Crowd-scale Interactive Formal Reasoning and Analytics

Ethan Fast[1], **Colleen Lee**[1], **Alex Aiken**[1], **Michael Bernstein**[1], **Daphne Koller**[1], **Eric Smith**[2]
Stanford University[1], Kestrel Institute[2]
{ethan.fast, clee0, aiken, msb, koller}@cs.stanford.edu, eric.smith@kestrel.edu

ABSTRACT

Large online courses often assign problems that are gradable by simple checks such as multiple choice, but these checks are inappropriate for domains in which students may produce an infinity of correct solutions. One such domain is *derivations*: sequences of logical steps commonly used in assignments for technical, mathematical and scientific subjects. We present DeduceIt, a system for creating, grading, and analyzing derivation assignments across arbitrary formal domains. DeduceIt supports assignments in any logical formalism, provides students with incremental feedback, and aggregates student paths through each proof to produce instructor analytics. DeduceIt benefits from checking thousands of derivations on the web: it introduces a *proof cache*, a novel data structure which leverages a crowd of students to decrease the cost of checking derivations and providing real-time, constructive feedback. We evaluate DeduceIt with 990 students in an online compilers course, finding students take advantage of its incremental feedback and instructors benefit from its structured insights into confusing course topics. Our work suggests that automated reasoning can extend online assignments and large-scale education to many new domains.

Author Keywords

MOOC, theorem prover, formal logic, online education

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: Graphical user interfaces

General Terms

Human Factors; Design

INTRODUCTION

As online courses enroll thousands of students [22], course staff are unable to provide feedback on assignments and instead turn to automatic grading systems. Automatic grading works well for multiple choice quizzes, but it remains difficult in domains where students may construct many correct solutions [2, 10]. *Derivations*, or sequences of logical steps where each step follows from its predecessors, are one popular assignment type that technical, mathematical, or scientific subjects often assign in their offline problem sets. However, because derivations encode open-ended reasoning, they are uncommon in large online courses today.

With derivation assignments, it is important to provide *personalized, realtime feedback* to students and instructors. While a grading system can report whether an answer is correct (e.g., through multiple choice or string matching), it doesn't know enough about a student's reasoning to provide guidance for a wrong answer. Most grading systems offer limited feedback that is unassociated with the intermediate steps of a solution [1]. But derivations are prone to a great deal of internal variation, and often have many right answers [3], so we face the challenge of providing *constructive* suggestions [9]. Similarly, students tend to learn better when subject to *real-time* feedback loops [8], for example help from human TAs in a traditional course setting [9, 3, 30]. However, theorem prover calls are expensive computations [14] and often do not complete at interactive speeds. Instructors, only seeing the final product, also remain largely uninformed about the paths that students take to construct derivations.

In this paper, we pursue large-scale online derivation assignments for any formal domain and at any level of abstraction. Critically, we suggest that *hosting these derivations online, at large scale, allows realtime personalized feedback for students and detailed analytics for instructors*. Unlike previous work focused on single domains like calculus (e.g., [31, 23]), we allow instructors to support derivations in *any* formal domain: for example, compilers, regular expressions, or mathematics. Instructors may also specify levels of proof detail for an assignment; for instance, they may require that students report some derivation rules but elide others.

Our approach illustrates the interface benefits that follow from the scale of online courses. Rather than provide slow feedback based entirely on proof search, we create a *proof cache* that saves other students' previous attempts at the derivation and can thus provide realtime feedback and guidance to students. By aggregating these paths, we enable instructors to analyze student progress and challenges.

In this paper we present:

- *DeduceIt*: A web-based system which allows instructors to to specify a general class of derivation exercises, scales to a large workload of students, and enables students to complete assignments at varied levels of abstraction. DeduceIt provides students with constructive and real-time feedback.

Submitted for review.

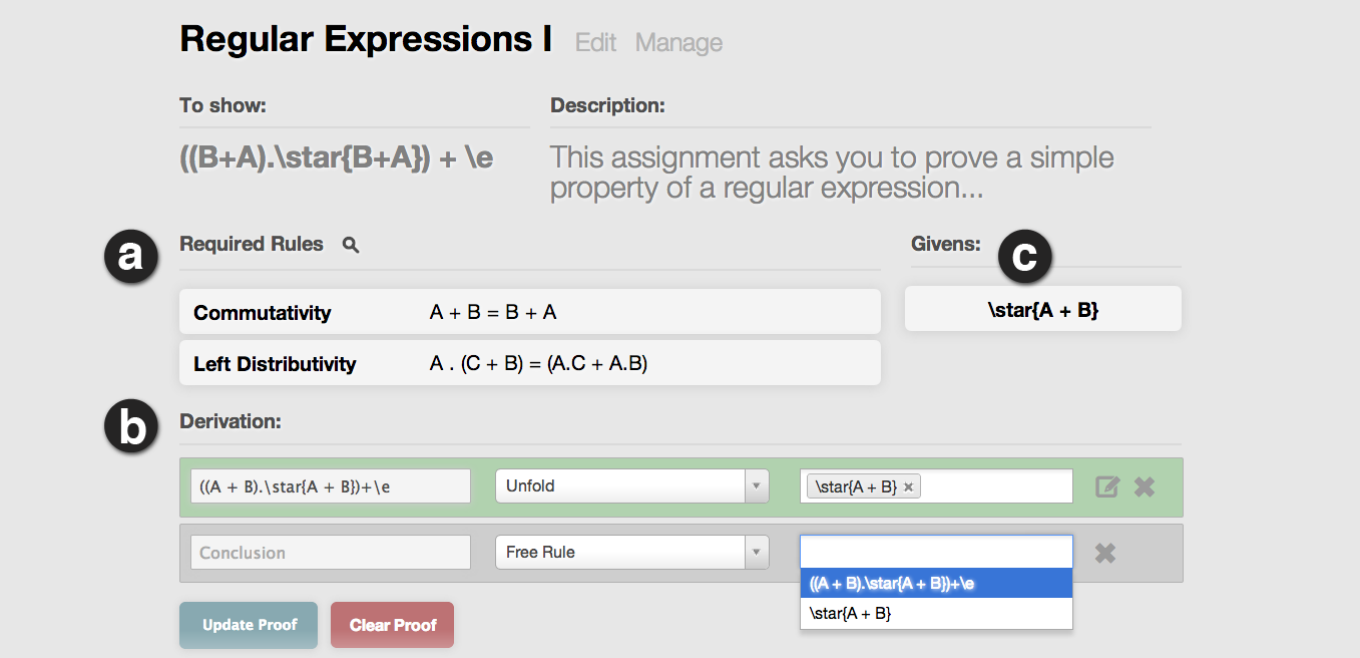

Figure 1: The DeduceIt interface: (a) set of available rules (b) interactive derivation component (c) set of given assumptions

- The *proof cache*: A novel data structure which records the history of every attempted derivation, reusing computations from previously derived steps. We show that this cache increases derivation-checking efficiency by 87% and provides feedback in less than one second, much faster than what is possible in a single user system.
- An *empirical evaluation* of DeduceIt: a deployment of DeduceIt to 990 students online for assignments in type checking, regular expressions, finite automata, and several other topics. We detail how: 1) students take advantage of DeduceIt's incremental feedback for exploration, and 2) instructors use DeduceIt to obtain structured insights into confusing course topics.

The rest of this paper is organized as follows: we begin with related work and a motivating example, then we present DeduceIt's interface and capabilities. Next, we describe the implementation of the system and provide an empirical evaluation, using data collected from students in an online class. We close with reflections, conclusions, and future work.

RELATED WORK

At its core, DeduceIt is an interactive proof solving system. Many existing tools provide users with rich and automated feedback as a form of proof assistance [23, 6, 24, 21, 15, 4, 25, 31], but they are tied to specific formalisms or require background knowledge generally not assumed in a MOOC audience. For instance, some tools allow users to explore complex problem domains, but they require knowledge of an underlying programming language that students or instructors may find difficult to understand (e.g., [23, 24]); other tools present a more pedagogically oriented interface, but they are limited in the kinds of formalisms they can support (e.g, [31]).

Automated theorem provers have been applied to educational settings [27, 28, 25, 4], and the idea of using theorem provers to enable education has engendered much discussion among researchers [10, 2]. However, systems designed to address educational use have also tended toward interfaces designed for a single domain (e.g., geometry [25]) or present unstructured interactions that are designed for users already familiar with programming (e.g., ProveEasy [4]). DeduceIt aims to support constrained interactions across many (incomplete) formalisms at just enough depth for students to explore.

Other work addresses automatic feedback and grading systems with a more domain specific focus. Simple forms of automatic grading have been studied in both physical and online classrooms [12]. These systems tend to restrict an assignment's answer domain with assumptions like multiple choice answers, concrete-valued answers (e.g., answers that match against a string or numerical value), or a set of test cases to evaluate a programming assignment. Tools can assist with formal reasoning tasks like model checking or logical inference [18], but these tools aren't designed toward pedological ends. Further, automated grading and plagiarism detection systems have long evaluated student program code—where solutions may be unstructured and creative—in computer science departments [5, 26], but we are concerned with a system that works across all kinds of derivations.

Finally, crowdsourcing research has enabled new forms of problem-solving and evaluation. For instance, peer consistency evaluation can be used to effectively judge the accuracy—and grade—of a student's answer, even in the absence of a ground truth [13] and crowd-based peer assessment has been successfully applied to grade student assignments

[29, 17]. DeduceIt does not require human evaluation; it instead uses crowds to guide its exploration of the proof space.

SCENARIO

DeduceIt provides students with feedback and support for any formal domain, and allows instructors to specify the level of abstraction at which a student may work though a derivation. In this section, we demonstrate these ideas.

Regular Expressions I is an assignment we use to introduce students to DeduceIt. Here, students must show that two regular expressions are equal: $(A+B)* \equiv ((B+A).(B+A)*)+\epsilon$. We assign the syntax $+$, $*$, ϵ, and . as union, Kleene closure, empty string, and concatenation. Students must derive the expression $((B+A).(B+A)*)+\epsilon$ starting from a set of givens, in this case the single expression $(A+B)*$, using given properties of regular expressions.

Constraints on the Derivation Interface

DeduceIt limits the actions available to students at each step of the derivation. Students may freely enter expressions into the conclusion box of a derivation, but the rules and givens input fields are constrained to lists of possibilities. This offers students a form of guidance.

Since our student has been given only $(A + B)*$ from which to start her derivation, she reasons that she must select this expression from the givens input field on the first step. From the rule field she must select a rule from the list of assignment rules, and in the conclusion field she must enter a newly derived expression. She selects the "Unfold" rule, which expands a Kleene star expression. She then tries Unfold on the full starting expression, mentally applying the rule to the given and typing out $((A + B).(A + B)*) + \epsilon$ in the conclusion. She submits the step.

The derivation returns with her step highlighted in green: it was successful. A new set of input fields lies below her previous entry, querying her for the next step.

Working at Specifiable Levels of Abstraction

Since our student's previous conclusion is nearly identical to the goal, she wants to use it for her next step. If she can transform its two sub-expressions $A + B$ to $B + A$, then she will have proven the goal and completed the assignment. She decides to apply Commutativity to each of the two $A + B$ sub-expressions, entering her result in the conclusion input box on the next line of the derivation. This new conclusion is identical to the goal expression: $((B + A).(B + A)*) + \epsilon$

She selects "Commutativity" from the rule input field, selects $((B + A).(B + A)*) + \epsilon$ from the givens field, and submits the step. DeduceIt checks her derivation, infers through proof search that she means to apply Commutativity twice—on the two appropriate sub-expressions—and responds that her derivation is correct. She has completed this assignment.

Here our student *interacts with DeduceIt at a relatively high level of abstraction*: she applies Commutativity twice, simultaneously, in a single derivation step. Under a different assignment setup she might be required to enter each application of Commutativity separately; or alternatively, she might be allowed to apply Commutativity freely without needing to specify it as a separate proof step.

Providing Constructive and Real-time Feedback

Immediate feedback is critical to the design of interactive educational systems [8]. If DeduceIt were running locally on her computer, the student would have needed to wait up to four seconds for each incremental piece of feedback. Instead, feedback is available in under one second, which allows her to move from one step to another with confidence that her prior reasoning is correct. This acceleration is possible because DeduceIt can utilize proof steps that other students computed previously.

Displaying Analytics and Hints

The instructor opens the analytics interface. He sees that many students are struggling to complete an easy exercise, and in particular that they are often veering into a dead-end proof path, so he annotates that subtree of the derivation with a hint suggesting why the direction will not be fruitful. When a student next tries this direction, she receives the feedback and reroutes, saving herself some frustration.

DEDUCEIT

DeduceIt is an online derivation system that supports arbitrary formal domains and specifiable levels of abstraction. It takes advantage of large courses by creating a cache of proofs that it uses to optimize verification and power derivation analytics for the instructor.

Interacting with a DeduceIt Derivation

Each DeduceIt assignment contains an interactive derivation (Figure 1). To move forward in the derivation, a student must derive a conclusion by applying the selected rule on the selected givens(s).

On each derivation step, DeduceIt responds with one of several kinds of *feedback*: if the derivation is so far valid, all its steps will turn green; if the system cannot parse the conclusion of some step in the derivation, that step will turn yellow; or if the system can parse but not prove the conclusion of some step in the derivation, that step will turn red. Each valid step preceding an invalid step will remain green.

DeduceIt derivations have *constraints* that aim to make it easier for students to complete derivations while avoiding syntactical mistakes that are unrelated to learning. Students may only select expressions they have already derived (or members of the set of starting givens) from the givens input field. Likewise, students may only select rules which are associated with the current assignment from the rules input field.

Finally, if an invalid step has any *hint annotations* on the proof tree, a hint appears adjacent to that step. Hints take the form of a question mark, expanding into text on mouseover.

Assignment Analytics on the Proof Tree

Because many students complete each derivation, DeduceIt can give instructors real-time access to assignment analytics. DeduceIt creates a *proof tree* for each assignment (Figure 2), which tracks the history of all associated derivations. These

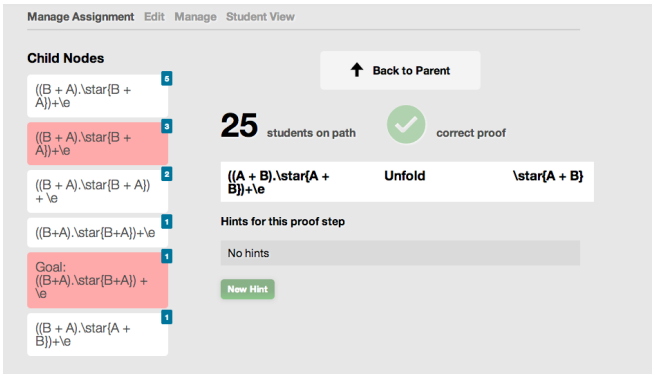

Figure 2: Annotating a node on the proof tree.

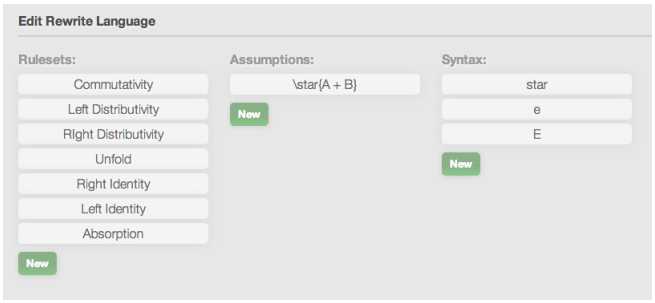

Figure 3: Instructors must specify a rewrite language for each assignment.

derivations share a common root node, and each step in a derivation maps from a parent node to a child node via a step in the proof. Every node keeps track of derivation state information and a count of how many students have traversed it. An instructor may annotate any node in the tree with hints, and these will be visible to a student if she enters upon that path in her own derivation. An instructor may also use this tree to override the behavior of the underlying derivation checker, e.g., forcibly label a given derivation step valid. This can be useful for steps which make use of particularly long chains of free rules that aren't discovered and verified because of limitations in proof search.

Extensions to the Derivation Interface

Based on our experiences with DeduceIt, we have developed several other forms of derivation feedback which we have not yet deployed for an online class.

Displaying the Proof Path

DeduceIt maintains aggregate data about every derivation, so the system knows when students are proceeding down well-traveled paths in a derivation, down correct but—so far—more lengthy paths, or down paths which have not yet led to the goal. DeduceIt can optionally display this information with a status indicator, a colored circle of green, yellow, or orange at the top of the derivation, indicating whether students are on a common path, an uncommon but successful path, or an as yet unsuccessful path.

DeduceIt could provide students even more fine-grained information: the exact number of other successful or unsuccessful students who have worked to the current state of their derivation; or detection of the exact step in their derivation where they left the well-traveled path. It is also possible to further process the proof tree. By collapsing some nodes which are syntactically different but semantically equivalent (e.g., the expressions $(1 + 2) + 3$ and $1 + (2 + 3)$) we could construct a more meaningful notion of a derivation path.

Providing Automatic Hints

DeduceIt allows instructors to set up hints for any derivation step by annotating an assignment's proof tree. Optionally, DeduceIt can construct these hints automatically using proof search by holding the rule and assumption fields constant and searching for alternative valid conclusions.

For instance, suppose that students enter $x = 3 + 1$, *Balance Equation*, and $x + 1 = 3$ in the fields for conclusion, rule, and assumptions. This is an incorrect step, so DeduceIt will highlight it in red. But the hint-generating DeduceIt is able to give students more specific feedback, e.g., "Balance Equation can be applied, but your conclusion is incorrect." Here proof search finds a viable alternative conclusion, $x = 3 - 1$, so DeduceIt knows it is possible to apply the selected rule upon the selected given.

Other hint-generating systems might hold constant different parts of the derivation (e.g., searching for rules and assumptions to match a given conclusion), or draw inspiration from Model-Tracing Tutors [11].

Background: Term Rewriting Systems

To understand the instructor's authoring interface, we begin by reviewing the major concepts behind *term rewriting systems*. DeduceIt uses a term rewriting system to verify student derivations. Term rewriting systems consist of a set of expressions with nested sub-expressions, and relations which define transformations on those expressions; each transformation is called a *rewrite rule* [19, 24]. Term rewrite rules have a left side, which must match a term for the rule to be applied, and a right side, which defines the new expression produced by the rule. Variables may appear in the rule (declared in the system with $ notation) which binds to the term or its subexpressions. For example, in a language which supports integers, symbols, and the binary operators $+$, $-$, and $=$ (this happens to be a subset of the default DeduceIt language), one such rule might be: $(\$x + \$y = \$z) \rightarrow (\$x = \$z - \$y)$. The system can apply this rule to the expression $2 + 1 = 3$ to produce $2 = 3 - 1$.

Assignment Creation Across Arbitrary Formalisms

Taking advantage of term rewriting systems, instructors can use DeduceIt to define nearly any kind of formal assignment. To create an assignment, an instructor must specify four things:

1. Rewrite language: Every assignment has a default rewrite language composed of variables, symbols, integers, and several common unary and binary operators. In many assignment domains this will be sufficient, but an instructor

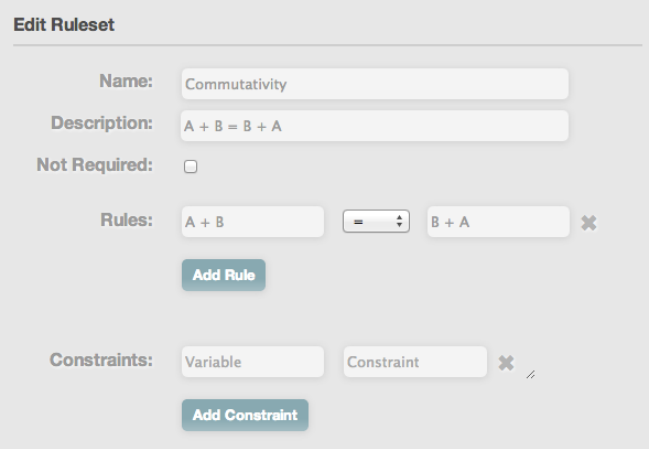

Figure 4: Editing a ruleset on the instructor interface.

may optionally augment the language with extra syntax for functions and constants (Figure 3).

2. Rulesets: Named sets of rewrite rules which a student may apply while working through an assignment's derivation. One ruleset corresponds to many underlying rewrite rules. This is necessary because an instructor may want to refer to several distinct rules by the same name (e.g., $1 * X \rightarrow X$ and $X * 1 \rightarrow X$ are two distinct rewrite rules describing the multiplicative identity.)

3. The given expression(s): A set of expressions which serve as the starting point of a derivation.

4. The goal expression: The desired result of a derivation.

Most assignments have a small number of rulesets and given expressions, because DeduceIt is designed for pedagogical purposes and not large, general proofs. This makes debugging fairly straightforward, and in practice assignments take between 5 and 30 minutes to prepare and up to another 30 to test, as reported by the TA for the class in our evaluation.

Customizing an Assignment Language

Instructors can customize an assignment's language, allowing students to use familiar notation. By default DeduceIt provides the unary operators ˜ and $-$, and the binary operators ., &, |, ,, $*$, \, $+$, $-$, $=$, $\neq$, $\leq$, $\geq$, $<$, $>$, $=>$, $:=$, and $\rightarrow$, listed in order of precedence. DeduceIt also supports variables (only used when defining rewrite rules), symbols, and integers. For example, three expressions in the rewrite language are: "$\$p, (\$p => \$q) \rightarrow \$q$", "$a.b.b.a$", and "$x + 2 = y$".

Note that with several exceptions (the notation specific to rewrites: $:=$, $\rightarrow$, and \$) the operators used in expressions have no meaning without an accompanying set of rules; they simply determine the parsing of an expression.

Instructors may add two kinds of syntax to DeduceIt's standard rewrite language: constants and functions. This is convenient in many domains, where familiarly named functions and constants enhance the readability of an assignment. For these custom functions and constants, constants appear as a symbol value preceded by a backslash, e.g., $\backslash e$. Functions look much the same, except they have arguments which they accept in brackets, e.g., $\backslash sin\{x\}$.

Defining an Assignment Domain with Rulesets

To control the level of abstraction at which students complete an assignment, instructors define a ruleset for each student-facing rule. A ruleset defines a *set* of transformations that a student may use in a derivation, because each rule often requires multiple rewrite rules for its implementation, e.g., $1 * X \rightarrow X$ and $X * 1 \rightarrow X$ for the multiplicative identity. Each ruleset has a name, a student-facing description, a set of rewrite rules, and an optional set of constraints (Figure 4). The name and description of a ruleset are all a student sees when using DeduceIt; students do not need to understand rewrite rules.

To allow students to elide less important proof steps, the instructor may choose whether a ruleset is *required* or *free*. Required rulesets must be named explicitly when a student uses them in a derivation. Free rulesets, however, may be elided. Behind the scenes, DeduceIt will attempt to fill in free steps automatically through proof search. This is useful when a student must use trivial transformations which are necessary to the derivation but unimportant to the assignment.

Instructors can define both *strict* and *context independent* rewrite rules. To apply a strict rule, its left side must match exactly on an expression, whereas a context independent rule may be applied to an expression or any of its subexpressions. For many assignments a single context independent rule can take the place of many strict rules. This eases the burden on an instructor and usually speeds up proof search. However, strict rewrite rules cannot be entirely replaced with context independent rules. Some transformations may be context dependent (e.g., lexing character values of a string). Moreover, *strict rewrite rules allow instructors to simulate complex assignment domains which would otherwise lie outside the scope of a typical rewrite rule based system.*

For example, we use strict rewrite rules to define a type checking assignment over a specific domain of input $\backslash type\{t\}\{expr\}$. Because we tailor such rules specifically to the context of this assignment, we do not need the power of full type analysis: rewrite rules are sufficient. Further, as students see only the description of each rule, which tends to be general, and not the underlying implementation as rulesets, which is often more specific, DeduceIt appears to have expressive capability far beyond rewrite systems. Instructors would use a similar approach to deploy assignments in domains like parsing or multivariate calculus.

Sharing Rewrite Languages

Some rewrite languages are common enough that it makes sense to share them among different assignments: for instance, many assignments might use the languages defined for basic algebraic manipulation, predicate logic, or the expansion of regular expressions. DeduceIt allows instructors to reuse a rewrite language across assignments.

IMPLEMENTATION AND PROOF CACHE

The core implementation challenge of DeduceIt is to run a large-scale theorem prover in real time. DeduceIt is a system formed of three distinct components: a frontend interface, a backend theorem prover, and a database. The frontend manages all user interactions (for both students and instructors) as described in the previous section, the backend theorem prover exposes an online API which the frontend calls—when necessary—to check student derivations, and the database stores all the data associated with users and assignments, including the various proof trees which together constitute the proof cache.

The frontend is a Ruby on Rails web application deployed on Heroku, the backend is a Haskell application also deployed on Heroku, and the database is a MongoDB installation running on MongoHQ. Each of these components can be scaled to serve arbitrary numbers of students.

Theorem Prover

To verify student derivations, DeduceIt applies proof search over a term rewriting system. DeduceIt's theorem prover also includes a parser which is constructed dynamically on each theorem prover call, allowing instructors to define custom assignment syntax. The parser deconstructs the arguments of a theorem prover call into expression terms of the rewrite language before it passes these terms to the prover.

The Theorem Prover API

The DeduceIt prover is wrapped in a web server which can be queried via an API over the following arguments: *rulesets*, *assumptions*, *syntax*, and *conclusion*. To every query the prover will respond with either "proven", "unproven", or "syntax error." The frontend uses this API to assess the validity of each step in a derivation.

The *syntax* argument defines any new syntax on the default rewrite language. The rest of the parameters are used for proof search: DeduceIt tries to prove the provided *conclusion* expression using the set of rewrite rules defined in *rulesets* on the starting expressions in *assumptions*.

Proof Search

Proof search applies rewrite rules iteratively upon a group of expressions to generate new expressions. In general, a search may be considered either *forward*, starting from the known expressions and working toward the desired expressions, or *backward*, starting from the desired expressions and working toward the known expressions with an inverted set of rules. DeduceIt supports both kinds of search.

In practice, DeduceIt conducts one round of forward search and one round of backward search; the two rounds of search then meet in the middle, i.e., they check for terms generated in common. We introduce backward search to ensure DeduceIt will correctly check rewrite languages which allow the introduction of new symbols, e.g., predicate logic and the rule $\$a \rightarrow \$a \vee \$b$ (note here that $\$b$ is a variable in the rewrite rule which binds to *any expression*, so forward search cannot enumerate every possible value of $\$b$). Moreover, the meet-in-the-middle approach tends to be more effective than two rounds of only forward or backward search [14].

We find the system is limited to two rounds of search under reasonable time constraints, with an upper bound of 4 seconds for students interacting with a web application. At each iteration of search DeduceIt applies the set of available rewrite rules—defined by the collection of rulesets—non-deterministically and exhaustively. Notably, this delay falls into the third classification of Nielsen's work on response times, below the ten-second limit for keeping a user focused on his or her dialog with the system [20]. While the theorem prover could be made more efficient, the next section will demonstrate that in many cases this is unnecessary.

Proof Cache

DeduceIt introduces the idea of a *proof cache*, which takes advantage of other students' explorations to make the entire system run at more interactive speeds.

The proof cache tracks all responses that the frontend application receives from the prover. This cache is composed of many proof trees, one for each assignment, and together these proof trees track the aggregate history of every attempted derivation. Proof trees are a useful tool for managing assignments, but they can also, by acting as a cache, dramatically improve the overall performance of the system.

DeduceIt's bottleneck is in its prover; proof search is by far the most computationally expensive aspect of the system. A proof cache can limit the outbound queries to the prover and so, intuitively, increase the performance of the system.

To check a new derivation step using the proof cache, DeduceIt queries the proof tree for that derivation's assignment: if the new step already exists in the tree, then the system doesn't need to query the prover; it simply returns the stored result.

Besides the benefits of providing an infrastructure for hints and a summary of how well students are doing in an assignment, the proof cache has a very significant performance benefit for two reasons:

1. Most students complete derivations with steps that other students have used previously or future students will use.

2. It is more efficient to reuse the intermediate results of one student's derivation to check the validity of a derivation step than to check every derivation step with the prover.

An average prover call takes $810ms$, whereas a typical cache lookup takes only $23ms$. This improvement moves DeduceIt up a level on Nielsen's scale of responsiveness, under one second, to the level at which a user's flow of thought is uninterrupted [20]. We evaluate the performance impact of the proof cache more fully in the next section.

EVALUATION

DeduceIt hypothesizes that online derivation systems can scale to many students, across formal domains, and provide useful instructor analytics. To evaluate DeduceIt, we deployed it in the winter 2013 offering of Coursera's *Compilers* class, assigning several DeduceIt problems every week of the course. Out of the 7625 students who enrolled, 990 of them completed at least one DeduceIt assignment.

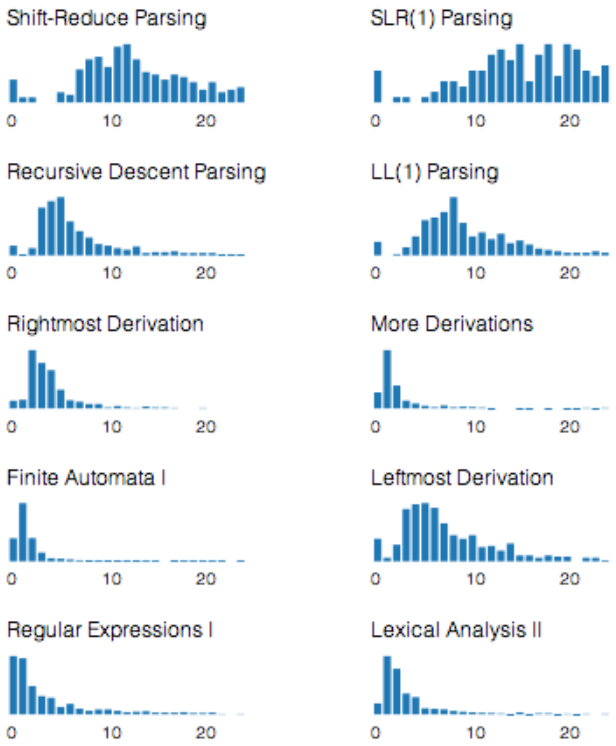

Figure 5: Normalized histograms show student time distributions for each assignment. The x-axis measures time in minutes. These time distributions are approximately normal with right skew and centered upon a fixed time value which tracks loosely with an assignment's difficulty.

In this section we provide evidence for two claims:

1. *Students successfully use DeduceIt for derivations*. We show through data analysis and user feedback that students engage with DeduceIt to solve problems.
2. *DeduceIt provides instructors with insights that translate into future course improvements*. Here we provide examples from our experience with the Compilers course.

We run several analyses over ten DeduceIt assignments, measuring the distribution of students' time spent on assignments and derivation steps, the success rate of students, the average number of student errors, and the performance impact of the DeduceIt's proof cache.

Assignment Time Distributions

First, we analyze the amount of time students spent on each assignment. These time distributions are approximately normal with right skew and centered upon a fixed time value which tracks loosely with an assignment's difficulty (Figure 5). For example, in the *Rightmost Derivation* assignment students are clustered around 4 minutes, and in the *Shift-Reduce Parsing* assignment they are clustered around 10 minutes; the *Rightmost Derivation* assignment is generally considered easier by students than *Shift-Reduce Parsing*. This relation holds true among the remainder of the assignments. To determine assignment difficulty in offline courses, instructors would typically need to guess or to ask students, who may have faulty recall. DeduceIt could easily be expanded to support knowledge-tracing [7] and provide instructors with more detailed analytics.

Rule Time Distributions

We also observe the time distribution which governs rule application across assignments. Supposing some rules are easier or more natural to apply than others, we can use DeduceIt to find rules associated with longer elapsed time between steps, revealing what parts of the course material are more likely to confuse students. On average, it takes students 163 seconds (roughly 3 minutes) between derivation steps.

Rule time distributions may also help instructors diagnose problematic derivation paths. For example, "Nonterm Y input end" is the rule associated with the longest elapsed time (1482s). This rule comes from *LL(1) Parsing* and represents an operation on a lookup table. Most students using "Nonterm Y input end" have entered a valid but misleading conclusion on the previous derivation step, which is an ideal candidate for hint annotation on the assignment proof tree (e.g., a hint which suggests, "While this step is correct, it doesn't lead to the solution.").

For a second use-case of this data, we examine the assignment *Regular Expressions I*. Here, the rules "Left Identity" and "Right Identity" are used by students most quickly, at about 100 seconds between applications. Students apply the "Unfold" rule most slowly, at approximately 400 seconds in its application. This suggests Unfold may be the most challenging property of regular expressions for students.

More generally, these analytics grant the instructors fine-grained insight into students' understanding, even on complex assignments. Instructors might use such information to better focus the emphasis of their lectures.

Student Success Rates

Next, we examine the success rates of students across assignments. These results are displayed in Table 1. Success rates are uniformly high, ranging from 93% to 99%, and these rates track reported assignment difficulty in a manner consistent with time distributions.

High success rates do not necessarily suggest that students are *learning* from these assignments, but we have received feedback that supports this idea. Several students told us that they found DeduceIt's derivation constraints (e.g., a list of available rules and givens, and step-by-step validation) useful for understanding the domain of a problem. One student mentioned, "Being able to step through the parsing actions and 'be' the parser and know immediately when I made a mistake was incredibly helpful in coming to truly understand how the different parsing styles work." However, these constraints beget certain tradeoffs: DeduceIt trades flexibility in assignment notation for ease of verification, and other students told us that assignment syntax got in the way, that DeduceIt "is difficult to use since you have to learn the syntax of each problem before actually applying what you know." An

Assignment Name	Description	Min	Total	Rules	Cache	Success	Valid	Syn.	Sem.
Regular Expressions I	Prove two regexes are equivalent	2	185	7	3412	96%	55%	11%	34%
Lexical Analysis II	Show the sequence of moves of a lexer	6	119	4	5963	97%	63%	6%	30%
Finite Automata I	Show that an automaton accepts a string	5	53	10	4589	98%	73%	3%	23%
Leftmost Derivation	Perform a leftmost derivation	10	97	3	10857	95%	56%	6%	38%
Rightmost Derivation	Perform a rightmost derivation	10	78	3	10857	98%	81%	1%	17%
More Derivations	Parse a sequence of roman numerals	6	61	9	9042	99%	81%	1%	18%
Recursive Descent Parsing	Show each state of a recursive descent parse	13	155	4	9465	96%	69%	1%	29%
LL(1) Parsing	Derive string using parse table	13	69	11	4750	99%	83%	1%	15%
Shift-Reduce Parsing	Show a shift-reduce parse of a string	20	151	8	6894	94%	85%	2%	12%
SLR(1) Parsing	Parse string using SLR(1) action table	15	99	33	4815	93%	60%	2%	36%

Table 1: *Min* is the shortest observed derivation path. *Total* is the observed number of unique derivations paths. *Rules* is the number of rules in the assignment. *Cache* is the size of the proof cache, e.g., the observed number of unique steps. *Success* is the success rate among students who attempted the assignment. *Valid* is the percentage of derivation steps which are valid. *Syn.* and *Sym.* are the percentages of observed syntax and semantic errors, respectively. (Note: Leftmost and Rightmost Derivation share the same subset of the proof cache.) Assignments are listed in the order they were assigned.

analysis of learning is outside the scope of this paper, but on balance we consider the feedback promising.

Student Error Rates

We next compute the rate of student mistakes, where an error is a conclusion which doesn't follow from the givens. Error rates among derivation steps for all the DeduceIt assignments are substantial—on average 29.4%, or approximately one error for every three correct steps—despite the high overall success rates reported in Table 1. We expect this result: students will always get some things wrong. Learning rarely takes place without error, whether indeliberate or exploratory as students use the system as a means of externalizing cognition [16]. Ideally, we might distinguish between two classes of error: errors which stem from a misunderstanding of course material, and errors which stem from misunderstandings (or exploratory actions) with DeduceIt.

To approximate the impact of these two classes of error, we divide student mistakes into two categories: syntax errors, and semantic errors (*Syn.* and *Sem.*, Table 1). Syntax errors arise from entering an expression which DeduceIt cannot parse, whereas semantic errors arise from parseable derivation steps which cannot be proven. While semantic errors are certain to contain faulty logic, this is not obviously true for syntax errors: a student might have intended to write a correct step and simply failed to use the system properly. We use the rates of student syntax errors as a proxy for measuring rates of student confusion with the DeduceIt system.

Across all assignments syntax errors are low, never exceeding 11%. Notably, the highest rate of syntax errors occurred in the first class assignment, *Regular Expressions I*, despite the fact that students generally considered this assignment the easiest. The syntax error rates of subsequent assignments never exceeded 6%, although these later assignments were much more difficult. In fact, in the last 6 assignments, the average syntax error rate was only 1.3%. This is likely a result of students learning to use the system. Low overall rates among syntax errors suggest that DeduceIt did not confuse students to the detriment of assignment completion and correctness.

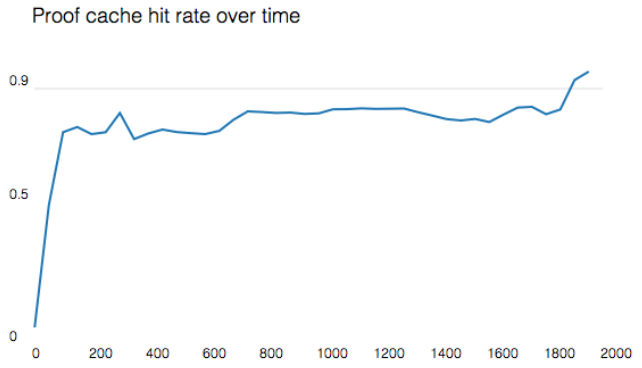

Figure 6: The x-axis measures the total number of derivation steps entered into the system. The y-axis measures the cache hit rate as a percentage.

Performance Impact of the Proof Cache

We test a hypothesis which provides the general intuition behind the proof cache: *most derivations contain steps that other students have previously used or will use in the future*. Of 50,099 total derivation steps in the class, only 5,407 (10%) of them are unique. So, 90% of derivation steps are reused, which indicates that a proof cache may be useful.

Next, we examine our second hypothesis: *it is more efficient to use the proof cache than to check every derivation step with the prover*. To test this hypothesis, we measured the overall proof cache hit rate. In Figure 6, we show the cache hit rate plotted against time. By the 1900th student interaction, the cache has reached a hit rate of 90%. Since the average theorem prover call takes $810ms$ and the average cache lookup takes $23ms$, we can compute an average

time of $810ms * 0.10 + 23ms * 0.90 = 101.7ms$. This represents a cost savings of 87% and qualitatively improves usability: running the cache allows DeduceIt to move up one level on Nielsen's hierarchy of responsiveness [20]. As DeduceIt adds more attempted derivation steps to the cache, the hit rate should continue to climb, albeit at a slower rate.

Further, the proof cache has benefits over memoization in a single user system. The proof cache hit rate is high because many students tend to enter the same derivation steps. A single student is less likely to enter the same derivation step more than once. Collectively, students reuse 90% of derivation steps, whereas on average, individual students reuse fewer than 17% of steps.

LIMITATIONS AND FUTURE WORK

In our Compilers course we encountered several notable limitations and tradeoffs in the DeduceIt system.

First, supporting arbitrary formalisms is a challenging design goal for a pedagogical system. DeduceIt achieves this goal by encouraging instructors to create *incomplete* instead of full formalisms. When necessary, complex rewrite rules that are difficult to implement generally are instead handled by assignment-specific rules, providing students with an illusion of working in a more general theory. In this way, DeduceIt trades completeness for expressiveness and simplicity, making it easier for instructors to create and debug assignments at the expense of totally covering a domain. However, other valuable systems might approach this tradeoff differently.

Second, a DeduceIt assignment must adopt the syntax of the default rewrite language. This works well for many kinds of assignments (e.g., algebra, basic calculus, predicate logic, regular expressions, and type checking), since instructors can still define custom functions and constants. However, in some assignments, natural notation differs more considerably from DeduceIt's rewrite language (and its possible extensions); this creates unnecessary friction on the derivation interface. For example, *LL(1) Parsing* uses a series of rules to define a lookup table: one student commented that some such rules were defined in "less than ideal notation." This is an issue we hope to fix in future versions of DeduceIt. Expressions and rules in the rewrite language do not need to be rendered as text on the student view; customizable HTML trees would allow instructors far more latitude in defining the appearance of assignment notation.

Third, while DeduceIt responds to students with information about semantic errors, syntax errors, correct steps, and annotated hints, some students felt the system wasn't offering them sufficient guidance. One student mentioned, "I spent more than half an hour trying to understand why DeduceIt was not accepting my derivations." Our ongoing work in hint generation addresses this concern. In the current system, not every incorrect step on the proof tree can be annotated by an instructor; an improved system might generate such hints automatically or notify instructors when several students reach in a dead end of the proof tree, encouraging them to add a hint in real-time.

Fourth, student interactions are constrained by proof search. Although instructors can mark rules as free, and this allows students to elide those rules in their derivations, the system is currently limited to two rounds of search. In future work we might improve the efficiency of our theorem prover or distribute proof search client-side, passing a larger share of computation onto the students' machines.

Fifth, although DeduceIt's proof cache provides a dramatic performance benefit, there are several ways the design of the cache can be improved. For instance, changing the cache representation from a tree into a directed acyclic graph (DAG) would be faster and more space efficient. Likewise, collapsing semantically equivalent derivation steps would also make lookups faster and improve the cache hit rate.

Finally, this paper does not directly address DeduceIt's effect on teaching outcomes: that is, how the system compares with traditional educational tools, and how it impacts student learning. We hope to study this question in future work.

CONCLUSION

For online education to succeed, it must move beyond simple multiple-choice problems to the kinds of open-ended assignments used by real courses. Peer grading holds promise but is difficult to apply in many domains [13, 17]. For many online courses that use formal, structured reasoning, automated systems can provide guidance through interactive assignments. We have presented DeduceIt, a system exploring this potential for student derivations in arbitrary formal domains.

DeduceIt enables instructors to set up reusable assignments for any kind of formal derivation, which can be completed by students at specifiable levels of abstraction. The system provides students with constraints — rules that may be applied only to the set of proven or given expressions — which provide support as the students work, and DeduceIt responds in real-time with information about the correctness of each derivation step: whether a mistake is due to a syntactic or semantic error, and hint annotations.

Students and instructors are successfully engaging with DeduceIt. Out of the 990 students who used DeduceIt in our online Compilers class, we observe an overall assignment completion rate of 96.5% and an average time to completion of 12.7 minutes over 10 assignments. Similarities in student proofs allow us to provide feedback to students at speeds that would be impossible using standalone systems. Further, instructors are using DeduceIt to perform course analytics that were quite difficult to track before. By analyzing assignment data using structures like DeduceIt's proof tree, instructors can determine which parts of an assignment are most difficult for students to complete, or which course concepts are least well understood.

DeduceIt suggests that online education might allow for assignments with complex reasoning requirements and multiple correct answers. Generalizing the system beyond formal derivations, or asking students to check and approve each others' claims, might create opportunities for more general instances of structured reasoning in domains such as chemistry, physics, or even law. DeduceIt and the compilers course are available at `http://www.coursera.org/course/compilers`.

ACKNOWLEDGMENTS

Special thanks to Joel Brandt and our colleagues at Stanford. We'd also like to thank Coursera, for helping us deploy our Compilers course; the Compilers course students, for their hard work and feedback; and finally our reviewers for their very useful suggestions.

REFERENCES

1. Coursera support documentation. `http://support.coursera.org`.
2. Bennett, R. E., and Bejar, I. I. Validity and automated scoring: It's not only the scoring. *Educational Measurement: Issues and Practice 17*, 4 (1998), 9–17.
3. Bennett, R. E., Steffen, M., Singley, M. K., Morley, M., and Jacquemin, D. Evaluating an automatically scorable, open-ended response type for measuring mathematical reasoning in computer-adaptive tests. *Journal of Educational Measurement 34*, 2 (1997), pp. 162–176.
4. Burstall, R. Proveeasy: Helping people learn to do proofs. *In Proc. ENTCS 2000* (2000), 16 – 32.
5. Cheang, B., Kurnia, A., Lim, A., and Oon, W.-C. On automated grading of programming assignments in an academic institution. *Comput. Educ. 41*, 2 (2003), 121–131.
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. Maude as a metalanguage. *In Proc. WRLA 1998 15* (1998).
7. Corbett, A., and Anderson, J. Knowledge tracing: Modeling the acquisition of procedural knowledge. *In Proc. UMUAI 1994* (1994), 253–278.
8. Corbett, A. T., and Anderson, J. R. Locus of feedback control in computer-based tutoring: impact on learning rate, achievement and attitudes. In *Proc. CHI '01* (2001).
9. Gallien, T., and Oomen-Early, J. Personalized versus collective instructor feedback in the online courseroom: Does type of feedback affect student satisfaction, academic performance and perceived connectedness with the instructor? *International Journal on E-Learning 7*, 3 (2008), 463–476.
10. Hearst, M. The debate on automated essay grading. *Intelligent Systems and their Applications, IEEE 15*, 5 (2000), 22–37.
11. Heffernan, N. T., Koedinger, K. R., and Razzaq, L. Expanding the model-tracing architecture: A 3rd generation intelligent tutor for algebra symbolization. *Int. J. Artif. Intell. Ed.* (2008), 153–178.
12. Hernan-Losada, I., Pareja-Flores, C., and Velazquez-Iturbide, A. Testing-based automatic grading: A proposal from bloom's taxonomy. *In Proc. ICALT 2008* (2008), 847–849.
13. Huang, S.-W., and Fu, W.-T. Enhancing reliability using peer consistency evaluation in human computation. *In Proc. CSCW 2013* (2013), 639–648.
14. Kaindl, H., and Kainz, G. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research 7* (1997), 283–317.
15. Kaufmann, M., and Moore, J. S. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng. 23*, 4 (1997), 203–213.
16. Kirsh, D., and Maglio, P. P. On Distinguishing Epistemic from Pragmatic Action. *Cognitive Science 18*, 4 (1994), 513–549.
17. Kulkarni, C., Pang, K., Le, H., Chia, D., Papadopoulos, K., Cheng, J., Koller, D., and Klemmer, S. Peer and self assessment in massive online design classes. *ACM TOCHI* (2013).
18. Lapets, A., Skowyra, R., Bassem, C., Kfoury, A., and Bestavros, A. Towards an infrastructure for integrated accessible formal reasoning environments. *In Proc. UITP 2012*.
19. Mart-Oliet, N., and Meseguer, J. Rewriting logic: Roadmap and bibliography. *J. Log. Algebr. Program. 81* (2001).
20. Nielsen, J. *Usability Engineering*. Morgan Kaufmann, 1993.
21. Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
22. Pappano, L. Massive open online courses are multiplying at a rapid pace. `http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html`.
23. Paulin-Mohring, C. Inductive definitions in the system coq rules and properties. *TLCA 1993* (1993).
24. Paulson, L. C. The foundation of a generic theorem prover. *Journal of Automated Reasoning 5* (1989).
25. Ritter, S., Towle, B., Murray, R., Hausmann, R., and Connelly, J. A cognitive tutor for geometric proof. *In Prof. ITS 2010* (2010), 453–453.
26. Schleimer, S., Wilkerson, D. S., and Aiken, A. Winnowing: local algorithms for document fingerprinting. *In Proc. ACM SIGMOD 2003* (2003), 76–85.
27. Suppes, P. The next generation of interactive theorem provers. *7th International Conference on Automated Deduction 170* (1984), 303–315.
28. Suppes, P. Student use of an interactive theorem prover. *Contemporary Mathematics 29* (1984).
29. Tosic, M., and Nejkovic, V. Trust-based peer assessment for virtual learning systems. *In Proc. SocInfo 2010* (2010), 176–191.
30. VanLehn, K. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist 46*, 4 (2011), 197–221.
31. Windsteiger, W. Theorema 2.0: A graphical user interface for a mathematical assistant system. *CEUR Workshop Proceedings* (2012), 73–81.