A Framework for Computer-Aided Design of Educational Domain Models

Eric Butler, Emina Torlak, and Zoran Popović

University of Washington
{edbutler,emina,zoran}@cs.washington.edu

Abstract. Many educational applications, from tutoring to problem generation, are built on a formal model of the operational knowledge for a given domain. These *domain models* consist of rewrite rules that experts apply to solve problems in the domain; e.g., factoring, $ax+bx \rightarrow (a+b)x$, is one such rule for K-12 algebra. Domain models currently take hundreds of hours to create, and they differ widely in how well they meet educational objectives such as maximizing problem-solving efficiency. Rapid, objective-driven creation of domain models is a key challenge in the development of personalized educational tools.

This paper presents RuleSy, a new framework for computer-aided authoring of 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 sound *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 designing models for a variety of educational domains.

1 Introduction

A key challenge in the design of educational applications is modeling the operational knowledge that captures the expertise for a given domain. This knowledge takes the form of a *domain model*, which consists of *condition-action* rules that experts apply to solve problems in the domain. For example, factoring, $ax + bx \rightarrow (a + b)x$, is one such rule for K-12 algebra: its condition recognizes problem states that trigger rule application, and the action specifies the result. Educational applications rely on domain models to automate tasks such as

problem and progression generation [1], hint and feedback generation [2], student modeling [3], and misconception detection [4].

At present, domain models are created by hand, taking hundreds of hours of development time to model a single hour of instructional material [5]. This limits applications to using one out of many possible models that capture the operational knowledge for a domain. Yet recent research [6] shows that some students need over six times more content than others to master a domain. To best serve a broad population of students, applications therefore need multiple models that optimize different educational objectives [7,8].

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, $c_0 + c_1 \rightarrow c_2$, 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, $c_0x + c_1x \rightarrow c_2x$, which composes factoring and constant folding? While such compound rules, or *tactics*, are redundant with respect to the axioms, standard domain models (e.g., [9]) include them to enable efficient problem solving with fewer steps and less cognitive load [10]. But there is a limit to how many rules students can remember, so the optimal set of axioms and tactics depends on the desired tradeoff between maximizing solving efficiency and minimizing the memorization burden.

This paper presents a new approach for rapid, objective-driven creation of domain models that is based on program synthesis. We realize this approach in RuleSy, a framework that assists developers with creating tactics and domain models that optimize desired objectives. The RuleSy framework was motivated by practical experience: the first and last authors work for Enlearn, an educational technology company building adaptive K12 applications that need custom domain models. Developers of such applications are the intended users of this work, and Enlearn is in the process of adopting key ideas from RuleSy.

RuleSy aids developers by generating an optimal domain model given a set of axioms for the domain, a set of example problems, and an optimization objective expressed in terms of rule and solution costs. Using the axioms and the problems, RuleSy synthesizes an exhaustive set of tactics that combine multiple axioms into advanced problem-solving strategies. Each of these tactics is a sound rule that shortens the solution to at least one example problem compared to using the axioms alone. Following synthesis, RuleSy applies discrete optimization to produce a subset of the axioms and tactics (i.e., a domain model) that both solves the example problems and optimizes the given objective.

RuleSy's algorithms are designed to solve three core technical challenges:

- *Specification.* Synthesizing tactics requires a functional specification of their behavior. Since tactics compose multiple axioms, a sequence of axioms may seem to provide such a specification. For example, we may expect $\mathbf{A} \circ \mathbf{A}$, where $\mathbf{A}$ is the additive identity rule $x + 0 \rightarrow 0$, to describe the tactic $(x+0)+0 \rightarrow x$. But because a condition-action rule can fire on *any* part of the problem state, $\mathbf{A} \circ \mathbf{A}$ describes a set of distinct tactics that also includes, e.g., $x+0 = y+0 \rightarrow x = y$. RuleSy addresses this challenge with a new approach

for extracting tactic functions from the shortest axiom-based solutions to the example problems. Each resulting tactic specification is sound with respect to the axioms, and useful for solving at least one example in fewer steps.
- *Synthesis.* Given a tactic specification, the next challenge is to find a rule that implements it. RuleSy represents rules as programs that operate on problem states expressed as (abstract syntax) trees. Because these trees are unbounded in size, the rule synthesis query cannot be expressed in existing systems for syntax-guided synthesis (e.g., [11,12,13]). To address this challenge, RuleSy employs an efficient new reduction to a set of syntax-guided synthesis queries over (carefully constructed) trees of bounded size. Our reduction exploits the structure of RuleSy's specifications and programs to ensure that the synthesized rules are sound over trees of any size, and to asymptotically reduce the size of the synthesis search space.
- *Optimization.* The final challenge is to search the axioms and tactics for a domain model that both solves the examples and optimizes the input objective. Finding such a model is undecidable in general, since an arbitrary set of condition-action rules (i.e., a candidate model) may not be terminating [14]. RuleSy addresses this challenge with a new algorithm for deciding a more constrained variant of the model optimization task: it 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.

To evaluate our algorithms, we used RuleSy to model the domain of introductory K-12 algebra, comparing the output to a standard textbook [9] and a prior tool [15] for this domain. Applying RuleSy to examples and axioms from the textbook, we find that it both recovers the tactics presented in the book and discovers new ones. We also find that RuleSy can recover the textbook's domain model, as well as discover variants that optimize different objectives. Finally, we find that RuleSy significantly outperforms the prior tool, both in terms of output quality and runtime performance.

To show that RuleSy generalizes beyond K-12 algebra, we used it to model the domain of propositional logic proofs. Applying RuleSy to textbook [16] axioms and exercises, we find that it synthesizes both new and standard tactics for this domain (e.g., modus ponens), just as it did for K-12 algebra.

The rest of the paper is organized as follows. Section 2 illustrates RuleSy on a toy algebra domain. Section 3 describes RuleSy's language of condition-action rules. Section 4 describes the new algorithms for specification mining, rule synthesis, and model optimization. Section 5 presents our two case studies. Section 6 reviews related work, and Section 7 concludes.

2 Overview

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

2.1 Examples, Axioms, and Objectives

Figure 1 shows our example problems, axioms, and objective for toy algebra.

Examples. The problems (1b) are represented as syntax trees. We consider a tiny subset of algebra that includes equations of the form $x + \sum_i c_i = c_k$, where x is a variable and c_i, c_k are integer constants. Experts solve these problems by applying condition-action rules until they obtain an equation of the form $x = c$.

Axioms. The axioms (1a) are expressed as programs in the RuleSy language (Section 3). A rule program 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, the axiom **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 $(+\ e\ldots)$. The shown axioms can solve all problems in the toy algebra domain. For example, we can solve p_1 in two steps by applying $\mathbf{B} \circ \mathbf{A}$ to obtain $x + 1 + -1 = 5 \rightarrow_{\mathbf{B}} x + 0 = 5 \rightarrow_{\mathbf{A}} x = 5$. RuleSy uses the axioms to synthesize tactic rules (Figure 3) that can solve the example problems in fewer steps.

Objective. The educational objective (1c) is expressed as a function of rule and solution costs. Rule cost measures the complexity of a rule's syntactic representation. Solution cost measures the efficiency of a solution in terms of the tree edits performed to solve an example problem. These costs are proxy measures for the difficulty of learning and applying knowledge in a given domain model [10]. RuleSy selects a domain model that best balances the trade-off between rule complexity and solution efficiency specified by the objective.

2.2 Specifications, Tactics, and Domain Models

Specifications. To help with domain modeling, RuleSy first needs to synthesize a set of useful tactics, which can solve the input problems more efficiently than the axioms alone. For example, we could solve p_1 in one step if we had a "cancelling opposite constants" tactic that composes the axioms **B** and **A**. RuleSy determines which rules to synthesize, and how those rules should behave, by mining tactic specifications (Section 4.1) from the shortest solutions to the example problems that are obtainable with the axioms (Figure 2a).

A RuleSy specification takes the form of a *plan* for applying a sequence of axioms (Figure 2b). A plan describes which axioms to apply, in what order, and how. Since an axiom may be applied to a problem in multiple ways, a plan associates each axiom 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 specifies a generic tactic for cancelling opposite constants; for example, it solves p_1 in one step by reducing the expression $(+\ x\ 1\ -1)$ to x (Figure 2c). In essence, plans

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

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

(define C ; Adding opposite: (= (+ e0 ...) e1) → (= (+ (- e0) e0 ...) (+ e1 (- e0)))
  (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 in the RuleSy language (Section 3).

```
; Problem p0 : x + 0 = 3
(= (+ x 0) 3)

; Problem p1 : x + 1 + −1 = 5
(= (+ x 1 -1) 5)

; Problem p2 : x + 2 = −4
(= (+ x 2) -4)
```

(b) Example problems.

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

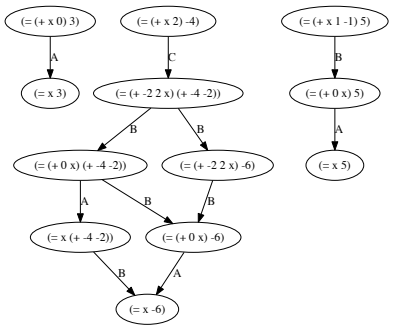
(c) A sample objective function, where $\alpha \in [0, 1]$, and $\mathcal{G}$ contains the shortest solutions obtained with the rules $\mathcal{R}$.

Fig. 1: The inputs to RuleSy for the toy algebra domain.

are functional specifications of tactics that can help solve the example problems in fewer steps—and that are amenable to sound and efficient synthesis.

Tactics. Given a set of plans, RuleSy synthesizes the corresponding tactics (Section 4.2), expressed in the same language (Section 3) as the input axioms. Figure 3 shows two sample tactics synthesized for the plans (e.g., Figure 2b) mined from the toy examples and axioms. These tactics perform fewer tree edits than the axiom sequences they replace, leading to cheaper solutions. For example, the tactic **BA** performs two tree edits, while the axiom sequence $\mathbf{B} \circ \mathbf{A}$ performs three such edits. But the tactic also applies to fewer problem states than the axioms. RuleSy uses discrete optimization, guided by the input objective, to decide which axioms and tactics to include in a domain model.

Domain Models. The RuleSy optimizer (Section 4.3) searches the axioms and tactics for a domain model that is sufficient to solve the example problems, while minimizing the objective over all shortest solutions obtainable with such models. Figure 4 shows two sample optimal models for the toy algebra domain. The models $\mathcal{R}_{0.1}$ and $\mathcal{R}_{0.9}$ minimize the toy objective (Figure 1c) for different values of the weighting factor α (0.1 and 0.9, respectively). The model $\mathcal{R}_{0.1}$ includes more rules because lower values of α emphasize solution efficiency over domain model economy. RuleSy helps with rapid navigation of such design tradeoffs.



(a) All shortest solutions for the toy algebra problems and axioms (Figure 1).

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

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

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

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

Fig. 2: A sample plan (b) mined from the shortest solutions (a) to the toy algebra problems. The plan specifies a tactic for canceling opposite constants (c).

```
; 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 toy plans (e.g., Figure 2b).

3 A Language for Condition-Action Rules

This section presents the RuleSy language for specifying condition-action rules. The language is parametric in its definition of problem states. For concreteness, we present an instantiation of RuleSy for the domain of K-12 algebra. We describe another instantiation, for the domain of propositional logic, in Section 5.

Specifying Problems and Rules. The RuleSy domain-specific language (DSL) for algebra represents rules as *programs* that operate on problems expressed as *terms* (Figure 5a). RuleSy is parametric in the definition of terms, but the structure of rules 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

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

Fig. 4: Optimal domain models for toy algebra (Figures 1 and 3).

$$
\begin{array}{rcl}
\mathit{program} & := & \texttt{(Rule } \mathit{cond}\ \mathit{action}\texttt{)} \\
\mathit{cond} & := & \texttt{(Condition (Pattern } \mathit{pattern}\texttt{)} \\
& := & \qquad\qquad \texttt{(Constraint } \mathit{constr}\texttt{))} \\
\mathit{pattern} & := & \texttt{\_} \mid \texttt{(ConstTerm)} \mid \texttt{(VarTerm)} \mid \texttt{(BaseTerm)} \\
& \mid & \texttt{(Term } \mathit{op}\ \mathit{pattern}^{+}\texttt{)} \\
& \mid & \texttt{(Term } \mathit{op}\ \mathit{pattern}^{+}\ \texttt{etc)} \\
\mathit{constr} & := & \texttt{true} \mid \mathit{pred} \mid \texttt{(And } \mathit{constr}\ \mathit{constr}\texttt{)} \\
\mathit{pred} & := & \texttt{(Eq? } \mathit{ref}\ \mathit{const}\texttt{)} \mid \texttt{(Neq? } \mathit{ref}\ \mathit{const}\texttt{)} \\
\mathit{action} & := & \texttt{(Action } \mathit{cmd}^{+}\texttt{)} \\
\mathit{cmd} & := & \texttt{(Remove } \mathit{ref}\texttt{)} \mid \texttt{(Replace } \mathit{ref}\ \mathit{expr}\texttt{)}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{expr} & := & \mathit{const} \mid \mathit{obj} \\
\mathit{obj} & := & \texttt{(Make } \mathit{op}\ \mathit{expr}^{+}\texttt{)} \\
& \mid & \texttt{(Cons } \mathit{expr}\ \mathit{ref}\texttt{)} \\
& \mid & \texttt{(Cons } \mathit{expr}\ \mathit{obj}\texttt{)} \\
\mathit{const} & := & \mathit{int} \mid \mathit{ref} \mid \texttt{(Apply } \mathit{op}\ \mathit{const}^{+}\texttt{)} \\
\mathit{ref} & := & \texttt{(Ref)} \mid \texttt{(Ref } \mathit{int}^{+}\texttt{)} \\
\mathit{term} & := & \mathit{int} \mid \mathit{var} \mid \texttt{(}\mathit{op}\ \mathit{term}^{+}\texttt{)} \\
\mathit{int} & := & \text{integer literal} \\
\mathit{var} & := & \text{identifier} \\
\mathit{op} & := & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=}
\end{array}
$$

(a) Syntax for the algebra DSL.

$$
\begin{array}{lll}
[\![\texttt{(Rule } c\ a\texttt{)}]\!]t & = \textbf{if } [\![c]\!]t \textbf{ then } [\![a]\!]t \textbf{ else } \bot & \\
[\![\texttt{(Condition } p\ b\texttt{)}]\!]t & = [\![p]\!]t \wedge [\![b]\!]t & \\
[\![\texttt{(Pattern } p\texttt{)}]\!]t & = [\![p]\!]t & \\
[\![\texttt{(Constraint } b\texttt{)}]\!]t & = [\![b]\!]t & \\
[\![\texttt{(Term } o\ p_1 \ldots p_k\texttt{)}]\!]t & = (t = (o\ t_1 \ldots t_k)) \wedge & \\
 & \quad \forall_{1 \le i \le k} [\![p_i]\!]t_i & \\
[\![\texttt{(Term } o\ p_1 \ldots p_k\ \texttt{etc)}]\!]t & = (t = (o\ t_1 \ldots t_n)) \wedge & \\
 & \quad n \ge k \wedge \forall_{1 \le i \le k} [\![p_i]\!]t_i & \\
[\![\texttt{(ConstTerm)}]\!]t & = \mathit{literal}(t) & \\
[\![\texttt{(VarTerm)}]\!]t & = \mathit{variable}(t) & \\
[\![\texttt{(BaseTerm)}]\!]t & = \mathit{literal}(t) \vee \mathit{variable}(t) &
\end{array}
$$

$$
\begin{array}{lll}
[\![\texttt{\_}]\!]t & = \text{true} & \\
[\![\texttt{true}]\!]t & = \text{true} & \\
[\![\texttt{(Eq? } r\ e\texttt{)}]\!]t & = ([\![r]\!]t = [\![e]\!]t) & \\
[\![\texttt{(Neq? } r\ e\texttt{)}]\!]t & = ([\![r]\!]t \neq [\![e]\!]t) & \\
[\![\texttt{(And } b_1\ b_2\texttt{)}]\!]t & = [\![b_1]\!]t \wedge [\![b_2]\!]t & \\
[\![\texttt{(Action } a_1 \ldots a_k\texttt{)}]\!]t & = ([\![a_1]\!] \parallel \ldots \parallel [\![a_k]\!])(t) & \\
[\![\texttt{(Remove } r\texttt{)}]\!]t & = \mathit{rm}(t, \mathit{index}(r)) & \\
[\![\texttt{(Replace } r\ e\texttt{)}]\!]t & = \mathit{replace}(t, \mathit{index}(r), [\![e]\!]t) & \\
[\![\texttt{(Make } o\ e_1 \ldots e_k\texttt{)}]\!]t & = (o\ [\![e_1]\!]t\ \ldots\ [\![e_k]\!]t) & \\
[\![\texttt{(Cons } e_1\ e_2\texttt{)}]\!]t & = \mathit{cons}([\![e_1]\!]t, [\![e_2]\!]t) & \\
[\![\texttt{(Apply } o\ e_1 \ldots e_k\texttt{)}]\!]t & = [\![o]\!]([\![e_1]\!]t, \ldots, [\![e_k]\!]t) & \\
[\![\texttt{(Ref } i_1 \ldots i_k\texttt{)}]\!]t & = \mathit{ref}(t, [i_1, \ldots, i_k]) &
\end{array}
$$

$$
\begin{array}{ll}
\mathit{index}(\texttt{(Ref } i_1 \ldots i_k\texttt{)}) & = [i_1, \ldots, i_k] \\
\mathit{replace}(t, [], s) & = s \\
\mathit{replace}((o\ t_1 \ldots t_k), [i], s) & = (o\ t_1\ \ldots\ t_{i-1}\ s\ t_{i+1}\ \ldots\ t_k) \\
\mathit{replace}((o\ t_1 \ldots t_k), [i, j, \ldots], s) & = (o\ t_1 \ldots \mathit{replace}(t_i, s, [j, \ldots]) \ldots t_k) \\
\mathit{rm}((o\ t_1 \ldots t_k), [i]) & = (o\ t_1\ \ldots\ t_{i-1}\ t_{i+1}\ \ldots\ t_k) \\
\mathit{rm}((o\ t_1 \ldots t_k), [i, j, \ldots]) & = (o\ t_1\ \ldots\ \mathit{rm}(t_i, [j, \ldots])\ \ldots\ t_k) \\
\mathit{cons}(t, (o\ t_1 \ldots t_k)) & = (o\ t\ t_1\ \ldots\ t_k)
\end{array}
$$

$$
\begin{array}{l}
\mathit{fire}(R, t) = \{[\![R]\!]t \mid [\![R]\!]t \neq \bot\} \textbf{ where } \mathit{literal}(t) \vee \mathit{variable}(t) \\
\mathit{fire}(R, t) = \{[\![R]\!]t^{\beta} \mid [\![R]\!]t^{\beta} \neq \bot \wedge \beta \in \mathcal{B}(\mathit{pattern}(R), t)\} \cup \bigcup_{1 \le i \le n} \{\mathit{replace}(t, [i], s) \mid s \in \mathit{fire}(R, t_i)\} \\
\qquad \textbf{where } t = (o\ t_1 \ldots t_n)
\end{array}
$$

(b) Semantics for the algebra DSL. The expression $(o\ t_1\ \ldots\ t_n)$ constructs a term with the given operator and children; $\parallel$ stands for parallel function composition; $[x, \ldots]$ is a sequence; and other notation is described in Definitions 1-4.

Fig. 5: Syntax (a) and semantics (b) for the RuleSy algebra DSL.

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.

Semantics of Rule Firing. Rule programs denote partial functions from terms to terms (Figure 5b). If a term satisfies the rule's condition, the result is a term; otherwise, the result is an undefined value ($\bot$). We solve problems by applying rules exhaustively, via $\mathit{fire}(R, t)$, on permutations and subterms of a given term. Permuting a term (Definition 2) reorders the arguments to any commutative operators while leaving the rest of the term's structure unchanged. To fire a rule on 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 4). 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.

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 1). Since both $+$ and $*$ are commutative, valid tree permutations $\Pi(t)$ for t consist of the following mappings (Definition 2):

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

Next, we observe that the scope (Definition 3) of **A**'s pattern consists 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 4): $\beta_0 = \{[] \mapsto [], [1] \mapsto [1], [2] \mapsto [2]\}$ 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 **A** applies only to t^{β_1}, so $\mathit{fire}(\mathbf{A}, t)$ yields $\{(*\ x\ 2)\}$.

Definition 1 (Tree Indices). *A* tree index *is a finite sequence of positive integers that identifies a subterm of a term as follows: $\mathit{ref}(t, []) = t$; $\mathit{ref}(\text{(}o\ t_1 \ldots t_k\text{)}, [i]) = t_i$ if $1 \leq i \leq k$; $\mathit{ref}(\text{(}o\ t_1 \ldots t_k\text{)}, [i, j, \ldots]) = \mathit{ref}(t_i, [j, \ldots])$ if $1 \leq i \leq k$; $\mathit{ref}(t, \mathit{idx}) = \bot$ otherwise. We write $\mathit{refs}(t)$ for the set $\{\mathit{idx} \mid \mathit{ref}(t, \mathit{idx}) \neq \bot\}$.*

Definition 2 (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 (Scopes). *A tree index idx is in the* scope *of a pattern p if $\mathit{scope}(p, \mathit{idx}) \neq \bot$ where: $\mathit{scope}(p, []) = p$; $\mathit{scope}(\text{(}\texttt{Term}\ o\ p_1 \ldots p_k\text{)}, [i]) = p_i$ if $1 \leq i \leq k$; $\mathit{scope}(\text{(}\texttt{Term}\ o\ p_1 \ldots p_k\text{)}, [i, j, \ldots]) = \mathit{scope}(p_i, [j, \ldots])$ if $1 \leq i \leq k$; $\mathit{scope}(\text{(}\texttt{Term}\ o\ p_1 \ldots p_k\ \texttt{etc}\text{)}, \mathit{idx}) = \mathit{scope}(\text{(}\texttt{Term}\ o\ p_1 \ldots p_k\text{)}, \mathit{idx})$; and $\mathit{scope}(p, \mathit{idx}) = \bot$ otherwise. We write $\mathit{scope}(p)$ to denote the set $\{\mathit{idx} \mid \mathit{scope}(p, \mathit{idx}) \neq \bot\}$.*

Definition 4 (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. 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 5), which contain no invalid references. A reference expression `(Ref` $i_1 \ldots i_n$`)` specifies an index $[i_1, \ldots, i_n]$ into the matched term's abstract syntax tree (Definition 1). If a term matches the pattern of a well-formed program, then every reference in that program specifies a valid index into the term's AST. Additionally, `Apply` and `Cons` expressions reference subterms of the right kind; the program's actions edit disjoint subtrees of the term's AST; and (in)equality predicates only compare subterms matched by terminal patterns. RuleSy consumes and creates only well-formed programs.

Definition 5 (Well-Formed Programs). *Let R be a rule with the condition `(Condition (Pattern` p`) (Constraint` b`))` and action `(Action` $a_1 \ldots a_n$`)`. We say that R is* well-formed *if the following constraints hold:*

- $index(r) \in scope(p)$ *for all references r in R.*
- $scope(p, index(r)) =$ `(ConstTerm)` *for all references r in all* `Apply` *expressions.*
- $scope(p, index(r)) =$ `(Term ...)` *for all* `(Cons` e r`)` *expressions.*
- $scope(p, index(r)) \neq$ `(Term ...)` *for all references r in all* `Eq?`, `Neq?` *predicates.*
- *Let r_k denote the first argument to a command a_k in R. For all distinct a_i, a_j in R, $index(r_i)$ is not a prefix of $index(r_j)$ and vice versa.*

4 Rule Mining, Synthesis, and Optimization

Given an educational objective, example problems, and axioms for solving those problems, RuleSy produces an optimal domain model in three stages: (1) specification mining, (2) rule synthesis, and (3) domain model optimization. This section presents the algorithms underlying each stage and states their guarantees. Proofs of these statements are available in our technical report on RuleSy [17].

4.1 Specification Mining

Specification mining takes as input a set of examples and axioms, and produces a set of specifications for tactic rules. We describe the key challenge in specifying tactics; show how our notion of *execution plans* addresses it; and present our FindSpecs algorithm for computing these plans.

Specifying Tactics. To enable efficient synthesis of useful rules, a tactic specification should capture the semantics of a rule—i.e., a partial function—that

can help solve some problems in fewer steps than the axioms alone. But natural forms of specification, such as axiom sequences, do not satisfy this requirement. To see why, consider the axiom sequence $\mathbf{I} \circ \mathbf{B}$, where $\mathbf{I}$ implements factoring (Figure 6) and $\mathbf{B}$ implements constant folding (Figure 1a). Intuitively, we would like $\mathbf{I} \circ \mathbf{B}$ to specify the tactic **IB** for combining like terms (Figure 6). Yet no interpretation of this sequence captures the meaning of the tactic. If we interpret $\mathbf{I} \circ \mathbf{B}$ using the *fire* semantics, the result is a non-functional relation that includes the meaning of multiple tactics. For example, firing $\mathbf{I} \circ \mathbf{B}$ on the term $(+\,(*\,2\,x)\,(*\,3\,x)\,(*\,4\,y)\,(*\,5\,y))$ produces both $(+\,(*\,5\,x)\,(*\,4\,y)\,(*\,5\,y))$ and $(+\,(*\,9\,y)\,(*\,2\,x)\,(*\,3\,x))$. But if we interpret $\mathbf{I} \circ \mathbf{B}$ as the composition of the partial functions denoted by its axioms—i.e., as $\lambda t.[\![\mathbf{B}]\!]([\![\mathbf{I}]\!]t)$—the resulting relation is empty and thus fails to specify a useful tactic.

```
(define I ; (+ (* e0 e) (* e1 e) ...) → (+ (* (+ e0 e1) e) ...)
  (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))))))

(define IB ; (+ (* c0 e) (* c1 e) ...) → (+ (* c e) ...), c = c0 + c1
  (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))))))
```

Fig. 6: The tactic **IB** for combining like terms combines factoring (**I**) and constant folding (**B** in Figure 1a), but no interpretation of $\mathbf{I} \circ \mathbf{B}$ captures its behavior.

Execution Plans. We address the challenge of specifying tactic rules with *execution plans*. An execution plan (Definition 7) is a partial function from terms to terms, encoded as a sequence of execution steps (Definition 6). An execution step combines a rule R with a tree index idx and a binding β for R's pattern. The step $\langle R, idx, \beta \rangle$ uses the binding β, if it is valid for the subterm $ref(t, idx)$ 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. Execution plans thus satisfy our requirement for tactic specifications by defining useful functional relations.

Definition 6 (Execution Step). *An execution step $\langle R, idx, \beta \rangle$ consists of a rule program R, tree index idx, and a binding β for R's pattern. A step denotes a partial function over terms where $[\![\langle R, idx, \beta \rangle]\!]t = replace(t, idx, [\![R]\!]s^\beta)$ 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 7 (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 7. FindSpecs first obtains a *solution graph* (Definition 8) of all shortest solutions to each example problem (line 3). It then applies the FindPlan procedure to compute an execution plan for every path between every pair of nodes in each resulting graph (line 6). These plans specify the set of sound partial functions (Definition 10) that can shorten the solution to at least one example problem (Theorem 1).

```
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. 7: 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.

Definition 8 (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 satisfies the* Reduced *predicate.*

FindPlan takes as input a path p in a solution graph and produces a general execution plan (Definition 7) for *replaying* that path (Definition 9). The first loop, at lines 10-12, creates a plan that replays the path p from n_0 to n_k exactly: i.e., $[\![S]\!]n_0 = n_k$. 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 9 (Replaying Paths). *Let $p = n_0 \to_{R_1} \ldots \to_{R_k} n_k$ 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 $n_k = replace(n_0, idx, [\![S]\!]ref(n_0, idx))$.*

To illustrate, consider applying FindPlan to the path (= (+ x 1 -1) 5) $\to_{\mathbf{B}}$ (= (+ 0 x) 5) $\to_{\mathbf{A}}$ (= x 5) in Figure 2a. The Solve procedure computes this path p by firing $\mathbf{B}$ with $idx = [1]$, $\beta_{\mathbf{B}} = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3]\}$, and $\mathbf{A}$ 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.

Definition 10 (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 11 (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 1. *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 [13]: it draws candidate programs from a given syntactic space, and uses an automatic verifier to check if a chosen candidate satisfies the specification. We illustrate the challenges of classic syntax-guided synthesis for rule programs; show how our *best-implements* query addresses them; and present the FindRules algorithm for sound, complete, and efficient solving of this query.

Classic Synthesis for Rule Programs. In our setting, the classic synthesis query takes the form $\exists R. \forall t. [\![R]\!]t = [\![S]\!]t$, where R is a well-formed program and S is an execution plan. Existing tools [13,11,12] cannot solve this query soundly because it involves verifying candidate programs over terms of unbounded size.

But even if we weaken the soundness guarantee to functional correctness over bounded inputs, these tools can fail to find useful rules because the classic

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

query is overly strict for our purposes. To see why, 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 algebra DSL. Such a tactic cannot be expressed as a well-formed program (Definition 5). But many useful specializations of this tactic are expressible, e.g.:

```
(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 useful tactics for domain model optimization, an ideal synthesis query for RuleSy would admit many such specialized yet widely applicable implementations of S.

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 12), and we search for them with the FindRules algorithm (Figure 8), which is a sound and complete synthesis procedure for the best-implements query (Theorem 2).

Definition 12 (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 $\forall t.[\![pattern(R)]\!]t \implies [\![R]\!]t = [\![S]\!]t$.*

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 [17] that

this formula has a small model property when R is well-formed (Definition 5): 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. Encoding the set $\mathbb{T}$ symbolically (rather than explicitly) enables FindRules to discharge its verification task efficiently with an off-the-shelf SMT solver [18].

Efficient Search. FindRules accelerates synthesis by exploiting the observation that a best implementation of $\langle S, src, tgt\rangle$ must fire on src to produce tgt, which has two key consequences. First, because S replays a path from src to tgt (Theorem 1), 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 13) 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 efficiently search for a best implementation R with that pattern, by using an off-the-shelf synthesizer [12] to perform two independent searches for R's condition (line 13) and action (line 14). These two searches explore an exponentially smaller candidate space than a single search for the condition and action [17], without missing any correct rules (Theorem 2).

Definition 13 (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)`*;* `(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* $i \in [1..k]$.

Theorem 2. *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.*

4.3 Rule Set Optimization

After synthesizing the tactics $\mathcal{T}$ for the examples T and axioms $\mathcal{A}$, RuleSy applies discrete optimization to find a subset of $\mathcal{A} \cup \mathcal{T}$ that minimizes the objective function f. We formulate this optimization problem in a way that guarantees termination. In particular, our Optimize algorithm (Figure 9) 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 3). Restricting the optimization to shortest solutions enables us to decide whether an arbitrary rule set

1: **function** OPTIMIZE(T: set of terms, $\mathcal{A}$,$\mathcal{T}$: set of rules, f: objective)
2: $\quad\mathcal{G}_{\mathcal{A}\cup\mathcal{T}} \leftarrow \{\}$
3: $\quad$**for** $t \in T$ **such that** $\neg$REDUCED(t) **do**
4: $\quad\quad\langle N, E_{\mathcal{A}}\rangle \leftarrow$ SOLVE($t, \mathcal{A}$) ▷ *Solve with axioms*
5: $\quad\quad E_{\mathcal{T}} \leftarrow \bigcup_{R\in\mathcal{T}} \bigcup_{s,t\in N}\{\langle s,t\rangle \mid t \in \mathit{fire}(R,s)\}$ ▷ *Tactic edges*
6: $\quad\quad\mathcal{G}_{\mathcal{A}\cup\mathcal{T}} \leftarrow \mathcal{G}_{\mathcal{A}\cup\mathcal{T}} \cup \{\langle N, E_{\mathcal{A}} \cup E_{\mathcal{T}}\rangle\}$
7: $\quad f_\emptyset \leftarrow \lambda\mathcal{R}.\mathcal{G}.$ **if** $\langle\emptyset,\emptyset\rangle \in \mathcal{G}$ **then return** ∞ **else return** $f(\mathcal{R},\mathcal{G})$
8: $\quad$**return** $\min_{\mathcal{R}\subseteq\mathcal{A}\cup\mathcal{T}} f_\emptyset(\mathcal{R}, \{\text{RESTRICT}(G,\mathcal{R}) \mid G \in \mathcal{G}_{\mathcal{A}\cup\mathcal{T}}\})$
9: **function** RESTRICT($\langle N, E\rangle$: solution graph, $\mathcal{R}$: set of rules)
10: $\quad t \leftarrow$ source of the graph $\langle N, E\rangle$
11: $\quad E_{\mathcal{R}} \leftarrow \{\langle \mathit{src}, \mathit{tgt}\rangle_R \in E \mid R \in \mathcal{R}\}$ ▷ *Edges with labels in* $\mathcal{R}$
12: $\quad \mathit{paths} \leftarrow \bigcup_{\hat{t}\in N\wedge\text{REDUCED}(\hat{t})} \mathit{allPaths}(t, \hat{t}, \langle N, E_{\mathcal{R}}\rangle)$
13: $\quad E \leftarrow \bigcup_{p\in\mathit{paths}} \mathit{pathEdges}(p)$
14: $\quad N \leftarrow \{n \mid \exists e \in E\,.\,\mathit{source}(e) = n \vee \mathit{target}(e) = n\}$
15: $\quad$**return** $\langle N, E\rangle$ ▷ *Solution graph for* t *and* $\mathcal{R}$ *or* $\langle\emptyset,\emptyset\rangle$

Fig. 9: OPTIMIZE takes as input a set of terms T, axioms $\mathcal{A}$ for reducing T, tactics $\mathcal{T}$ synthesized from $\mathcal{A}$ and T using FINDRULES and FINDSPECS, and an objective function f. The output is a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that minimizes f.

$\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ can solve an example $t \in T$ without having to invoke SOLVE$(t, \mathcal{R})$, which may not terminate for an arbitrary term t and rule set $\mathcal{R}$ in our DSL.

The OPTIMIZE procedure works in three steps. First, for each example term $t \in T$, lines 4-5 construct a solution graph $\langle N, E_{\mathcal{A}} \cup E_{\mathcal{T}}\rangle$ that contains shortest solutions for t and 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 [18]. For other objectives (e.g., Figure 1c), we use a greedy algorithm to find a locally minimal solution (thus weakening the optimality guarantee in Theorem 3).

Theorem 3. *Let $\mathcal{T}$ be a set of tactics synthesized by* RULESY *for terms T and axioms $\mathcal{A}$, and let f be a total function from sets of rules and solution graphs to positive real numbers.* OPTIMIZE$(T, \mathcal{A}, \mathcal{T}, f)$ *returns a set of rules $\mathcal{R} \subseteq \mathcal{A}\cup\mathcal{T}$ that can solve each term in T, and for all such $\mathcal{R}' \subseteq \mathcal{A} \cup \mathcal{T}$, $f(\mathcal{R}, \{\text{SOLVE}(t, \mathcal{R}) \mid t \in T\}) \leq f(\mathcal{R}', \{\text{SOLVE}(t, \mathcal{R}') \mid t \in T\})$.*

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 [15] for modeling the domain of introductory K-12 algebra?
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 textbook [9] 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. We conducted two case studies to answer these questions, finding positive answers to each. The implementation source code and evaluation data are available online [19].

5.1 Case Study with Algebra (RQ 1–3)

We performed three experiments in the domain of K-12 algebra 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 are presented below.

Table 1: Example problems (a) and axioms (b) for the algebra case study.

(a) Example problems.

ID	Source	#
P_R	RuleSynth [15]	55
P_T	Chapter 2, Sections 1-4 of Hall et al. [9]	92

(b) Axioms.

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 1a) and axioms (Table 1b) from a standard algebra textbook [9], and compare system output (607 tactics) to the tactics from the textbook. Since the book demonstrates rules on examples rather than explicitly, determining which rules are shown involves some interpretation. For example, we interpret the transformation $5x+2-2x = 2x+14-2x \rightarrow 3x+2 = 14$ as demonstrating two independent tactics, one for each side of the equation, rather than one tactic with unrelated subparts. The second column of Table 2 lists all the tactics presented in the book. We find that RuleSy recovers each of them or a close variation.

In addition to recovering textbook tactics, RuleSy finds interesting variations on rules commonly taught in algebra class. Figure 10 shows an example, which isolates a variable from a negated fraction and an addend. This rule composes 9 axioms, demonstrating RuleSy's ability to discover advanced tactics.

```
(define MBALNGOHG ; Isolate a variable from a negated fraction and an addend:
  (Rule           ; (= (+ (- (/ (* x ...) b)) c) e) → (= (* x ...) (* b (- c e)))
    (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. 10: A custom algebra tactic discovered by RuleSy.

```
(define xpq ; Modus ponens: if I ⊨ A → B and I ⊨ A, then I ⊨ B.
  (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. 11: A proof tactic synthesized by RuleSy.

Quality of Synthesized Domain Models (RQ 2). We next evaluate RuleSy's ability to recover textbook domain models along with variations that optimize different objectives. An important part of creating domain models for educational tools (and curricula in general) is choosing the *progression*—the sequence in which different concepts (i.e., rules) should be learned. We use RuleSy and the objective function shown in Figure 1c to find a progression of optimal domain models for the problems (P_T in Table 1a) and axioms (Table 1b) in [9], 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 [9]. 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 1c with $\alpha \in \{.05, .125, .25\}$; and all available rules (axioms and tactics), coupled with the constraint that $D_1 \cup \ldots \cup D_{n-1} \subseteq D_n$.

Table 2 shows the resulting progressions of optimal domain models for [9], 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. This result

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

Table 2: A textbook [9] progression and the corresponding optimal domain models found by RuleSy, using 3 settings of α (Figure 1c). 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 1a and the axioms in Table 1b. We use the same problems as the original evaluation of RuleSynth because its algorithms encounter performance problems on the (larger) textbook problems P_T. Given these inputs, RuleSy synthesizes 144 tactics, which include the 13 rules synthesized by RuleSynth. Figure 12 graphs the rate of rules produced by each system, which accounts for the time to mine specifications and synthesize rules for those specifications. 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 set. This process is neither sound nor complete, so RuleSynth can produce incorrect tactics and miss tactic specifications found by RuleSy.

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 algebra DSL. Given these tactics, the axioms in Table 1b, and the examples P_R, RuleSynth finds an optimal rule set in 20 seconds, 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 13 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 [20], so custom educational tools could help by teaching a variety of proof strategies.

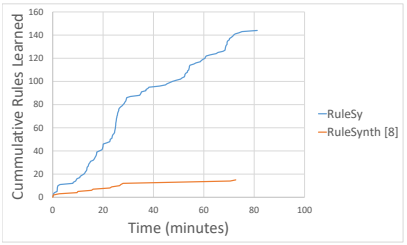


Fig. 12: The number of rules synthesized by RuleSy (blue) and RuleSynth (orange) over time on the same inputs. RuleSy learns 10× more rules at a quicker rate.

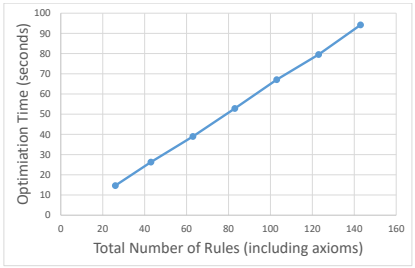


Fig. 13: Optimizer performance on design spaces derived from P_R of various sizes, using the objective in Figure 1c with $\alpha = .125$.

Table 3: The axioms [16] 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 $I \vDash A \to B$, then $I \nvDash A \mid I \vDash B$
y	Implication 2	If $I \nvDash A \to B$, then $I \vDash A$
z	Implication 3	If $I \nvDash A \to B$, then $I \nvDash B$

We instantiated RuleSy with a DSL for expressing semantic proofs. The DSL represents problem states as proof trees, consisting of a set of branches, each containing 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 $(p \wedge q) \to (p \to q)$, by assuming a falsifying interpretation and applying proof rules to arrive at a contradiction in every branch. Tactics apply multiple proof steps (i.e., axioms) at once.

We applied this instantiation of RuleSy to the axioms (Table 3) and proof exercises (3 in total) from a textbook [16]. The system synthesized 85 rules in 72 minutes. The resulting rules includes interesting general proof rules for each of the exercises. For example, given $(p \wedge (p \to q)) \to q$, RuleSy mines and synthesizes the modus ponens tactic shown in Figure 11. These results show RuleSy's applicability and effectiveness extend beyond the domain of K-12 algebra.

6 Related Work

Automated Rule Learning. Automated rule learning is a well-studied problem in Artificial Intelligence and Machine Learning. RuleSy is most closely related to

rule learning approaches in discrete planning domains, such as cognitive architectures [21]. Its learning of tactics from axioms is similar to chunking in SOAR [22], knowledge compilation in ACT [23], and macro-learning from AI planning [24]. But unlike these systems, RuleSy can learn rules for transforming problems represented as trees, and express objective criteria over rules and solutions.

Inductive Logic Programming. Within educational technology, researchers have investigated automated learning of rules and domain models for intelligent tutors [25]. Previous efforts have focused on applying inductive logic programming to learn a domain model from a set of expert solution traces [26,27,28,29,30]. RuleSy, in contrast, uses a small set of axioms and example problems to synthesize an exhaustive set of sound tactics, and it searches the axioms and tactics for a model that optimizes a desired objective.

Program Synthesis. Prior educational applications of program synthesis and automated search include problem and solution generation [31,1], hint and feedback generation [32,33,34], and checking of student proofs [35]. RuleSy solves a different problem: generating condition-action rules and domain models. General approaches to programming-by-example [36,37] have investigated the problem of learning useful programs from a small number of input-output examples, with no general soundness guarantees. RuleSy, in contrast, uses axioms to verify that the synthesized programs are sound for all inputs, relying on examples only to bias the search toward useful programs (i.e., tactics that shorten solutions).

Term Rewrite Systems. RuleSy helps automate the construction of rule-based domain models, which are related to term rewrite systems [38]. Our work can be seen as an approach for learning rewrite rules, and selecting a cheapest rewrite system that terminates on a given finite set of terms. RuleSy terms are a special case of recursive data types, which have been extensively studied in the context of automated reasoning [39,40,41]. Our rule language 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 domain models expressed as condition-action rules. RuleSy is based on new algorithms for mining specifications of tactic rules from examples and axioms, synthesizing sound implementations of those specifications, and selecting an optimal domain model from a set of axioms and tactics. Thanks to these algorithms, RuleSy efficiently recovers textbook tactic rules and models for K-12 algebra, discovers new ones, and generalizes to other domains. As the need for tools to support personalized education grows, RuleSy can help tool developers rapidly create domain models that target individual students' educational objectives.

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

References

1. Andersen, E., Gulwani, S., Popović, Z.: A trace-based framework for analyzing and synthesizing educational progressions. In: CHI. (2013)
2. O'Rourke, E., Andersen, E., Gulwani, S., Popović, Z.: A framework for automatically generating interactive instructional scaffolding. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. CHI '15, New York, NY, USA, ACM (2015) 1545–1554
3. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: Lessons learned. The journal of the learning sciences **4**(2) (1995) 167–207
4. VanLehn, K.: Mind bugs: The origins of procedural misconceptions. MIT press (1990)
5. Murray, T.: Authoring intelligent tutoring systems: An analysis of the state of the art. International Journal of Artificial Intelligence in Education **10** (1999)
6. Liu, Y.E., Ballweber, C., O'rourke, E., Butler, E., Thummaphan, P., Popović, Z.: Large-scale educational campaigns. ACM Trans. Comput.-Hum. Interact. **22**(2) (March 2015) 8:1–8:24
7. Demski, J.: 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
8. Redding, S.: Getting personal: The promise of personalized learning. Handbook on innovations in learning (2013) 113–130
9. Charles, R.I., Hall, B., Kennedy, D., Bellman, A.E., Bragg, S.C., Handlin, W.G., Murphy, S.J., Wiggins, G.: Algebra 1: Common Core. Pearson Education, Inc. (2012)
10. Sweller, J.: Cognitive load during problem solving: Effects on learning. Cognitive science **12**(2) (1988) 257–285
11. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM (2006)
12. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. (2014)
13. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering. (2015) 1–25
14. Dershowitz, N., Jouannaud, J.P.: Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA (1990) 243–320
15. Butler, E., Torlak, E., Popović, Z.: A framework for parameterized design of rule systems applied to algebra. In: Intelligent Tutoring Systems, Springer (2016)
16. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
17. Butler, E., Torlak, E., Popovic, Z.: Synthesizing optimal domain models for educational applications. Technical Report UW-CSE-17-10-02, https://www.cs.washington.edu/tr/2017/10/UW-CSE-17-10-02.pdf, University of Washington (2017)

18. Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: 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 (2008) 337–340
19. Butler, E., Torlak, E., Popovic, Z.: Rulesy source code and data. `https://github.com/edbutler/nonograms-rule-synthesis`
20. Harel, G., Sowder, L.: Toward comprehensive perspectives on the learning and teaching of proof. Second handbook of research on mathematics teaching and learning **2** (2007) 805–842
21. Langley, P., Laird, J.E., Rogers, S.: Cognitive architectures: Research issues and challenges. Cognitive Systems Research **10**(2) (2009) 141–160
22. Laird, J.E., Newell, A., Rosenbloom, P.S.: Soar: An architecture for general intelligence. Artificial Intelligence **33**(1) (1987) 1 – 64
23. Anderson, J.R., Lebiere, C.: The atomic components of thought. (1998)
24. Korf, R.E.: Macro-operators: A weak method for learning. Artificial Intelligence **26**(1) (1985) 35 – 77
25. Koedinger, K.R., Brunskill, E., Baker, R.S., McLaughlin, E.A., Stamper, J.: New potentials for data-driven intelligent tutoring system development and optimization. AI Magazine **34**(3) (2013) 27–41
26. Matsuda, N., Cohen, W.W., Koedinger, K.R.: 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). (2005) 1–8
27. Li, N., Cohen, W., Koedinger, K.R., Matsuda, N.: A machine learning approach for automatic student model discovery. In: Educational Data Mining 2011. (2010)
28. Li, N., Schreiber, A.J., Cohen, W., Koedinger, K.: Efficient complex skill acquisition through representation learning. Advances in Cognitive Systems **2** (2012)
29. Jarvis, M.P., Nuzzo-Jones, G., Heffernan, N.T.: Applying machine learning techniques to rule generation in intelligent tutoring systems. In: Intelligent Tutoring Systems, Springer (2004) 541–553
30. Schmid, U., Kitzelmann, E.: Inductive rule learning on the knowledge level. Cognitive Systems Research **12**(3) (2011) 237–248
31. Gulwani, S.: Example-based learning in computer-aided stem education. Communications of the ACM **57**(8) (2014) 70–80
32. Lazar, T., Bratko, I.: Data-driven program synthesis for hint generation in programming tutors. In: Intelligent Tutoring Systems, Springer (2014) 306–311
33. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. (2013)
34. Tillmann, N., de Halleux, J., Xie, T., Bishop, J.: Constructing coding duels in Pex4Fun and Code Hunt. In: ISSTA, ACM (2014) 445–448
35. Lee, C.: DeduceIt: a tool for representing and evaluating student derivations. Stanford Digital Repository: http://purl.stanford.edu/bg823wn2892 (2012)
36. Polozov, O., Gulwani, S.: 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 (2015) 107–126
37. Liang, P., Jordan, M.I., Klein, D.: Learning programs: A hierarchical bayesian approach. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10). (2010) 639–646
38. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. Citeseer (1989)

39. Oppen, D.C.: Reasoning about recursively defined data structures. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM (1978) 151–157
40. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. Acm Sigplan Notices **45**(1) (2010) 199–210
41. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. Electronic Notes in Theoretical Computer Science **174**(8) (2007) 23–37