthesizer so that the resulting implementation satisfies the specification. Ideally, a sketch would consist of a single hole, and the synthesizer would fill that hole with a correct condition-action program. But this naive approach does not scale in practice. To achieve scalability, RuleSy implements a sound and complete technique for synthesizing conditions independently of actions.

Figure 3 shows two sample tactics synthesized for our toy algebra tutor using the specifications (e.g., Figure 2c) mined from the toy examples and axioms. Note that the tactics perform fewer tree manipulations than the axiom sequences they replace, leading to cheaper (and shorter) solutions. For example, the tactic **BA** performs two tree manipulation actions, while the axiom sequence $\mathbf{B} \circ \mathbf{A}$ performs three such actions. But the tactic also applies to fewer problem states than the axioms. RuleSy uses discrete optimization to decide which axioms and tactics to include in a domain model, depending on the desired educational objective.

```
; Canceling opposite constants:
; (+ c -c e ...) → (+ e ...).
(define BA
  (Rule
    (Condition
      (Pattern
        (Term + (ConstTerm) (ConstTerm) _ etc))
      (Constraint
        (Eq? (Ref 1) (Apply - (Ref 2)))))
    (Action
      (Remove (Ref 1))
      (Remove (Ref 2)))))
```

```
; Move negated constant to other side with
; only one other term:
; (= (+ c1 e ...) c2) → (= (+ e ...) c), c = c2 - c1.
(define CBAB
  (Rule
    (Condition
      (Pattern
        (Term = (Term + (ConstTerm) _ etc)
                (ConstTerm)))
      (Constraint true))
    (Action
      (Remove (Ref 1 1) 0)
      (Replace
        (Ref 2)
        (Apply - (Ref 2) (Ref 1 1))))))
```

Fig. 3. Sample tactics synthesized for the specifications (e.g., Figure 2b) mined from the toy inputs (Figure 1b and 1a).

2.4 Rule Set Optimization

The optimizer takes as input the objective function, the example problems, and the axiom and tactic rules. The objective returns a positive real value when applied to a set of rules and a set of solutions. Given these inputs, the optimizer finds a subset of the rules that is sufficient to solve the example problems, while minimizing the objective over all shortest solutions obtainable with such sufficient subsets.

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

Fig. 4. Optimal rule sets for our toy problems (Figure 1b), rules (Figure 1a and 3), and objective (Figure 1d).

```
program := (Rule cond action)
   cond := (Condition (Pattern pattern) (Constraint constr))
pattern := _ | (ConstTerm) | (VarTerm) | (BaseTerm)
         | (Term op pattern+) | (Term op pattern+ etc)
 constr := true | pred | (And constr constr)
   pred := (Eq? ref const) | (Neq? ref const)

 action := (Action cmd+)
    cmd := (Remove ref) | (Replace ref expr)

   expr := const | obj
    obj := (Make op expr+) | (Cons expr ref)
         | (Cons expr obj)
  const := int | ref | (Apply op const+)
    ref := (Ref) | (Ref int+)

   term := int | var | (op term+)
    int := integer literal
    var := identifier
     op := + |- |* | / | =
```

Fig. 5. Syntax for the algebra DSL. The notation $form^+$ means one or more repetitions of the given form.

Figure 4 shows two sample optimal rule sets for our toy domain. The rule sets $\mathcal{R}_{0.1}$ and $\mathcal{R}_{0.9}$ minimize the value of the toy objective (Figure 1d) for different values of the weighting factor `alpha`. Setting `alpha` to 0.1 emphasizes solution efficiency, while setting it to 0.9 emphasizes rule set economy. As a result, $\mathcal{R}_{0.1}$ includes more rules than $\mathcal{R}_{0.9}$. RuleSy thus helps with both creating new tactics and using them to form domain models that best satisfy desired objectives.

3 A DOMAIN-SPECIFIC LANGUAGE FOR CONDITION-ACTION RULES

This section presents a RuleSy language for specifying condition-action rules, instantiated for the domain of K-12 algebra. We use this instantiation to show examples throughout the paper, and to describe our rule synthesis and optimization algorithms. But RuleSy generalizes beyond the domain of K-12 algebra: it is applicable to any domain in which problems are expressed as (abstract syntax) trees and rules as condition-action programs that transform those trees. We describe another instantiation of RuleSy, for the domain of propositional logic, in Section 5.

3.1 Specifying Problems and Rules

Our algebra DSL (Figure 5) represents rules as *programs* that operate on algebra problems expressed as *terms*. A term may be an integer constant, a variable, or an expression that combines terms using standard arithmetic operators. RuleSy is parametric in the definition of terms; for example, we could instantiate the framework with boolean terms by using boolean constants instead of integers and logical connectives instead of arithmetic operators. The structure of rules, on the other hand, is fixed. A rule consists of a *condition*, which determines if the rule is applicable to a given term, and an *action*, which specifies how to transform terms. Conditions include a *pattern* to match against the term's structure and a boolean *constraint* to evaluate on that structure. Actions are sequences of term editing operations, such as removing or replacing a subterm. Both constraints and actions can use *references* to identify specific subterms of the term to which the rule is being applied. The toy problems (Figure 1b) and rules (Figures 1a and 3) from Section 2 are all valid terms and programs in the algebra DSL.

3.2 Interpreting Rules

Semantics of Rule Firing. Rule programs denote partial functions from terms to terms (Figure 6). If a term satisfies the rule's condition, the result is a term; otherwise, the result is an undefined value ($\perp$). We solve algebra problems by applying rules exhaustively, via $\mathit{fire}(R, t)$, on permutations and subterms of a given term. Permuting a term (Definition 3.3) reorders the arguments to any commutative operators while leaving the rest of the term's structure unchanged. To fire a rule on

$$
\begin{array}{ll}
[\![(\texttt{Rule}\, c\, a)]\!]\, t & = \textbf{if } [\![c]\!]\, t \textbf{ then } [\![a]\!]\, t \textbf{ else } \bot \\
\\
[\![(\texttt{Condition}\, p\, b)]\!]\, t & = [\![p]\!]\, t \wedge [\![b]\!]\, t \\
[\![(\texttt{Pattern}\, p)]\!]\, t & = [\![p]\!]\, t \\
[\![(\texttt{Constraint}\, b)]\!]\, t & = [\![b]\!]\, t \\
\\
[\![(\texttt{Term}\, o\, p_1 \ldots\, p_k)]\!]\, t & = (t = (o\ t_1 \ldots\, t_k)) \wedge \forall_{1 \le i \le k} [\![p_i]\!]\, t_i \\
[\![(\texttt{Term}\, o\, p_1 \ldots\, p_k\, \texttt{etc})]\!]\, t & = (t = (o\ t_1 \ldots\, t_n)) \wedge n \ge k \wedge \forall_{1 \le i \le k} [\![p_i]\!]\, t_i \\
[\![(\texttt{ConstTerm})]\!]\, t & = \mathit{literal}(t) \\
[\![(\texttt{VarTerm})]\!]\, t & = \mathit{variable}(t) \\
[\![(\texttt{BaseTerm})]\!]\, t & = \mathit{literal}(t) \vee \mathit{variable}(t) \\
[\![\_]\!]\, t & = \text{true} \\
[\![\texttt{true}]\!]\, t & = \text{true} \\
[\![(\texttt{Eq?}\, r\, e)]\!]\, t & = ([\![r]\!]\, t = [\![e]\!]\, t) \\
[\![(\texttt{Neq?}\, r\, e)]\!]\, t & = ([\![r]\!]\, t \neq [\![e]\!]\, t) \\
[\![(\texttt{And}\, b_1\, b_2)]\!]\, t & = [\![b_1]\!]\, t \wedge [\![b_2]\!]\, t \\
\\
[\![(\texttt{Action}\, a_1 \ldots\, a_k)]\!]\, t & = ([\![a_1]\!] \parallel \ldots \parallel [\![a_k]\!])(t) \\
[\![(\texttt{Remover})]\!]\, t & = \mathit{rm}(t,\, \mathit{index}(r)) \\
[\![(\texttt{Replacer}\, e)]\!]\, t & = \mathit{replace}(t,\, \mathit{index}(r),\, [\![e]\!]\, t) \\
\\
[\![(\texttt{Make}\, o\, e_1 \ldots\, e_k)]\!]\, t & = (o\ [\![e_1]\!]\, t\ \ldots\ [\![e_k]\!]\, t) \\
[\![(\texttt{Cons}\, e_1\, e_2)]\!]\, t & = \mathit{cons}([\![e_1]\!]\, t,\, [\![e_2]\!]\, t) \\
[\![(\texttt{Apply}\, o\, e_1 \ldots\, e_k)]\!]\, t & = [\![o]\!]([\![e_1]\!]\, t,\, \ldots,\, [\![e_k]\!]\, t) \\
[\![(\texttt{Ref}\, i_1 \ldots\, i_k)]\!]\, t & = \mathit{ref}(t,\, [i_1,\, \ldots,\, i_k]) \\
\\
\mathit{index}((\texttt{Ref}\, i_1 \ldots\, i_k)) & = [i_1,\, \ldots,\, i_k] \\
\mathit{replace}(t,\, [],\, s) & = s \\
\mathit{replace}((o\ t_1 \ldots\, t_k),\, [i],\, s) & = (o\ t_1\ \ldots\ t_{i-1}\ s\ t_{i+1} \ldots\ t_k) \\
\mathit{replace}((o\ t_1 \ldots\, t_k),\, [i, j, \ldots],\, s) & = (o\ t_1 \ldots \mathit{replace}(t_i,\, s,\, [j, \ldots]) \ldots\, t_k) \\
\mathit{rm}((o\ t_1 \ldots\, t_k),\, [i]) & = (o\ t_1\ \ldots\ t_{i-1}\ t_{i+1} \ldots\ t_k) \\
\mathit{rm}((o\ t_1 \ldots\, t_k),\, [i, j, \ldots]) & = (o\ t_1\ \ldots\ \mathit{rm}(t_i,\, [j, \ldots])\ \ldots\ t_k) \\
\mathit{cons}(t,\, (o\ t_1 \ldots\, t_k)) & = (o\ t\ t_1\ \ldots\ t_k) \\
\\
\mathit{fire}(R,\, t) = \{[\![R]\!]\, t \mid [\![R]\!]\, t \neq \bot\} \textbf{ where } \mathit{literal}(t) \vee \mathit{variable}(t) \\
\mathit{fire}(R,\, t) = \{[\![R]\!]\, t^{\beta} \mid [\![R]\!]\, t^{\beta} \neq \bot \wedge \beta \in \mathcal{B}(\mathit{pattern}(R),\, t)\} \cup \\
\qquad \bigcup_{1 \le i \le n} \{\mathit{replace}(t,\, [i],\, s) \mid s \in \mathit{fire}(R,\, t_i)\} \\
\qquad \textbf{where } t = (o\ t_1 \ldots\, t_n)
\end{array}
$$

Fig. 6. Semantics for the algebra DSL (Figure 5). The expression $(o\ t_1\ \ldots\ t_n)$ constructs a term with the given operator and children; $\parallel$ stands for parallel function composition; $[x, \ldots]$ denotes a sequence; and other notation is described in Definitions 3.2-3.5.

a term, we permute the term "just enough" to establish a one-to-one mapping between the rule's pattern and the term's structure (Definition 3.5). By establishing such mappings for all subterms of a term, *fire* implements the intuitive notion of rule application given in Section 2: a rule fires on all subterms that satisfy the rule's condition, ignoring the order of arguments to commutative operators.

Example 3.1. To illustrate the semantics of *fire*, consider firing the rule **A** from Figure 1a on the term $t = (+\ (*\ x\ 2)\ 0)$. 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 $t, (*\ x\ 2), x, 2, 0$, respectively (Definition 3.2). Since both addition and multiplication are commutative, valid tree permutations $\Pi(t)$ for t consist of the following mappings (Definition 3.3):

- $t^{\pi_0} = t$ where $\pi_0 = \{[] \mapsto [], [1] \mapsto [1], [1, 1] \mapsto [1, 1], [1, 2] \mapsto [1, 2], [2] \mapsto [2]\}$;
- $t^{\pi_1} = (+\ (*\ 2\ x)\ 0)$ where $\pi_1 = \{[] \mapsto [], [1] \mapsto [1], [1, 1] \mapsto [1, 2], [1, 2] \mapsto [1, 1], [2] \mapsto [2]\}$;
- $t^{\pi_2} = (+\ 0\ (*\ x\ 2))$ where $\pi_2 = \{[] \mapsto [], [1] \mapsto [2], [1, 1] \mapsto [2, 1], [1, 2] \mapsto [2, 2], [2] \mapsto [1]\}$; and
- $t^{\pi_3} = (+\ 0\ (*\ 2\ x))$ where $\pi_3 = \{[] \mapsto [], [1] \mapsto [2], [1, 1] \mapsto [2, 2], [1, 2] \mapsto [2, 1], [2] \mapsto [1]\}$.

Next, we observe that the scope (Definition 3.4) of **A**'s pattern consist of the indices $\{[], [1], [2]\}$. Finally, we use the permutations of t and the scope of **A** to compute all valid bindings for **A** and t (Definition 3.5): $\beta_0 = \{[] \mapsto [], [1] \mapsto [1], [2] \mapsto [2]\}$ with $t^{\beta_0} = t^{\pi_0}$ and $\beta_1 = \{[] \mapsto [], [1] \mapsto$

$[2], [2] \mapsto [1]\}$ with $t^{\beta_1} = t^{\pi_2}$. The rule $\mathbf{A}$ applies only to the binding β_1 (and none of the subterms of t), so $\mathit{fire}(\mathbf{A}, t)$ yields $\{(* \; x \; 2)\}$.

Definition 3.2 (Tree Indices). A *tree index* is a finite sequence of positive integers that identifies a subterm of a term as follows:

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

We write $\mathit{refs}(t)$ to denote the set $\{\mathit{idx} \mid \mathit{ref}(t, \mathit{idx}) \neq \bot\}$.

Definition 3.3 (Tree Permutations). A function π is a *tree permutation* for a term t if it defines a bijective mapping from $\mathit{refs}(t)$ to itself. A permutation π is *valid* for t if it reorders only the children of commutative operators in t. That is, for each $[i_1, \ldots, i_n] \in \mathit{refs}(t)$, $\pi([i_1, \ldots, i_n]) = [j_1, \ldots, j_n]$ such that $\pi([i_1, \ldots, i_{n-1}]) = [j_1, \ldots, j_{n-1}]$ and $i_n = j_n$ or $\mathit{ref}(t, [i_1, \ldots, i_{n-1}]) = (\mathit{op} \; \ldots)$ where op is commutative. We write $\Pi(t)$ to denote the set of all valid permutations of t, and t^π to denote the term obtained by applying $\pi \in \Pi(t)$ to $\mathit{refs}(t)$.

Definition 3.4 (Scopes). A tree index idx is in the *scope* of a condition pattern p if $\mathit{scope}(p, \mathit{idx}) \neq \bot$ where:

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

We write $\mathit{scope}(p)$ to denote the set $\{\mathit{idx} \mid \mathit{scope}(p, \mathit{idx}) \neq \bot\}$.

Definition 3.5 (Bindings). Let β be a bijection from tree indices to tree indices with a finite domain $\mathit{dom}(\beta)$ and range $\mathit{ran}(\beta)$. We say that β is a *binding* for a pattern p if the domain of β is the scope of p; i.e., $\mathit{dom}(\beta) = \mathit{scope}(p)$. A binding β is *valid* for a term t if there is a permutation $\pi \in \Pi(t)$ such that $\beta^{-1} \subseteq \pi$ and for all $[i_1, \ldots, i_n] \in \mathit{refs}(t)$, if $[i_1, \ldots, i_k] \in \mathit{ran}(\beta)$ and $[i_1, \ldots, i_{k+1}] \notin \mathit{ran}(\beta)$, then $\pi([i_1, \ldots, i_n]) = \beta^{-1}[i_1, \ldots, i_k] \oplus [i_{k+1}, \ldots, i_n]$, where $\oplus$ stands for sequence concatenation. We define $\mathit{bind}(\beta, t)$ to return an arbitrary but deterministically chosen permutation $\pi \in \Pi(t)$ for which β is valid, if one exists, or $\bot$ otherwise. We write $\mathcal{B}(p, t)$ to denote the set $\{\beta \mid \mathit{dom}(\beta) = \mathit{scope}(p) \wedge \mathit{bind}(\beta, t) \neq \bot\}$ of all valid bindings for p and t, and we write t^β to denote $t^{\mathit{bind}(\beta, t)}$.

Semantics of Conditions and Actions. Rule conditions denote functions from terms to booleans, and actions are functions from terms to terms. A condition maps a term to 'true' if the term matches the condition's pattern and satisfies its constraint. The pattern $(\texttt{Term} \; o \; p_1 \ldots p_k)$ matches terms t with the operator o and with k children that match the subpatterns $p_1 \ldots p_k$. If a **Term** pattern ends with **etc**, then t can have an arbitrary number of unmatched children, in addition to the k matched children. Other patterns match literals, variables, both, or all terms. Constraints capture conditions that are not expressible through pattern matching, such as two subterms being syntactically equal. Actions apply a set of parallel functional edits to disjoint subterms of the input term t. Actions can create new terms (via **Make**), and both conditions and actions can evaluate expression terms with literal arguments (via **Apply**).

Well-formed Rule Programs. The meaning of rule conditions and actions is defined only for *well-formed programs* (Definition 3.6), which contain no invalid references. A reference expression $(\texttt{Ref} \; i_1 \ldots i_n)$ specifies an index $[i_1, \ldots, i_n]$ into the matched term's syntax tree (Definition 3.2). The empty index [] identifies the term itself; $[i]$ identifies the term's i^{th} child; $[i, j]$ identifies the

i^{th}'s child j^{th} child and so on. If a term matches the pattern of a well-formed program, then every reference in that program is guaranteed to specify a valid index into the term's abstract syntax tree. Additionally, **`Apply`** and **`Cons`** expressions always reference subterms of the right kind; the program's actions always 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. RuleSy consumes and creates only well-formed programs.

Definition 3.6 (Well-Formed Programs). Let R be a rule with the condition (**`Condition`** (**`Pattern`** p) (**`Constraint`** b)) and action (**`Action`** $a_1 \dots a_n$). We say that R is *well-formed* if the following constraints hold:

- Each (**`Ref`** $i_1 \dots i_n$) expression in R specifies a tree index that is in the scope of p: $[i_1, \dots, i_n] \in scope(p)$.
- For all references e_i in an (**`Apply`** o $e_1 \dots e_n$) expression, $scope(p, index(e_i))$ is a (**`ConstTerm`**) pattern.
- If e_2 is a reference in a (**`Cons`** e_1 e_2) expression, then $scope(p, index(e_2))$ is a (**`Term`** ...) pattern.
- Let r_k denote the first argument to a command a_k in R, which is always a reference. For all distinct a_i, a_j in R, $index(r_i)$ is not a prefix of $index(r_j)$ and vice versa.
- For all references e in an **`Eq?`** or **`Neq?`** predicate, $scope(p, index(e))$ is not a **`Term`** pattern.

4 RULE MINING, SYNTHESIS, AND OPTIMIZATION

The RuleSy framework computes an optimal set of condition-action rules for a given educational objective, example problems, and axioms for solving those problems. Section 2 illustrates the three stages of this computation: (1) specification mining, (2) rule synthesis, and (3) rule set optimization. This section presents the algorithms underlying each stage and states their guarantees.[1]

4.1 Specification Mining

As illustrated in Section 2.2, specification mining takes as input a set of examples and 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 introduce the problem of specifying tactics next; show how our notion of *execution plans* addresses it; and present the FindSpecs algorithm for computing such plans from a given set of examples and axioms.

Specifying Tactics. To enable synthesis of useful rules, a tactic specification should satisfy two requirements: (1) it should describe a partial function from terms to terms, and (2) this function should capture a general problem-solving strategy that helps solve some problems in fewer steps than axioms alone. The first requirement arises from the semantics of the RuleSy DSL: since rule programs denote partial functions from terms to terms, synthesis would necessarily fail on specifications that capture non-functional relations.[2] The second requirement ensures that no individual axiom denotes the specified function, and that the function is useful for solving problems in the target domain. Naïve specification approaches can easily satisfy one of these requirements but not both.

To see why, suppose that we specify the semantics of tactics using axiom sequences such as $\mathbf{I} \circ \mathbf{B}$, where $\mathbf{I}$ implements factoring (Figure 7a) and $\mathbf{B}$ implements constant folding (Figure 1a). Intuitively,

[1]Proofs of these statements are available in Appendix A.

[2]In this paper, we are interested in synthesizing a rule program R that satisfies a given specification S on all terms, i.e., $\forall t. [\![R]\!]t = [\![S]\!]t$. In principle, one could instead synthesize a rule program R that satisfies S when fired on all terms, i.e., $\forall t. fire(R, t) = [\![S]\!]t$, which would enable use of non-functional specifications S. But in order to verify that a candidate program R is correct according to the latter formulation, the synthesizer would have to reason about the semantics of *fire*, which, in turn, requires reasoning about the semantics of R on all permutations and subterms of an arbitrary term t—an intractable task even for state-of-the-art SMT solvers.

we would like this sequence to capture the meaning of the tactic **IB** (Figure 7b) for combining like terms, which applies both factoring and constant folding. Yet no interpretation of the sequence $\mathbf{I} \circ \mathbf{B}$ captures the meaning of **IB** while satisfying our requirements. If we interpret $\mathbf{I} \circ \mathbf{B}$ using the *fire* semantics as $\lambda t.\mathit{fire}(\mathbf{B}, \mathit{fire}(\mathbf{I}, t))$, the resulting specification is non-functional, violating the first requirement. For example, $\mathbf{I} \circ \mathbf{B}$ can *fire* on the term $(+\,(*\,2\,x)\,(*\,3\,x)\,(*\,4\,y)\,(*\,5\,y))$ in two different ways, thus mapping the input term to both $(+\,(*\,5\,x)\,(*\,4\,y)\,(*\,5\,y))$ and $(+\,(*\,9\,y)\,(*\,2\,x)\,(*\,3\,x))$. But if we interpret this sequence as the composition of the partial functions denoted by its axioms—i.e., as $\lambda t.[\![\mathbf{B}]\!]([\![\mathbf{I}]\!]t)$—the resulting specification is trivially empty, violating the second requirement. In particular, no term produced by **I** is directly consumable by **B**. As a result, axiom sequences alone are insufficient to capture the behavior of useful tactics we would like to synthesize, such as combining like terms. The same is true for other naïve forms of specification—e.g., input-output terms extracted from solutions to example problems.

```
; Factoring:
; (+ (* e0 e) (* e1 e) ...) → (+ (* (+ e0 e1) e) ...)
(define I
  (Rule
    (Condition
      (Pattern
        (Term + (* _ _) (* _ _) etc))
      (Constraint
        (Eq? (Ref 1 2) (Ref 2 2))))
    (Action
      (Remove (Ref 1))
      (Replace
        (Ref 2 1)
        (Make + (Ref 1 1) (Ref 2 1))))))
```

(a) Axiom for term factoring.

```
; Combining like terms:
; (+ (* c0 e) (* c1 e) ...) → (+ (* c e) ...), c = c0 + c1
(define IB
  (Rule
    (Condition
      (Pattern
        (Term + (* (ConstTerm) _) (* (ConstTerm)
              _) etc))
      (Constraint
        (Eq? (Ref 1 2) (Ref 2 2))))
    (Action
      (Remove (Ref 1))
      (Replace (Ref 2 1)
               (Apply + (Ref 1 1) (Ref 2 1))))))
```

(b) Tactic for combining like terms.

Fig. 7. Combining like terms (**IB**) is a common algebra tactic that composes factoring (**I**) and constant folding (**B** in Figure 1a). Yet the axiom sequence $\mathbf{I} \circ \mathbf{B}$ does not capture its specification. Under the *fire* semantics, $\mathbf{I} \circ \mathbf{B}$ defines a non-functional relation, but composing the functions denoted by **I** and **B** yields the empty relation: i.e., $[\![\mathbf{I}]\!] \circ [\![\mathbf{B}]\!] = \emptyset$ according to the semantics in Figure 6.

Execution Plans. We address the challenge of specifying tactic rules with *execution plans*. An execution plan (Definition 4.2) is a partial function from terms to terms, encoded as a sequence of execution steps (Definition 4.1). An execution step combines a rule R with a tree index *idx* and a binding β for R's pattern. The step $\langle R, \mathit{idx}, \beta\rangle$ uses the binding β, if it is valid for the subterm $\mathit{ref}(t, \mathit{idx})$ of a term t, to evaluate the rule R. An execution step thus specifies where to apply a rule (i.e., to which subterm of a term) and how (i.e., to which permutation of the subterm), while an execution plan composes a sequence of such rule applications. For example, the plan $[\langle \mathbf{I}, [], \beta_0\rangle, \langle \mathbf{B}, [1,1], \beta_0\rangle]$, where β_0 denotes the identity binding, captures the behavior of the combine-like-terms rule on terms of the form $(+\,(*\,c_0\,e)\,(*\,c_1\,e)\,\ldots)$. 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. In essence, execution plans satisfy our requirements for tactic specifications by accounting for the semantics of *fire* (which ensures usefulness and generality) while defining functional relations (which enables synthesis).

Definition 4.1 (Execution Step). An execution step $\langle R, \mathit{idx}, \beta\rangle$ combines a rule program R with a tree index *idx* and a binding β for R's pattern. A step denotes a partial function from terms to

terms as follows: $[\![\langle R, idx, \beta\rangle]\!]t = replace(t, idx, [\![R]\!]s^{\beta})$ if $s = ref(t, idx)$, $\beta \in \mathcal{B}(pattern(R), s)$, and $[\![R]\!]s^{\beta} \neq \bot$; otherwise, $[\![\langle R, idx, \beta\rangle]\!]t = \bot$.

Definition 4.2 (Execution Plan). An execution plan S is a finite sequence of execution steps $[\langle R_1, idx_1, \beta_1\rangle, \ldots, \langle R_n, idx_n, \beta_n\rangle]$. The plan $\mathcal{S}$ composes its steps as follows: $[\![S]\!]t_0 = t_n$ if $[\![\langle R_i, idx_i, \beta_i\rangle]\!]t_{i-1} = t_i$ and $t_i \neq \bot$ for all $1 \leq i \leq n$; otherwise, $[\![S]\!]t_0 = \bot$. The plan S is *general* if the step indices $idx_1, \ldots, idx_n$ have the empty index [] as their greatest common prefix.

Computing Plans. RuleSy mines execution plans from a set of example problems and axioms using the FindSpecs procedure shown in Figure 8. FindSpecs works in two phases. First, it uses the axioms to Solve the example problems (line 3), producing a set of *solution graphs* (Definition 4.3) that encode all shortest solutions to a given problem. The Solve procedure (Appendix A) computes these graphs by performing breadth-first search to find all shortest sequences of the axioms that transform a given term to a reduced form, such as $(= x\ c)$ for the toy algebra domain. Having solved each problem, FindSpecs then computes an execution plan for every path between every pair of nodes in the problem's solution graph (line 6). The resulting plans capture the behavior of all axiom compositions (i.e., tactics) that can shorten the solution to at least one example problem (Theorem 4.8).

```
1:  function FindSpecs(T: set of terms, A: set of well-formed programs)
2:    S ← {}
3:    for all ⟨N, E⟩ ∈ {Solve(t, A) | t ∈ T} do
4:      for all src, tgt ∈ N do
5:        paths ← allPaths(src, tgt, ⟨N, E⟩)          ▷ All paths from src to tgt
6:        S ← S ∪ {⟨FindPlan(p), src, tgt⟩ | p ∈ paths ∧ |p| > 1}
7:    return S                                        ▷ Execution plans for T and A

8:  function FindPlan(p : n_0 →_{R_1} n_1 →_{R_2} ... →_{R_k} n_k)
9:    S ← an empty array of size k with indices starting at 1
10:   for all 1 ≤ i ≤ k do
11:     idx, β ← firingParameters(R_i, n_{i-1}, n_i)
12:     S[i] ← ⟨R_i, idx, β⟩
13:   root ← greatestCommonPrefix({idx | ⟨R, idx, β⟩ ∈ S})
14:   for all 1 ≤ i ≤ k do                  ▷ Drop the common prefix from all indices
15:     ⟨R, idx, β⟩ ← S[i]
16:     S[i] ← ⟨R, dropPrefix(idx, root), β⟩
17:   return S                                        ▷ A general execution plan for replaying p
```

Fig. 8. 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. The Solve procedure takes as input a term t and a set of rules $\mathcal{R}$, and outputs a solution graph that encodes all shortest solutions to t using $\mathcal{R}$. The plans produced by FindSpecs thus capture all compositions of the axioms that would lead to shorter solutions to the example problems if expressed as tactic rules.

Definition 4.3 (Solution Graph). A directed multigraph $G = \langle N, E\rangle$ is a *solution graph* for a term t, predicate Reduced, and rules $\mathcal{R}$ if $t \in N$; E is a set of labeled edges $\langle src, tgt\rangle_R$ such that $src, tgt \in N$, $R \in \mathcal{R}$, and $tgt \in fire(R, src)$; 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 term satisfies the Reduced predicate.

FindSpecs computes execution plans using the FindPlan procedure. FindPlan takes as input a path p in a solution graph and produces a general execution plan (Definition 4.2) for *replaying* that path (Definition 4.4). Intuitively, each step in the resulting plan specifies the rule application that created an edge in the path p during solving, and the plan itself specifies the sequence of rule applications that form p. In particular, given a path p from n_0 to n_k, FindPlan first creates a plan S that replays the path exactly: i.e., $[\![S]\!]n_0 = n_k$. The loop at lines 10-12 iterates over every edge $\langle n_{i-1}, n_i\rangle_{R_i}$ in p and creates an execution step for reproducing it. The function *firingParameters* (line 11) returns

the parameters used to *fire* the rule R_i on n_{i-1} to produce n_i. These include the index *idx* of the subterm to which R_i was applied, as well as the binding β for permuting that subterm. The resulting execution step (line 12) thus reproduces the edge $\langle n_{i-1}, n_i \rangle_{R_i}$: $[\![\langle R_i, idx, \beta \rangle]\!] n_{i-1} = n_i$. The second loop, at lines 13–16, generalizes S to be more widely applicable, while still replaying the path p.

Definition 4.4 (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 $[\langle R_1, idx_1, \beta_1 \rangle, \ldots, \langle R_k, idx_k, \beta_k \rangle]$, one for each edge in p, and there is an index $idx \in refs(n_0)$ such that $n_k = replace(n_0, idx, [\![S]\!] ref(n_0, idx))$.

Example 4.5. 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 2a. The Solve procedure computes this path p by firing

- **B** with $idx = [1]$, $\beta_{\mathbf{B}} = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3]\}$,
- **A** with $idx = [1]$, $\beta_{\mathbf{A}} = \{[] \mapsto [], [1] \mapsto [1]]\}$.

As a result, the loop at lines 10-12 executes twice to produce the plan $S = [\langle \mathbf{B}, idx, \beta_{\mathbf{B}} \rangle, \langle \mathbf{A}, idx, \beta_{\mathbf{A}} \rangle]$. 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 13-16 generalizes S to produce the plan in Figure 2b. This plan replays p but applies to *any* term with opposite constants as its second and third children.

Characterization. The FindSpecs algorithm is sound and complete (Theorem 4.8) in the following sense. Each generated plan describes a set of term transformations that are allowed by the axioms, without allowing any other transformations. Moreover, the union of these sets includes every transformation that can help solve at least one input problem in fewer steps.

Definition 4.6 (Soundness). Let f be a partial function from terms to terms. We say that f is *sound* with respect to a set of rules $\mathcal{R}$ if for every term t_0, $f(t_0) = \bot$ or there is a finite sequence of terms $t_1, \ldots, t_k$ such that $f(t_0) = t_k$ and $\forall i \in \{1, \ldots, k\}. \exists R \in \mathcal{R}. t_i \in fire(R, t_{i-1})$.

Definition 4.7 (Shortcuts). A path p is a *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 its sinks.

Theorem 4.8. *Let T be a set of terms, Reduced a predicate over terms, and $\mathcal{A}$ a set of rules. If every term in T can be Reduced 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 $\langle S, src, tgt \rangle \in \mathcal{S}$, $[\![S]\!]$ is sound with respect to $\mathcal{A}$, and (2) for every shortcut path p from src to tgt in a solution graph for $t \in T$, $\mathcal{A}$, and Reduced, there is a triple $\langle S, src, tgt \rangle \in \mathcal{S}$ such that S replays p.*

4.2 Rule Synthesis

RuleSy synthesizes tactics by searching for well-formed programs that satisfy specifications $\langle S, src, tgt \rangle$ 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 next; 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 syntax-guided synthesis query takes the form $\exists R.\forall t.[\![R]\!]t = [\![S]\!]t$, where R is a well-formed program in the RuleSy DSL and S is an execution plan. Existing tools [1, 34, 38] can, in principle, solve this query by searching for a correct R in a space of candidate programs defined by a syntactic sketch (**Rule** (**Condition** (**Pattern** $??_p$) (**Constraint** $??_c$)) (**Action** $\overline{??}_a$)), where $??_p$, $??_c$, and $\overline{??}_a$ stand for *holes* to be filled with a pattern, constraint, and action expressions, respectively. But in practice, these generic search engines fail to find useful rules because the classic synthesis query is (1) overly strict and (2) insufficiently tractable in our setting.

The classic synthesis query represents an all-or-nothing approach to tactic generation: if there is no well-formed program that exhibits *all* behaviors specified by a plan S, then no rule is generated even when there are programs that implement desirable subsets of S. For example, consider the specification $\langle S, src, tgt\rangle$ where S is $[\langle \mathbf{A}, [1], \beta_0\rangle, \langle \mathbf{A}, [2], \beta_0\rangle]$, src is $(+ (+ 0\ x) (+ 0\ y))$, tgt is $(+\ x\ y)$, $\mathbf{A}$ is the additive identity axiom (Figure 1a), and β_0 is the identity binding. The plan S specifies a general tactic for transforming a term of the form $(op (+ 0\ e_0) (+ 0\ e_1))$ to the term $(op\ e_0\ e_1)$, where op is any binary operator in our DSL. Such a tactic cannot be expressed as a well-formed program (Definition 3.6), since **Term** patterns cannot abstract over the operator, and using _ for the pattern makes it impossible to reference the first and second children of the matched term. But many useful specializations of this tactic are expressible, including the following generic rule for transforming *src* to *tgt*:

```
(Rule (Condition (Pattern (Term + (Term + (ConstTerm) _) (Term + (ConstTerm) _) etc))
                 (Constraint (And (Eq? (Ref 1 1) 0) (Eq? (Ref 2 1) 0))))
      (Action (Replace (Ref 1) (Ref 1 2)) (Replace (Ref 2) (Ref 2 2))))
```

Since we aim to generate a large set of useful rules 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 11), 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.

The Best-Implements Synthesis Query. To address the challenges of classic synthesis, we reformulate the synthesis task for RuleSy as follows: given $\langle S, src, tgt\rangle$, 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 $\langle src, tgt\rangle$ (Definition 4.9), and we search for them using the FindRules algorithm shown in Figure 9. Intuitively, FindRules generates rules that include as many transformations from $[\![S]\!]$ as possible (according to each rule's pattern), while always including the transformation $\langle src, tgt\rangle$ that shortens the solution to at least one example problem. We present FindRules next, highlighting how the best-implements query enables both sound and complete verification of candidate programs as well as efficient exploration of the candidate space.

Definition 4.9 (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 *best implements* S *for* $\langle src, tgt\rangle$ if $tgt \in fire(R, src)$ and $\forall t.[\![pattern(R)]\!]t \implies [\![R]\!]t = [\![S]\!]t$.

```
1:  function FindRules(S: plan, src, tgt: terms, k̄: ints)
2:    idx ← replayIndex(S, src, tgt)
3:    s, t ← ref(src, idx), ⟦S⟧ref(src, idx)
4:    p0 ← termToPattern(s)                          ▷ Most refined pattern that matches s
5:    R ← ⋃_{p0 ⊑ p} FindRule(p, S, s, t, k̄)
6:    return R                                       ▷ Rules that best implement S for ⟨src, tgt⟩

7:  function FindRule(p: pattern, S: plan, s, t: terms, k̄: ints)   ▷ ⟦p⟧s ∧ t=⟦S⟧s
8:    ??_c ← WellFormedConstraintHole(p, k̄)
9:    C ← (Condition (Pattern p) (Constraint ??_c))   ▷ Condition sketch
10:   ??̄_a ← WellFormedCommandHoles(p, k̄)
11:   A ← (Action ??̄_a)                              ▷ Action sketch with a sequence ??̄_a of holes
12:   𝕋 ← {t | ⟦p⟧t}                                 ▷ Symbolic representation of all terms that satisfy p
13:   c ← CEGIS(⟦C⟧s ∧ (∀τ ∈ 𝕋. ⟦C⟧τ ⟺ ⟦S⟧τ ≠ ⊥))
14:   a ← CEGIS(⟦A⟧s = t ∧ (∀τ ∈ 𝕋. ⟦S⟧τ ≠ ⊥ ⟹ ⟦A⟧τ = ⟦S⟧τ))
15:   return {(Rule c a) | c ≠ ⊥ ∧ a ≠ ⊥}
```

Fig. 9. The rule synthesis algorithm FindRules takes as input a bound $\bar{k}$ on the size of rule programs 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* (Definition 4.9).

Sound and Complete Verification. Verifying that a program R best implements a plan S involves checking that R produces the same output as S on all terms t accepted by R's pattern. The verification task is therefore to decide the validity of the formula $\forall t.[\![pattern(R)]\!]t \implies [\![R]\!]t = [\![S]\!]t$. We do so by observing that this formula has a small model property when R is well-formed (Definition 3.6): if the formula is valid on a carefully constructed finite set of terms $\mathbb{T}$, then it is valid on all terms. 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 (**VarTerm**) because all terms that satisfy (**VarTerm**) are isomorphic to the variable x up to a renaming.[3] Encoding the set $\mathbb{T}$ symbolically (rather than explicitly) enables FindRules to discharge its verification task efficiently with an off-the-shelf SMT solver [26].

Efficient Search. To accelerate the search for best implementations of a specification $\langle S, src, tgt\rangle$, we observe that all such implementations must fire on *src* to produce *tgt*, which has two key consequences. First, because S replays a path from *src* to *tgt* (by Theorem 4.8), *src* contains a subterm s at an index *idx* such that $t = [\![S]\!]s$ and $tgt = replace(src, idx, t)$ (lines 2-3). Any rule R that outputs t on s will therefore fire on *src* to produce *tgt*, so it sufficient to look for rules R that transform s to t, without having to reason about the semantics of *fire*. Second, if a rule accepts s, its pattern must be refined (Definition 4.10) by the most specific pattern p_0 (line 4) that accepts s. To construct p_0, we replace each literal in s with (**ConstTerm**), variable with (**VarTerm**), and operator o with the tokens **Term** o. Since p_0 refines finitely many patterns p, we can enumerate all of them (line 5). Once p is fixed through enumeration, FindRule can then efficiently search for a best implementation R with that pattern, by using an off-the-shelf synthesizer [38] to perform two independent searches for R's condition (line 13) and action (line 14). These two searches explore an exponentially smaller search space than a direct search for R within the generic sketch (**Rule** (**Condition** (**Pattern** p) (**Constraint** $??_c$)) (**Action** $\overline{??}_a$)), without missing any correct rules (Theorem 4.11). In particular, 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 itself 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 4.10 (Pattern Refinement). A condition pattern p_1 *refines* a pattern p_2 if $p_1 \sqsubseteq p_2$, where $\sqsubseteq$ is defined as follows: $p \sqsubseteq p$; $p \sqsubseteq \_$; (**ConstTerm**) $\sqsubseteq$ (**BaseTerm**); (**VarTerm**) $\sqsubseteq$ (**BaseTerm**);

[3] See Appendix A for the construction of the set $\mathbb{T}$ for other patterns.

```
1: function Optimize(T: set of terms, A,T: set of rules, f: objective)
2:   G_{A∪T} ← {}
3:   for t ∈ T such that ¬Reduced(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 ← ⋃_{î∈N∧Reduced(î)} allPaths(t, î, ⟨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 ⟨∅, ∅⟩
```

Fig. 10. Outline of the RuleSy optimization algorithm. The algorithm takes as input a set of terms T, axioms $\mathcal{A}$ for reducing T, macros $\mathcal{T}$ 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.

(**Term** o $p_1 \ldots p_k$) $\sqsubseteq$ (**Term** o $q_1 \ldots q_k$) if $p_i \sqsubseteq q_i$ for all $i \in [1..k]$; and (**Term** o $p_1 \ldots p_n$) $\sqsubseteq$ (**Term** o $q_1 \ldots q_k$ **etc**) if $n \geq k$ and $p_i \sqsubseteq q_i$ for all $i \in [1..k]$.

Characterization. The FindRules algorithm is a sound and complete procedure for synthesizing rules of size $\bar{k}$ that best implement a specification $\langle S, src, tgt \rangle$ (Theorem 4.11). Each rule returned by FindRules best implements S for $\langle src, tgt \rangle$, and the result set includes a sound rule R of size $\bar{k}$ if one exists. Composing these properties with those of FindSpecs, we find that all rules produced by RuleSy are sound with respect to the input axioms, and each shortens the solution to at least one input problem (Theorem 4.12).

Theorem 4.11. *Let S be an execution plan that replays a shortcut path from src to 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.*

Theorem 4.12. *Let T be a set of terms, $\mathcal{A}$ a set of rules that can Solve each term in T, and $\bar{k}$ a bound on the size of rule programs. Let $\mathcal{T} = \bigcup_{\langle S, src, tgt \rangle \in \mathcal{S}}$ FindRules$(S, src, tgt, \bar{k})$ where $\mathcal{S} =$ FindSpecs$(T, \mathcal{A})$. For every rule $R \in \mathcal{T}$, $[\![R]\!]$ is sound with respect to $\mathcal{A}$, and the longest path in Solve$(t, \mathcal{A} \cup \{R\})$ is shorter than the longest path in Solve$(t, \mathcal{A})$ for some term $t \in T$.*

4.3 Rule Set Optimization

After synthesizing tactic rules $\mathcal{T}$ 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 10) 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 4.13). 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$(t, \mathcal{R})$. The ability to perform this check without invoking Solve is crucial, since Solve$(t, \mathcal{R})$ may not terminate for an arbitrary term t and rule set $\mathcal{R}$ in our DSL.[4]

[4]To see why, note that breadth-first search would run forever trying to reduce the term $(= \; x \; (+ \, 1 \, 2))$ with just the rule $e \rightarrow (* \, 1 \, e)$.

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 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 [26]. For other objectives, we use a greedy algorithm to find a locally minimal solution (thus weakening the optimality guarantee in Theorem 4.13).

Theorem 4.13. *Let T, $\mathcal{A}$, and $\mathcal{T}$ be sets of terms and rules as defined in Theorem 4.12, and f 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 $\mathcal{R}' \subseteq \mathcal{A} \cup \mathcal{T}$, $f(\mathcal{R}, \{\text{Solve}(t, \mathcal{R}) \mid t \in T\}) \leq f(\mathcal{R}', \{\text{Solve}(t, \mathcal{R}') \mid t \in T\})$.*

5 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. Does RuleSy significantly outperform RuleSynth, a prior tool by Butler et al. [7]?
RQ 4. 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 gold standard—a textbook [8] written by domain experts. The third question evaluates the performance of RuleSy's algorithms by comparison to an existing tool for synthesizing tactics and domain models. The fourth question assesses the generality of our approach.

Our evaluation takes the form of two case studies. We address the first three questions by applying RuleSy to the domain of K-12 algebra. The algebra case study answers all questions positively and shows that RuleSy is an order-of-magnitude more efficient than RuleSynth [7], which is specialized to the domain of K-12 algebra. We address the fourth question by applying RuleSy to the domain of semantic proofs for propositional logic. This case study demonstrates that RuleSy extends to other domains.

5.1 Case Study with Algebra (RQ 1–3)

We assessed the quality of RuleSy's output and the performance of its algorithms by conducting three sets of experiments in the domain of K-12 algebra, designed to answer RQ 1-3. Each experiment was executed on an Intel 2nd 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 of our experiments are presented in the remainder of this section.

Table 1. The problems used for the algebra case study.

ID	Source	# Problems
P_R	RuleSynth [7]	55
P_T	Chapter 2, Sections 1-4 of Charles et al. [8]	92

Table 2. The axioms used for the algebra case study.

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	$\frac{x}{3} = 2 \rightarrow 3\left(\frac{x}{3}\right) = 2 * 3$

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 1) and axioms (Table 2) from a standard algebra textbook [8], and compare its output (607 tactics) to the tactics from the same textbook. Since the book demonstrates rules on examples rather than representing them explicitly, determining which rules are shown involves some amount of interpretation. For example, the book demonstrates the transformation of the term $5x + 2 - 2x = 2x + 14 - 2x$ to the term $3x + 2 = 14$ under the description "Simplify." We interpret this as demonstrating two independent tactics, one for each side of the equation, rather than one tactic with unrelated subparts. The second column of Table 3 lists all the tactics presented in the book, and we find that RuleSy recovers each of them or a close variation.

In addition to recovering textbook tactics, RuleSy also finds interesting variations on rules commonly taught in algebra class. Figure 11 shows an example of such a rule, which isolates a variable from a negated fraction and an addend. This rule composes 9 axioms, demonstrating RuleSy's ability to discover advanced new tactics.

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 to find a progression of optimal domain models for the problems (P_T in Table 1) and axioms (Table 2) in Charles et al. [8], 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 Charles et al. [8]. 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 1d 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 3 shows the resulting progressions of optimal domain models for Charles et al. [8], along with the rules that are introduced in the corresponding sections. For each rule presented in a 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.

```
; Isolate a variable from a negated
; fraction and an addend:
; (= (+(-(/(* x ...)b))c)e) → (= (* x ...)(*b(-ce)))
(define MBALNGOHG
  (Rule
    (Condition
      (Pattern
        (Term =
          (Term +
             (Term -
               (Term / (Term * (VarTerm) etc)
               (BaseTerm)))
             (ConstTerm))
          _))
      (Constraint true))
    (Action
      (Replace (Ref 1) (Ref 1 1 1 1))
      (Replace (Ref 2)
        (Make * (Ref 1 1 1 2)
                (Make - (Ref 1 2) (Ref 2)))))))
```

Fig. 11. A custom algebra tactic discovered by RuleSy.

```
; Modus ponens: if I ⊨ A → B and I ⊨ A,
;   then I ⊨ B.
(define xpq
  (Rule
    (Condition
      (Pattern
        (Term known
              (Term ⊨ (Term → _ _))
              (Term ⊨ _) etc))
      (Constraint
        (Eq? (Ref 1 1 1) (Ref 2 1))))
    (Action
      (Replace
        (Ref)
        (Cons (Make ⊨ (Ref 1 1 2))
              (Ref))))))
```

Fig. 12. A proof tactic synthesized by RuleSy.

This result demonstrates that RuleSy can recover textbook domain models, as well as find new models that optimize different objectives.

Table 3. A textbook [8] progression, along with optimal domain models found by RuleSy for the corresponding exercise problems, using for 3 different settings of α (Figure 1d). Row i shows the rules that the i^{th} model adds to the preceding models.

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**, **LNG**, **NG**, **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**

Comparison to Prior Work (RQ 3). We compare the performance of RuleSy to the prior system RuleSynth by applying both tools to the example problems P_R in Table 1 and the axioms in Table 2. We use the same problems as the original evaluation of RuleSynth because its algorithms encounter performance problems on the (larger) textbook problems P_T. We instead compare to its original evaluated output. Given these inputs, RuleSy synthesizes 144 tactics, which include the 13 rules synthesized by RuleSynth.[5] Figure 13 graphs the number of rules per minute produced by each system, which accounts for the time to mine specifications and synthesize rules for those specifications. These results show that our system both learns more rules and does so at a faster rate.

RuleSy outperforms RuleSynth thanks to the soundness and completeness of its specification mining and synthesis algorithms. RuleSynth employs a heuristic four-step procedure for synthesizing tactics: (1) use the axioms to solve the example problems; (2) extract pairs of input-output terms for all axiom sequences that appear in the solutions; (3) heuristically group those pairs into sets that are likely to be specifying the same tactics; and (4) synthesize a tactic for each resulting

[5]In particular, RuleSy synthesizes 13 programs that are semantically equivalent to those found by RuleSynth.

set. This process is neither sound nor complete, so RuleSynth can produce incorrect tactics and fail to identify tactic specifications found by RuleSy.

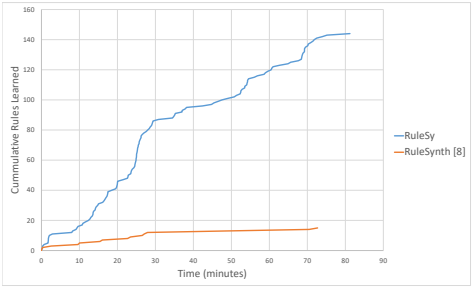


Fig. 13. The number of rules synthesized by RuleSy (blue) and RuleSynth (orange) over time on the same inputs. RuleSy learns 10× more rules at a quicker rate.

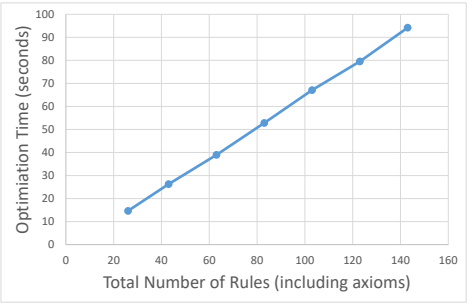


Fig. 14. Optimizer performance on design spaces derived from P_R of various sizes, using the objective in Figure 1d with $\alpha = .125$.

To show that RuleSy can efficiently explore spaces of rules to find optimal domain models, we compare its runtime performance to that of RuleSynth. Since the two systems use different input languages, we manually transcribed the 13 tactics generated by RuleSynth into our DSL. Given these tactics, the axioms in Table 2, and the examples P_R, RuleSynth finds an optimal rule set in 20 seconds (for the objective tradeoff parameter $\alpha = .4$), whereas RuleSy takes 14 seconds. As the optimization is superlinear in the number of rules, we can expect this performance difference to be magnified on larger rule sets. Figure 14 shows that RuleSy's optimization algorithm finds domain models quickly, even on much larger design spaces.

5.2 Case Study with Propositional Logic (RQ 4)

To evaluate the extensibility and generality of RuleSy, we applied it to the domain of semantic proofs for elementary propositional logic theorems. Many students have trouble learning how to construct proofs [13], so custom educational tools could help by teaching a variety of proof strategies.

Table 4. The axioms [6] used for the logic case study.

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 $I \nvDash B$

To conduct this case study, we instantiated RuleSy with a DSL for expressing semantic proofs. The DSL represents problem states as proof trees. Such a tree consists of a set of branches, each of which contains a set of facts that have been proven so far. The DSL encodes this proof structure with

commutative operators `branch` and `known`. The problem-solving task in this domain is to establish the validity of a propositional formula, such as $(p \wedge q) \rightarrow (p \rightarrow q)$, by assuming a falsifying interpretation and applying proof rules to arrive at a contradiction in every branch. Tactic rules apply multiple proof steps (i.e., axioms) at once. One of the paper's authors took two days to implement this DSL.

We applied the propositional instantiation of RuleSy to the axioms (Table 4) and proof exercises (3 in total) from the textbook by Bradley and Manna [6]. The system synthesized a total of 85 rules in 72 minutes. The resulting design space includes interesting general proof rules for each of the exercises. For example, given the problem $(p \wedge (p \rightarrow q)) \rightarrow q$, RuleSy automatically mines and synthesizes the modus ponens tactic shown in Figure 12. These results show that RuleSy's applicability and effectiveness extend beyond the domain of K-12 algebra.

6 RELATED WORK

Automated rule learning. Automated rule learning is a broad and 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 [18]. In particular, RuleSy learns macro rules from basic axioms, which is similar to chunking in SOAR [17], knowledge compilation in ACT [4], and macro-learning from AI planning [16]. But unlike these systems, RuleSy relies on program synthesis over a DSL as its primary rule learning mechanism. This design choice enables our framework to efficiently learn rules for transforming problems represented as trees, and to express domain-specific objective criteria over rules and solutions.

Inductive logic programming in education. Within educational technology, researchers have investigated automating various aspects of development, including learning rules and domain models for intelligent tutors [15]. Previous efforts have focused on using inductive logic programming (ILP) to learn a set of rules from a small set of expert-provided solution traces [14, 21, 22, 25, 32]. These efforts assume that the provided solution traces fully specify the desired domain model. RuleSy, in contrast, uses a small set of axioms and example problems to synthesize a large and diverse set of sound tactic rules. These rules, together with the basic axioms, define a design space of domain models, which can be automatically searched for a rule set that optimizes a desired educational objective.

Program synthesis in education. Prior educational applications of program synthesis and automated search include problem and solution generation [2, 12], hint and feedback generation [19, 33, 37], and checking of student proofs [20]. RuleSy solves a different problem—that of generating condition-action rules and domain models. General approaches to programming-by-example [23, 30] have investigated the problem of learning useful programs from a small number of training examples. One approach [30] produces a set of programs for the same input-output examples and chooses the best program based on ranking criteria; another [23] takes as input a set of examples that demonstrate different functions, and employs a prior to learn a program for each set. RuleSy, in contrast, relies on axioms to define program correctness, and it uses examples to bias the search toward useful programs (i.e., tactic rules that shorten solutions).

Term rewrite systems. RuleSy helps automate the construction of rule-based domain models, which are closely related to term rewrite systems [10]. Our work can be seen as an approach for learning compound rewrite rules from axioms, and then 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 [5, 28, 35]. Our DSL is designed to support effective automated reasoning by reduction to the quantifier-free theory of bitvectors.

7 CONCLUSION

This paper presented RuleSy, a framework for computer-aided development of custom domain models for educational applications. RuleSy helps create optimal domain models that are expressed as condition-action rules. Given a set of example problems and axiom rules for solving them, RuleSy synthesizes a set of tactic rules that combine the axioms into more efficient problem-solving strategies. The resulting tactics, together with the input axioms, form a design space of domain models. RuleSy searches this space for a model that both solves the example problems and optimizes a desired educational objective. Thanks to its new algorithms for specification mining, rule synthesis, and domain model optimization, RuleSy efficiently recovers textbook tactics 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 has the potential to help tool developers rapidly create a variety of custom domain models that target individual students' educational goals, skills, and traits.

A APPENDIX

Solving. FindSpecs (Figure 8) uses the Solve procedure (below) to find all shortest sequences of the axioms $\mathcal{R}$ that transform a term t to a reduced term $\hat{t}$. Solve finds these sequences by performing breadth-first search (lines 4–9). Each iteration of the search applies the rules $\mathcal{R}$ to the terms created in the previous iteration. The variable N stores the terms created so far; E stores the edges between those terms so that $\langle src, tgt\rangle_R \in E$ if some iteration of the search uses the rule R to transform *src* to *tgt*. If the search terminates, Solve constructs a solution graph (Definition 4.3) consisting of all shortest paths in $\langle N, E\rangle$ from t to a reduced term $\hat{t}$ (lines 10–13). This graph represents all shortest solutions to t that can be obtained using the rules $\mathcal{R}$.

```
1:  function Solve(t: term, R: set of well-formed programs)
2:    N ← {t}                          ▷ Set of terms reachable from t via the rules in R
3:    E ← {}                           ▷ Edges from N to N, labeled with rules from R
4:    while (∀n ∈ N. ¬Reduced(n)) do
5:      for all src ∈ {n ∈ N | ∀e ∈ E. n ≠ source(e)} do
6:        for all R ∈ R do                                  ▷ Apply all rules to src
7:          for all tgt ∈ fire(R, src) do
8:            N ← N ∪ {tgt}
9:            E ← E ∪ {⟨src, tgt⟩_R}
10:   paths ← ⋃_{t̂∈N∧Reduced(t̂)} allShortestPaths(t, t̂, ⟨N, E⟩)
11:   E ← ⋃_{p∈paths} pathEdges(p)          ▷ Edges comprising the shortest paths
12:   N ← {t} ∪ {n | ∃e ∈ E. source(e) = n ∨ target(e) = n}
13:   return ⟨N, E⟩                        ▷ Solution graph with all shortest solutions to t
```

Lemma A.1. *If a term t can be transformed to a Reduced form $\hat{t}$ by applying the rules $\mathcal{R}$, then 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 4–9). 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 10. Lines 11–12 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 13 therefore satisfies the definition of a solution graph, completing the proof. □

Lemma A.2. *Given a path p in a solution graph, FindPlan produces a general execution plan S that replays p.*

Proof. The proof consists of three parts. First, we show that $[\![\langle R_i, idx, \beta\rangle]\!] n_{i-1} = n_i$ at line 12. Because $\langle n_{i-1}, n_i\rangle_{R_i}$ is an edge in a solution graph, it follows from Definition 4.3 and the definition of *fire* (Figure 6) that line 11 is able to find an index *idx* and binding β such that n_i =

$replace(n_{i-1}, idx, [\![R_i]\!]ref(n_{i-1}, idx)^{\beta})$. As a result, the conditions in Definition 4.1 are satisfied, so $[\![\langle R_i, idx, \beta\rangle]\!]\ n_{i-1} = n_i$. Next, by induction on i and the definition of execution plans (Definition 4.2), we conclude that $[\![S]\!]n_0 = n_k$ at line 13 and S replays p (Definition 4.4). Finally, observe that *root* holds the greatest common prefix of the step indices $idx_1, \ldots, idx_k$ for the steps in S. Since the loop at lines 13–16 drops *root* from these indices, the plan S at line 17 is general (by Definition 4.2), and $n_k = replace(n_0, root, [\![S]\!]ref(n_0, root))$ so S replays p. □

Theorem 4.8. Let T be a set of terms, Reduced a predicate over terms, and $\mathcal{A}$ a set of rules. If every term in T can be Reduced 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 $\langle S, src, tgt\rangle \in \mathcal{S}$, $[\![S]\!]$ is sound with respect to $\mathcal{A}$, and (2) for every shortcut path p from src to tgt in a solution graph for $t \in T$, $\mathcal{A}$, and Reduced, there is a triple $\langle S, src, tgt\rangle \in \mathcal{S}$ such that S replays p.

Proof. Termination follows from Theorem A.1 and the fact that all loops in FindSpecs and FindPlan iterate over finite structures. Soundness (1) follows from line 6 of FindSpecs, Theorem A.2, and Definitions 4.1, 4.2, and 4.6. Completeness (2) follows from lines 3–6, Lemmas A.1–A.2, and Definitions 4.4 and 4.7. □

Lemma A.3. *Let p be a pattern, S be a plan, and s, t be terms such that $[\![p]\!]s$ and $t = [\![S]\!]s$. Also let* (**Rule** (**Condition** (**Pattern** p) (**Constraint** $??_c$)) (**Action** $\overline{??}_a$)) *be a sketch with well-formed holes of size $\bar{k}$. If this sketch includes a rule that best implements S for $\langle s, t\rangle$, 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} = (**Rule** C A) can be decomposed into two queries for the sketches C and A, where C is (**Condition** (**Pattern** p) (**Constraint** $??_c$)) and A is (**Action** $\overline{??}_a$). Then, we show that the CEGIS engine [38] invoked at lines 13–14 is sound and complete for these two queries. Hence, if R_{CA} contains a rule that best implements S for $\langle s, t\rangle$, 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 4.9 for S, s, and t, we pose the following synthesis query:

$$\exists ??_c, \overline{??}_a.\ t \in fire(R_{CA}, s) \wedge \forall \tau.[\![p]\!]\tau \implies [\![R_{CA}]\!]\tau = [\![S]\!]\tau \tag{1}$$

Using $[\![p]\!]s$ and $t = [\![S]\!]s$, we can simplify Equation 1 to

$$\exists ??_c, \overline{??}_a.\ t = [\![R_{CA}]\!]s \wedge \forall \tau.[\![p]\!]\tau \implies [\![R_{CA}]\!]\tau = [\![S]\!]\tau \tag{2}$$

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

$$(\textbf{if } [\![C]\!]\tau \textbf{ then } [\![A]\!]\tau \textbf{ else } \bot) = [\![S]\!]\tau \tag{3}$$

Equation 3 expands into the following formulas:

$$([\![C]\!]\tau \implies ([\![A]\!]\tau \neq \bot \wedge [\![A]\!]\tau = [\![S]\!]\tau)) \wedge (\neg[\![C]\!]\tau \implies [\![S]\!]\tau = \bot) \tag{4}$$

$$([\![C]\!]\tau \implies ([\![S]\!]\tau \neq \bot \wedge [\![A]\!]\tau = [\![S]\!]\tau)) \wedge ([\![S]\!]\tau \neq \bot \implies [\![C]\!]\tau) \tag{5}$$

$$([\![C]\!]\tau \implies [\![A]\!]\tau = [\![S]\!]\tau) \wedge ([\![C]\!]\tau \iff [\![S]\!]\tau \neq \bot) \tag{6}$$

$$([\![S]\!]\tau \neq \bot \implies [\![A]\!]\tau = [\![S]\!]\tau) \wedge ([\![C]\!]\tau \iff [\![S]\!]\tau \neq \bot) \tag{7}$$

After substituting (7) into (2), 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

$$\begin{aligned} \exists ??_c, \overline{??}_a.&[\![C]\!]s \wedge t = [\![A]\!]s \wedge \\ &\forall \tau \in \mathbb{T}.([\![S]\!]\tau \neq \bot \implies [\![A]\!]\tau = [\![S]\!]\tau) \wedge \\ &\forall \tau \in \mathbb{T}.([\![C]\!]\tau \iff [\![S]\!]\tau \neq \bot) \end{aligned} \tag{8}$$

Since $??_c$ occurs only in C and $\overline{??}_a$ only in A, we can rewrite Equation 8 into the following separately solvable conjuncts:

$$(\exists ??_c.\, [\![C]\!]s \wedge \forall \tau \in \mathbb{T}.\, [\![C]\!]\tau \iff [\![S]\!]\tau \neq \bot) \wedge \tag{9}$$

$$(\exists \overline{??}_a.\, t = [\![A]\!]s \wedge \forall \tau \in \mathbb{T}.\, [\![S]\!]\tau \neq \bot \implies [\![A]\!]\tau = [\![S]\!]\tau) \tag{10}$$

These are the formulas solved at lines 13–14 of FindRule.

CEGIS. FindRule uses an off-the-shelf CEGIS engine [38] to solve the synthesis queries at lines 13–14. 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 9–10 without affecting their satisfiability.

Case 1. p includes no `_` or `etc` tokens. Let v be the number of (`VarTerm`) and (`BaseTerm`) patterns in p, and V be a set of v fresh uninterpreted constants. Create a term t_p by removing the token `Term` from all patterns in p, and replacing each remaining pattern in t_p 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 $scope(p, idx) =$ (`ConstTerm`), V if $scope(p, idx) =$ (`VarTerm`), and $\mathbb{Z}_n \cup V$ if $scope(p, idx) =$ (`BaseTerm`). 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 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 compare them for (in)equality, and the action can use them as atomic components (in `Make`).

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 4.11. Let S be an execution plan that replays a shortcut path from src to 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. Because S replays a path from src to tgt in a solution graph (Definition 4.4), there is an index idx such that $tgt = replace(src, idx, [\![S]\!]ref(src, idx))$. Due to acylicity of solution graphs (Definition 4.3), $src \neq tgt$, and hence, idx is uniquely defined at line 2. As a result, all rules that best implement S for $\langle s, t\rangle$ at line 3 are the only rules that best implement S for $\langle src, tgt\rangle$. Line 4 defines p_0 to be the most refined pattern that matches s, so by Definition 4.10 and the semantics of patterns, p_0 refines the pattern of every rule applicable to s. Since there are finitely many such patterns (by Definition 4.10), termination, soundness (1), and completeness (2-3) follow from line 5 and Theorem A.3. □

Theorem 4.12. Let T be a set of terms, $\mathcal{A}$ a set of rules that can Solve each term in T, and $\bar{k}$ a bound on the size of rule programs. Let $\mathcal{T} = \bigcup_{\langle S, src, tgt\rangle \in \mathcal{S}}$ FindRules$(S, src, tgt, \bar{k})$ where

$\mathcal{S}$ = FindSpecs($T, \mathcal{A}$). For every rule $R \in \mathcal{T}$, $[\![R]\!]$ is sound with respect to $\mathcal{A}$, and the longest path in Solve($t, \mathcal{A} \cup \{R\}$) is shorter than the longest path in Solve($t, \mathcal{A}$) for some term $t \in T$.

Proof. The proof follows from Theorems 4.8-4.11, Definition 4.6, Definition 4.9, and semantics of rule programs. □

Theorem 4.13. Let T, $\mathcal{A}$, and $\mathcal{T}$ be sets of terms and rules as defined in Theorem 4.12, and f 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 $\mathcal{T}$ (Theorem 4.12), we can show that for every $\mathcal{R} \in \mathbb{R}$, 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-Reduced term $t \in T$, $\mathcal{G}_{\mathcal{A} \cup \mathcal{T}}$ contains a solution graph G consisting of Solve($t, \mathcal{R}$) for all $\mathcal{R} \in \mathbb{R}$. As a result, Restrict($G, \mathcal{R}$) returns Solve($t, \mathcal{R}$) if $\mathcal{R} \in \mathbb{R}$ and the empty graph otherwise. By line 7, $f_{\emptyset}(\mathcal{R}, \{\text{Restrict}(G, \mathcal{R}) \mid G \in \mathcal{G}_{\mathcal{A} \cup \mathcal{T}}\})$ is equal to $f(\mathcal{R}, \{\text{Solve}(t, \mathcal{R}) \mid t \in T\})$ for $\mathcal{R} \in \mathbb{R}$, and it is infinite otherwise. Consequently, the minimization operation at line 8 selects a cheapest set $\mathcal{R} \in \mathbb{R}$. □

REFERENCES

[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. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

[2] Erik Andersen, Sumit Gulwani, and Zoran Popović. 2013. A Trace-based Framework for Analyzing and Synthesizing Educational Progressions. In *CHI*.

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

[4] John R Anderson and Christian Lebiere. 1998. The Atomic Components of Thought. (1998).

[5] Clark Barrett, Igor Shikanian, and Cesare Tinelli. 2007. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic Notes in Theoretical Computer Science* 174, 8 (2007), 23–37.

[6] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[7] Eric Butler, Emina Torlak, and Zoran Popović. 2016. A Framework for Parameterized Design of Rule Systems Applied to Algebra. In *Intelligent Tutoring Systems*. Springer.

[8] Randall I. Charles, Basia Hall, Dan Kennedy, Allan E. Bellman, Sadie Chavis Bragg, William G. Handlin, Stuart J. Murphy, and Grant Wiggins. 2012. *Algebra 1: Common Core*. Pearson Education, Inc.

[9] Jennifer Demski. 2012. 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 (2012), 32.

[10] Nachum Dershowitz and Jean-Pierre Jouannaud. 1989. *Rewrite systems*. Citeseer.

[11] Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Handbook of Theoretical Computer Science (Vol. B). MIT Press, Cambridge, MA, USA, Chapter Rewrite Systems, 243–320. http://dl.acm.org/citation.cfm?id=114891.114897

[12] Sumit Gulwani. 2014. Example-based learning in computer-aided stem education. *Commun. ACM* 57, 8 (2014), 70–80.

[13] Guershon Harel and Larry Sowder. 2007. Toward comprehensive perspectives on the learning and teaching of proof. *Second handbook of research on mathematics teaching and learning* 2 (2007), 805–842.

[14] Matthew P Jarvis, Goss Nuzzo-Jones, and Neil T Heffernan. 2004. Applying machine learning techniques to rule generation in intelligent tutoring systems. In *Intelligent Tutoring Systems*. Springer, 541–553.

[15] Kenneth R Koedinger, Emma Brunskill, Ryan SJd Baker, Elizabeth A McLaughlin, and John Stamper. 2013. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine* 34, 3 (2013), 27–41.

[16] Richard E. Korf. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26, 1 (1985), 35 – 77. https://doi.org/10.1016/0004-3702(85)90012-8

[17] John E. Laird, Allen Newell, and Paul S. Rosenbloom. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33, 1 (1987), 1 – 64.

[18] Pat Langley, John E Laird, and Seth Rogers. 2009. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research* 10, 2 (2009), 141–160.
[19] Timotej Lazar and Ivan Bratko. 2014. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Intelligent Tutoring Systems*. Springer, 306–311.
[20] Colleen Lee. 2012. DeduceIt: a tool for representing and evaluating student derivations. Stanford Digital Repository: http://purl.stanford.edu/bg823wn2892. (2012).
[21] Nan Li, William Cohen, Kenneth R Koedinger, and Noboru Matsuda. 2010. A machine learning approach for automatic student model discovery. In *Educational Data Mining 2011*.
[22] Nan Li, Abraham J Schreiber, WW Cohen, and KR Koedinger. 2012. Efficient complex skill acquisition through representation learning. *Advances in Cognitive Systems* 2 (2012).
[23] Percy Liang, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 639–646.
[24] Yun-En Liu, Christy Ballweber, Eleanor O'rourke, Eric Butler, Phonraphee Thummaphan, and Zoran Popović. 2015. Large-Scale Educational Campaigns. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 8 (March 2015), 24 pages. https://doi.org/10.1145/2699760
[25] Noboru Matsuda, William W Cohen, and Kenneth R Koedinger. 2005. 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)*. 1–8.
[26] Leonardo Moura and Nikolaj Bjørner. 2008. *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*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Z3: An Efficient SMT Solver, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[27] Tom Murray. 1999. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education* 10 (1999).
[28] Derek C Oppen. 1978. Reasoning about recursively defined data structures. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 151–157.
[29] Eleanor O'Rourke, Erik Andersen, Sumit Gulwani, and Zoran Popović. 2015. A Framework for Automatically Generating Interactive Instructional Scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1545–1554.
[30] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Inter. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 107–126.
[31] Sam Redding. 2013. Getting personal: The promise of personalized learning. *Handbook on innovations in learning* (2013), 113–130.
[32] Ute Schmid and Emanuel Kitzelmann. 2011. Inductive rule learning on the knowledge level. *Cognitive Systems Research* 12, 3 (2011), 237–248.
[33] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 12.
[34] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 12. https://doi.org/10.1145/1168857.1168907
[35] Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision procedures for algebraic data types with abstractions. *Acm Sigplan Notices* 45, 1 (2010), 199–210.
[36] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.
[37] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. 2014. Constructing coding duels in Pex4Fun and Code Hunt. In *ISSTA*. ACM, 445–448.
[38] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 12.
[39] Kurt VanLehn. 1990. *Mind bugs: The origins of procedural misconceptions*. MIT press.
[40] Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist* 46, 4 (2011), 197–221.