Deduce It! Automatically Guiding Students through Derivations in a Massively Open On-line Course

Ethan Fast, Colleen Lee, Daphne Koller, Alex Aiken
Stanford University
{ethaen, clee0, koller, aiken}@cs.stanford.edu

ABSTRACT
This paper presents DeduceIt, a derivation checking system for massively open on-line courses. DeduceIt is a general system for creating and grading student assignments—it guides students through exercises which can be defined by instructors across arbitrary formal domains—but unlike other interactive theorem provers, it presents a web-based interface accessible to non-expert users, and it can be used by instructors and students without prior training. DeduceIt also introduces the idea of 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 describe our experience using DeduceIt and evaluate the system with data collected from thousands of students in an on-line class.

Author Keywords
What keywords to use?

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

General Terms
Human Factors; Design; Measurement.

INTRODUCTION
As instructors begin to teach hundreds of thousands of students in on-line classrooms, it becomes increasingly difficult for them to grade complex assignments and provide students with personal feedback. Here, the rise of massively open on-line courses (MOOCs) presents a departure from the standard educational model: TAs are not designed to scale.

When the size of an on-line course increases, the attention of its instructors and TAs necessarily becomes more scarce. They are unable to provide large numbers of students—to date the largest MOOC has enrolled more than 180,000—with feedback on their assignments; much less to present it in real-time. In particular, grading becomes intractable unless it is automated, which it often cannot be, particularly in complex domains where students may construct correct solutions in many different ways.

These constraints are important limitations to the MOOC model. Students tend to learn better when subject to tight feedback loops [?], and many subjects, if taken in a traditional course, grade students on proofs or derivations, where there are many ways to describe a correct answer.

MOOC platforms have attempted to automate away some of these issues, but providing thousands of students with *constructive* and *real-time* feedback has remained an unsolved problem. Coursera and Udacity give instructors out-of-the-box tools to build assignments and quizzes with automated grading features, but these tools lack flexibility: evaluation is constrained to discrete, instructor-specified solutions and offers students a limited feedback loop (e.g. "correct," or "incorrect"). For technical, mathematical, or scientific domains where assignment solutions are often *derivations*—many-staged and prone to individual variation—this limitation is particularly problematic.

To address these limitations we present:

- The *DeduceIt* system: A web-based framework in which it is possible to specify a general class of derivation exercises, yet which remains accessible non-expert users. DeduceIt provides students with constructive and real-time feedback.
- The idea of a *proof cache*: A novel data structure which records the history of every attempted derivation, reusing computations from previously derived steps to provide predictable performance under heavy load and enable real-time feedback on student progress. We show this cache can increase system efficiency by several orders of magnitude.
- An *empirical evaluation* of DeduceIt: A report on the data we have collected from a massive on-line class with thousands of students doing dozens of exercises.

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

RELATED WORK

Submitted for review.

There is quite a bit of related work on proof solving systems and derivation helpers. Several existing tools already provide users with rich and automated feedback as they work through a derivation, but these tools are principally designed for expert users and unsuited to the MOOC audience. For instance, the X tool and the Y tool might allow experts to explore problem domains in predicate logic or mathematics, but their interfaces are not designed to be accessible to lay students, or even to a typical course instructor.

Moreover, automatic feedback and grading systems are often domain-specific and do not grant an instructor sufficient flexibility in defining a set of disparate problems. Sometimes this approach works well: automated grading systems have long evaluated student program code in computer science departments (and now also commercially). But we are concerned here with more general systems.

MOTIVATING EXAMPLE

To better motivate how DeduceIt works and what problems it solves, we begin with an example assignment from the perspective of a MOOC student in Coursera's Compilers class. This student has recently finished her first week in the course, where she has learned among other things about the properties of regular expressions.

In this particular assignment we ask her to show equality between two regular expressions. Assuming $+$, $*$, and . as union, kleen closure, and concatenation, she must prove:

$$(A+B)* \equiv ((B+A).(B+A)*) + \epsilon$$

More concretely, the assignment asks our student to derive a particular expression. DeduceIt labels this the *goal*:

$$((B+A).(B+A)*) + \epsilon$$

To show the goal expression, our student must start from certain other expression(s). DeduceIt calls these the *givens*, and here they consist of the single starting expression:

$$(A+B)*$$

Our student also sees a description of the assignment and—perhaps most importantly—the available *rules* she may apply on the given expressions (or in some cases, sub-expressions of these givens) in order to derive the final, goal expression. In this assignment these rules specify several equality preserving transformations of regular expressions: Commutativity, Distributivity, Right Distributivity, Unfold, Identity, Left Identity, and Absorption. However, rules may not always preserve equality in general.

To complete a step in the derivation, our student may apply a rule to one or more given expressions, producing a new *conclusion*. As the derivation expands, the conclusion of each previous step is added to the set of givens. DeduceIt can provide several different kinds of feedback on each step, depending on the state of the derivation. Next, we will walk through a few such steps.

Completing a Derivation

As she reads over the assignment, our student realizes she has only been given one expression from which to start her derivation: $(A + B)$. She reasons this must go in the givens input field on the first step. (In fact, this is the only expression DeduceIt allows her to enter in that field). Now she has two input fields left. One asks for a rule; the other a conclusion. To come up with a conclusion, she must decide which rule to apply, and where to apply it. She supposes "Unfold" looks promising and tries it on the full starting expression, computing:

$$((A+B).(A+B)*) + \epsilon$$

This expression looks pretty close to her goal, so she enters it in the conclusion input field. After selecting "Unfold" as her rule, she tells DeduceIt to update her derivation. There is a brief computation, then the derivation returns with her previous step highlighted in green: it was successful. A new step lies below her previous entry—this one is empty—querying her for the next line of the derivation.

Our student now considers her previous conclusion: since it 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. While she is pretty sure one property of regular expressions does allow for this kind of transformation, she can't remember what it is called. So she looks through the assignment rules and sees it: Commutativity! Regular expressions are commutative under union.

Mentally, she applies Commutativity to each of the two $A + B$ sub-expressions, then enters her result in the conclusion input box on the next line of the derivation. This conclusion is identical to the goal expression:

$$((B+A).(B+A)*) + \epsilon$$

She selects "Commutativity" from the rule input box, chooses the conclusion from her previous step $((B+A).(B+A)*)+\epsilon$ as her new given, and tells DeduceIt to update her proof. DeduceIt checks her derivation, infers through proof search that she means to apply Commutativity twice—on the two appropriate sub-expressions—and responds that the derivation is correct: she has completed this assignment.

HOW DOES A DEDUCEIT ASSIGNMENT WORK?

We start with some useful background information about term rewrite systems which some readers may find necessary to understand DeduceIt, then we present an overview of DeduceIt's two primary assignment interfaces: creation and completion.

Term Rewriting Systems

DeduceIt uses a term rewriting system to verify student derivations. In general, rewriting systems consist of a set of objects and relations which define transformations on those objects. Each such transformation is called a rewrite rule. A term rewriting system is a rewriting system which operates on expressions with nested sub-expressions.

Term rewrite rules define transformations which occur between terms. The left hand side of a rule must match the entire term for a rule to be applied. 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 $x + 1 = y$ to produce $x = y - 1$.

While DeduceIt's derivation checker operates primarily as a term rewriting system, it differs from a standard system in several respects. First, DeduceIt supports an assumption of equality in certain rewrites, where specified by an instructor. Rewrite rules which assume equality can be applied recursively upon term subexpressions; the entire term need not match the rule, but only a subexpression of that term. This makes DeduceIt more efficient at verifying certain common derivations. Second, DeduceIt supports the dynamic evaluation of basic mathematical expressions. For instance, if DeduceIt is provided with a set of rules which govern basic algebra and the expression $1+x = 3$, it can derive $x = 2$ by using the dynamic rule $integer - integer \rightarrow eval(integer - integer)$.

Assignment Creation

The assignment creation page is the most complex portion of DeduceIt's interface. To create an assignment, an instructor must specify four things:

1. A rewrite language: DeduceIt provides every assignment with 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 may optionally augment the language with extra syntax for functions and constants.
2. The Rulesets: These are 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 which both describe 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.

A student works through an assignment by starting from the given expressions and applying valid transformations until she has reached the goal expression.

The Language

DeduceIt's default language should be familiar to anyone who has used an advanced calculator. It ships with the unary operators ˜ and $-$, and the binary operators ., &, |, ,, $*$, $\backslash$, $+$, $-$, $=$, $\neq$, $\leq$, $\geq$, $<$, $>$, $:=$, and $\rightarrow$ which we list in order of precedence. DeduceIt also supports variables (these may only be used when defining rewrite rules), symbols, and integers. For example, here are three legal expressions in the default rewrite language are:

$$\$p, (\$p => \$q) \rightarrow \$q$$

$$a.b.b.a$$

$$x + 2 = y$$

Note that the operators used in these expressions have no meaning without an accompanying set of rules; they simply dictate the construction of an expression parse tree.

Instructors may add two kinds of syntax to DeduceIt's standard rewrite language: constants and variable argument 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 DeduceIt adopts a notation inspired by latex. Constants appear in the language 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. $\backslash sin\{x\}$. Figure 0 shows DeduceIt's custom syntax creation view.

Rulesets

Rulesets are the most complicated part of an assignment. They define the sets of valid transformations which a student may use in a derivation. Each ruleset has a name, a student-facing description, a set of rewrite rules, and an optional set of constraints. The name and description of a ruleset are all a student sees when using DeduceIt, and these fields make an assignment more accessible; students do not need to understand rewrite rules to use the system.

Each rewrite rule in a ruleset is either a strict rewrite rule or an equality. To apply a strict rewrite rule, its left side must match exactly on an expression, wheres an equality may be applied to an expression or any of its subexpressions. Take as an example the expression $1 + x * x = 5$ and the strict rewrite rule $x * x \rightarrow x^2$; here the rewrite rule cannot be applied, since its left side doesn't bind with the entire expression. However, a similar rewrite rule which has been defined with an assumption of equality, $x * x := x^2$, will bind on $x * x$ and produce the transformation $1 + x^2 = 5$. In this way equalities are more powerful than strict rewrite rules, and for many assignments a single equality can take the place of many strict rules. This eases the burden on an instructor and usually speeds up proof search; the more rules an assignment has, the slower search tends to be.

Various constraints may be defined on the variable terms used in the ruleset. They operate in a manner similar to conditional rewrite rules, but are somewhat less general; we use only the condition of containment—whether a variable contains certain other terms—as a condition on ruleset application.

A ruleset may also be either "required" or "free." While a required ruleset must be named explicitly when a student uses it in a derivation, a "free" ruleset may be elided. Behind the scenes, DeduceIt will attempt to fill in such "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. For example, a student may want to rearrange terms in a mathematical expression as part of some broader rule application. DeduceIt's "free" rules allow the student to skip over these steps.

In figure 0 we show DeduceIt's Ruleset creation view.

Assignment Completion

The student view

Every submission should begin with an abstract of about 150 words, followed by a set of keywords. The abstract and keywords should be placed in the left column of the first page under the left half of the title. The abstract should be a concise statement of the problem, approach and conclusions of the work described. It should clearly state the paper's contribution to the field of HCI.

The first set of keywords will be used to index the paper in the proceedings. The second set are used to catalogue the paper in the ACM Digital Library. The latter are entries from the ACM Classification System [**?**]. In general, it should only be necessary to pick one or more of the H5 subcategories, see `http://www.acm.org/class/1998/ccs98.html`

Normal or Body Text

Please use a 10-point Times Roman font or, if this is unavailable, another proportional font with serifs, as close as possible in appearance to Times Roman 10-point. The Press 10-point font available to users of Script is a good substitute for Times Roman. If Times Roman is not available, try the font named Computer Modern Roman. On a Macintosh, use the font named Times and not Times New Roman. Please use sans-serif or non-proportional fonts only for special purposes, such as headings or source code text.

First Page Copyright Notice

Leave 3 cm (1.25 in.) of blank space for the copyright notice at the bottom of the left column of the first page. In this template a floating text box will automatically generate the required space. Note however that the text box is anchored to the **ABSTRACT** heading, so if that heading is deleted the text box will disappear as well. You can replace the default copyright notice by uncommenting the \toappear block at the beginning of the document and inserting your own text, for example, for versions under review.

Subsequent Pages

On pages beyond the first, start at the top of the page and continue in double-column format. The two columns on the last page should be of equal length.

Figure 1. With Caption Below, be sure to have a good resolution image (see item D within the preparation instructions).

Objects	Caption — pre-2002	Caption — 2003 and afterwards
Tables	Above	Below
Figures	Below	Below

Table 1. Table captions should be placed below the table.

References and Citations

Use a numbered list of references at the end of the article, ordered alphabetically by first author, and referenced by numbers in brackets [**?**, **?**, **?**, **?**]. For papers from conference proceedings, include the title of the paper and an abbreviated name of the conference (e.g., for Interact 2003 proceedings, use *Proc. Interact 2003*). Do not include the location of the conference or the exact date; do include the page numbers if available. See the examples of citations at the end of this document. Within this template file, use the `References` style for the text of your citation.

Your references should be published materials accessible to the public. Internal technical reports may be cited only if they are easily accessible (i.e., you provide the address for obtaining the report within your citation) and may be obtained by any reader for a nominal fee. Proprietary information may not be cited. Private communications should be acknowledged in the main text, not referenced (e.g., "[Robertson, personal communication]").

SECTIONS

The heading of a section should be in Helvetica 9-point bold, all in capitals. Use Arial if Helvetica is not available. Sections should not be numbered.

Subsections

Headings of subsections should be in Helvetica 9-point bold with initial letters capitalized. For sub-sections and sub-subsections, a word like *the* or *of* is not capitalized unless it is the first word of the heading.)

Sub-subsections

Headings for sub-subsections should be in Helvetica 9-point italic with initial letters capitalized. Standard \section, \subsection, and \subsubsection commands will work fine.

FIGURES/CAPTIONS

Place figures and tables at the top or bottom of the appropriate column or columns, on the same page as the relevant text (see Figure 1). A figure or table may extend across both columns to a maximum width of 17.78 cm (7 in.).

Captions should be Times New Roman 9-point bold. They should be numbered (e.g., "Table 1" or "Figure **??**"), centered and placed beneath the figure or table. Please note that the words "Figure" and "Table" should be spelled out (e.g., "Figure" rather than "Fig.") wherever they occur.

Papers and notes may use color figures, which are included in the page limit; the figures must be usable when printed in black and white in the proceedings. The paper may be accompanied by a short video figure up to five minutes in length. However, the paper should stand on its own without the video figure, as the video may not be available to everyone who reads the paper.

LANGUAGE, STYLE AND CONTENT

The written and spoken language of SIGCHI is English. Spelling and punctuation may use any dialect of English (e.g., British, Canadian, US, etc.) provided this is done consistently. Hyphenation is optional. To ensure suitability for an international audience, please pay attention to the following:

- Write in a straightforward style.
- Try to avoid long or complex sentence structures.
- Briefly define or explain all technical terms that may be unfamiliar to readers.
- Explain all acronyms the first time they are used in your text—e.g., "Digital Signal Processing (DSP)".
- Explain local references (e.g., not everyone knows all city names in a particular country).
- Explain "insider" comments. Ensure that your whole audience understands any reference whose meaning you do not describe (e.g., do not assume that everyone has used a Macintosh or a particular application).
- Explain colloquial language and puns. Understanding phrases like "red herring" may require a local knowledge of English. Humor and irony are difficult to translate.
- Use unambiguous forms for culturally localized concepts, such as times, dates, currencies and numbers (e.g., "1-5-97" or "5/1/97" may mean 5 January or 1 May, and "seven o'clock" may mean 7:00 am or 19:00). For currencies, indicate equivalences—e.g., "Participants were paid 10,000 lire, or roughly $5."
- Be careful with the use of gender-specific pronouns (he, she) and other gendered words (chairman, manpower, man-months). Use inclusive language that is gender-neutral (e.g., she or he, they, s/he, chair, staff, staff-hours, person-years). See [**?**] for further advice and examples regarding gender and other personal attributes.
- If possible, use the full (extended) alphabetic character set for names of persons, institutions, and places (e.g., Grønbæk, Lafreniére, Sánchez, Universität, Weißenbach, Züllighoven, Århus, etc.). These characters are already included in most versions of Times, Helvetica, and Arial fonts.

PAGE NUMBERING, HEADERS AND FOOTERS

Please submit your anonymous version for reviewing with page numbers centered in the footer. These must be removed in the final version of accepted papers, as page numbers, headers, and footers will be added by the conference printers. Comment out the \pagenumbering command at the top of the document to remove page numbers.

PRODUCING AND TESTING PDF FILES

We recommend that you produce a PDF version of your submission well before the final deadline. Your PDF file must be ACM DL Compliant. The requirements for an ACM Compliant PDF are available at: `http://www.sheridanprinting.com/typedept/ACM-distilling-settings.htm`.

Test your PDF file by viewing or printing it with the same software we will use when we receive it, Adobe Acrobat Reader Version 7. This is widely available at no cost from [**?**]. Note that most reviewers will use a North American/European version of Acrobat reader, which cannot handle documents containing non-North American or non-European fonts (e.g. Asian fonts). Please therefore do not use Asian fonts, and verify this by testing with a North American/European Acrobat reader (obtainable as above). Something as minor as including a space or punctuation character in a two-byte font can render a file unreadable.

BLIND REVIEW

For archival submissions, CHI requires a "blind review." To prepare your submission for blind review, remove author and institutional identities in the title and header areas of the paper. You may also need to remove part or all of the Acknowledgments text. Further suppression of identity in the body of the paper and references is left to the authors' discretion. For more details, see the submission guidelines and checklist for your submission category.

CONCLUSION

It is important that you write for the SIGCHI audience. Please read previous years' Proceedings to understand the writing style and conventions that successful authors have used. It is particularly important that you state clearly what you have done, not merely what you plan to do, and explain how your work is different from previously published work, i.e., what is the unique contribution that your work makes to the field? Please consider what the reader will learn from your submission, and how they will find your work useful. If you write with these questions in mind, your work is more likely to be successful, both in being accepted into the Conference, and in influencing the work of our field.

ACKNOWLEDGMENTS

We thank CHI, PDC and CSCW volunteers, and all publications support and staff, who wrote and provided helpful comments on previous versions of this document. Some of the references cited in this paper are included for illustrative purposes only. **Don't forget to acknowledge funding sources as well**, so you don't wind up having to correct it later.