Structure and Term Prediction for Mathematical Text

Oleksandr Polozov
Kyiv, Ukraine
NTUU "KPI"
polozov.alex@gmail.com

Sumit Gulwani
Redmond, WA, USA
Microsoft Research
sumitg@microsoft.com

Sriram Rajamani
Bangalore, India
Microsoft Research
sriram@microsoft.com

Abstract

Mathematical text is too cumbersome to write because of the need to encode a tree structure in a left to right linear order. This paper defines two novel problems, namely structure prediction from unstructured representation and sequence prediction within a mathematical session, to help address mathematical text entry. The effectiveness of our approach relies on the fact that normal mathematical text is highly *symmetric*.

Our solution to the structure prediction problem involves defining a ranking measure that captures symmetry of a mathematical term, and an algorithm for efficiently finding the structure with the highest rank. Our solution to the sequence prediction problem involves defining a domain-specific language for term transformations, and an inductive synthesis algorithm that can learn the likely transformation from the first couple of sequence elements. Our tool is able to predict the correct structure in 63% of the cases, and save more then half of sequence typing time in 52% of the cases on our benchmark collection. We argue that such algorithms are important components of human-computer interfaces for inputting mathematical text, be it through speech, keyboard, touch or multimodal interfaces.

Introduction

Inputting mathematical text into a computer remains a painful task, despite several improvements to document editing systems over the years. Currently, there are two main ways to input mathematical text. The first kind, pioneered by Don Knuth's TEXsystem, uses a markup language where the user encodes the expression structure using prefix notation, and the system compiles it and produces the visual rendering. The markup language, while extremely systematic and general, leads to unreadable text in encoded form, and is hard to use, even for an expert mathematician without detailed knowledge about TEX. The second kind, represented by commercial document editors such as Microsoft Word, provides WYSIWYG input facilities which are more intuitive, but require users to change cursor position several times, and switch back and forth between keyboard and mouse input. Despite these approaches, software does not provide the convenience comparable to writing mathematics using paper-and-pen.

In this paper, we derive inspiration from other domains, where computers provide significant advantages over composing text using paper-and-pen. Microsoft Visual Studio and Eclipse editors for programming provide a capability called "intellisense" where the software analyzes the input on the fly and provides intelligent suggestions for completion as the user types, not only saving users valuable time, but also providing a compelling advantage for using the software over paper-and-pen. Scorewriter software, such as Guitar Pro, Finale or Sibelius, provide similar facilities for input and display of music. Inspired by these editors from other domains, we believe that mathematical text processing software can significantly improve using automation to understand and analyze the text as it is input without requiring extensive markup, and intellisense capabilities to predict what the user is likely to input next. We believe that such intellisense algorithms are important components of human-computer interfaces for inputting mathematical text into a computer, be it through speech, touch, keyboard or multimodal interfaces.

Our observations. By exploring large corpus of mathematical texts, we have come up with two simple observations. First, we find that mathematical expressions that appear in real-life scenarios or in textbooks have some sense of inherent symmetry (or even "beauty"). It is unlikely that a term in an expression is highly asymmetric or unbalanced. Symmetric sub-terms of a single term are often similar to each other as well. Using this observation, we present an algorithm to predict the intended structure of mathematical expressions without requiring extensive markup input (in particular, explicit parenthesization) from the user. Second, we find that mathematical text is often organized into sessions, each consisting of mutually related expressions with an inherent progression. Using this observation, we present an algorithm to predict what sub-term the user is likely to input next. We have implemented both these algorithms and evaluated the efficacy of these algorithms in improving the productivity of mathematics content creation.

Our contributions. This paper makes the following contributions:

1. We define and motivate two new problems for help with creating mathematical text: inference of structure in a flat-

tened mathematical term, and prediction of terms in a term sequence.

2. We define a ranking measure for mathematical terms, and present an efficient algorithm (which exploits the structural dependence of the underlying grammar of the mathematical terms) to infer highest-ranked structure in a flattened term.
3. We define a language of term transformations, and present an efficient algorithm to infer term transformations from examples, and use this algorithm for sequence prediction.
4. We present experimental results that illustrate the efficacy of our techniques for content creation.

Problem Definition

Structure prediction. The first problem we are interested in solving is to infer the likely structured representation from a linear sequence of terms in infix format.

Problem 1. *Given an infix representation $t_1, \ldots, t_n$ of a term τ, infer likely structured representation of term τ.*

In particular, we do not expect the user to disambiguate grouping using parenthesis. For example, consider the term:

$$(\operatorname{cosec} A - \sin A)(\sec A - \cos A)(\tan A + \cot A) = 1 \quad (1)$$

Suppose the user enters this without parenthesis as:

$$\operatorname{cosec} A - \sin A \cdot \sec A - \cos A \cdot \tan A + \cot A = 1$$

Without parenthesis, its infix representation is the sequence of tokens: cosec, A, $-$, sin, A, $\cdot$, sec, A, $-$, cos, A, $\cdot$, tan, A, $+$, cot, A, $=$, 1. This sequence is what we typically pronounce, denoting the parentheses implicitly with pauses and intonation. There exist many ways to insert parenthesis into this sequence to get a valid expression (valid in terms of a grammar of mathematical expressions). Several of them are listed below:

$$\operatorname{cosec} A - \sin A \sec A - (\cos A \tan A + \cot A) = 1$$
$$\operatorname{cosec} A - \sin A \sec A - \cos A \tan A + \cot A = 1$$
$$\operatorname{cosec} A - (\sin A \sec A - \cos A \tan A) + \cot A = 1$$
$$(\operatorname{cosec} A - \sin A) \sec A - \cos A(\tan A + \cot A) = 1$$
$$(\operatorname{cosec} A - \sin A)(\sec A - \cos A)(\tan A + \cot A) = 1$$

However, the last one (which is the correct parenthesizing shown in Equation 1) "feels" more symmetric than the others. We formalize this by associating a rank with parenthesizing, where the ranking scheme favors symmetric terms over asymmetric ones. We present details of the ranking scheme and an algorithm for automatically predicting structure in section .

Sequence prediction. While structure prediction aims to recover inherent symmetric structure in most mathematical terms, sequence prediction aims to recover inherent progression in sequences of terms.

Problem 2. *Given a few initial terms $\tau_1, \ldots, \tau_k$ of a sequence σ and a relevant context (previous sequences $\tilde{\sigma}_1, \ldots, \tilde{\sigma}_s$), predict the rest of the terms $\tau_{k+1}, \ldots, \tau_n$ of the sequence σ.*

For example, consider the equation:

$$\tan A \cdot \tan 2A \cdot \tan 3A = \tan A + \tan 2A + \tan 3A \quad (2)$$

The arguments of the trigonometric function in the LHS follow the sequence: $A, 2A, 3A$. Thus, if the user has already typed "$\tan A \cdot \tan 2A \cdot$", suggesting "$\tan 3A$" for auto-completion could save about 1/3 of typing time for the LHS.

A related, but different problem, arises in the RHS of the above equation. Assume that the user types the RHS after typing the LHS. The RHS of this equation is a transformation of LHS, where operator "$\cdot$" is replaced with the operator "$+$". Thus, after the user has typed "$\tan A +$", we desire that our system suggests "$\tan 2A + \tan 3A$" as a suggestion for auto-completion, saving about 2/3 of typing time.

As another example, consider the Vandermonde matrix, used in polynomial interpolation:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^{n-1} \end{pmatrix} \quad (3)$$

Each of its elements can be expressed with the following formula: $x_i^j, i = 1 \ldots m, j = 0 \ldots n-1$. As the user types the first few elements of the first row, we wish to recognize this pattern and offer suggestions to auto-complete the first row. Further, after the first row has already been typed, we wish the remainder of the second row to be suggested as likely completion just after user has typed x_2.

Another example of a symbolic matrix with an inherent progression pattern is the following:

$$\begin{pmatrix} yz - x^2 & zx - y^2 & xy - z^2 \\ zx - y^2 & xy - z^2 & yz - x^2 \\ xy - z^2 & yz - x^2 & zx - y^2 \end{pmatrix} \quad (4)$$

Here every element of the matrix is of the form: $e_1e_2 - e_3^2$, where e_1, e_2, e_3 are the variables from the set $\{x, y, z\}$. Note that the values for each specific variable e_i follow the pattern $x \to y \to z \to x \to \ldots$ cyclically within each row. As soon as the user inputs the first two elements of the first row, we desire to suggest the third element as an auto-completion. Further, once the user has finished inputting the first row, and inputs the first element of the second (or third) row, we desire to suggest the rest of the row as auto-completion.

Intellisense and interfaces. Intellisense algorithms to solve the above problems are important components of user interfaces to input mathematics into the computer, regardless of whether we use typing or speech or touch to input mathematics. In both typing and speech interfaces, it is more natural to think about an equation as a linear sequence of tokens, without explicitly mentioning parentheses. Further, users make mistakes (see (MathWorks)) if we ask them to parenthesize expressions. By inferring parenthesization and structure, we can prevent these mistakes. In addition, we can save user effort if the computer can accurately predict and complete equations on behalf of the user. In this paper, we concentrate on evaluating our algorithms on material from widely used mathematical texts and empirically measuring the potential for productivity gains. Building complete user interfaces incorporating these algorithms, and performing user studies using these interfaces are left for future work.

$$
\begin{array}{rl}
Eqn & := Expr = Expr \\
Expr & := Poly \mid \text{sqrt}(Poly) \\
Poly & := Poly + Poly \\
 & \mid Poly - Poly \\
 & \mid Poly * Poly \\
 & \mid Poly \,/\, Poly \\
 & \mid Poly \,\hat{}\, Const \\
 & \mid (Poly) \\
 & \mid Trig \mid Term \\
Trig & := F(Arg)
\end{array}
\qquad
\begin{array}{rl}
F & := \sin \mid \cos \\
 & \mid \tan \mid \cot \\
 & \mid \sec \mid \text{cosec} \\
Arg & := Term \mid Local \\
Local & := Term + Term \\
 & \mid Term - Term \\
 & \mid Const * Arg \\
 & \mid Arg \,/\, Const \\
Term & := \texttt{var} \mid Const \\
Const & := \texttt{const}
\end{array}
$$

Figure 1: Grammar G of expressions

Structure Prediction Algorithm

Our core idea is to define a ranking measure over ASTs so that symmetric ASTs get higher rank than asymmetric ones, and our hypothesis (which we empirically validate) is that higher ranked ASTs are more likely to occur in mathematical texts than lower ranked ones. Further, we use hierarchy between non-terminals in the grammar to optimize the search for higher ranked ASTs to enable scaling.

Grammar. Figure 1 shows the grammar G of expressions we use. The grammar allows algebraic expressions over trigonometric functions. Square root is allowed at the top level of the AST (using the nonterminal $Expr$), followed by algebraic operations at the next level (using the nonterminal $Poly$), followed by trigonometric functions (such as sin, cos, etc.) at the next level (using the nonterminal $Trig$). Arguments of trigonometric functions are restricted to linear expressions, and exponents are restricted to constants. We find that this grammar is expressive enough to represent all the expressions we analyzed from well known mathematics textbooks. Note that the grammar is ambiguous, particularly due to the productions from the nonterminal $Poly$, and parentheses are needed to disambiguate expressions. The goal of our algorithm is to disambiguate among possible terms by defining an appropriate ranking measure for terms. Though the description of the algorithm below is given with respect to the grammar G, we observe that the concepts and algorithms are general, and applicable to other grammars as well.

Next, we define a similarity metric and use it to define a ranking measure for terms. Our desire is that terms with similar sub-terms should be ranked higher. Further, we want the terms with highly ranked sub-terms to get higher ranks.

Definition 1 (Similar terms)**.** *Two terms τ_1 and τ_2 derivable from grammar G are defined to be **similar** iff their ASTs have the same tree structure, and one can be exactly obtained from another using repeated application of the following transformations: (1) change of one variable to another, (2) change of one constant to another, (3) change of one trigonometric operation to another, (4) change of "$+$" to "$-$" or vice versa, and (5) swapping of the arguments of a binary operator.*

Definition 2 (Similarity measure)**.** *Let τ_1 and τ_2 be similar terms, and let c be the number of transformations needed to transform τ_1 to τ_2. We define **similarity measure** of τ_1 and τ_2 as $Sim(\tau_1, \tau_2) \stackrel{\text{df}}{=} 1/c$.*

Definition 3 (Rank)**.** *Let τ be a term derivable from grammar G. Let $\tilde{\tau}_1, \ldots, \tilde{\tau}_n$ be the top-level sub-terms of τ (children nodes of a root node of τ's AST). The **rank** of τ is defined as follows:*

$$
Rank(\tau) \stackrel{\text{df}}{=} 1 + \sum_{i=1}^{n} \left(Rank(\tilde{\tau}_i) \times \left(1 + \sum_{\substack{j=1 \\ j \neq i}}^{n} Sim(\tilde{\tau}_i, \tilde{\tau}_j) \right) \right)
$$

We illustrate this definition calculating ranks for various terms that represent various parenthesization of the following stream of tokens from Equation 1:

$$\text{cosec}\, A - \sin A \cdot \sec A - \cos A \cdot \tan A + \cot A$$

There are several terms that can constructed from grammar G for this sequence of tokens. The highest ranked terms are shown below with their respective ranks.

$(\text{cosec}\, A - \sin A) \cdot (\sec A - \cos A) \cdot (\tan A + \cot A)$	49
$\text{cosec}\, A - \sin A \cdot \sec A - \cos A \cdot \tan A + \cot A$	36
$(\text{cosec}\, A - \sin A) \cdot (\sec A - \cos A) \cdot \tan A + \cot A$	33
$(\text{cosec}\, A - \sin A) \cdot \sec A - \cos A \cdot (\tan A + \cot A)$	31
$\text{cosec}\, A - \sin A \cdot (\sec A - \cos A) \cdot (\tan A + \cot A)$	30

We note that the more symmetric and balanced a term is, the higher its rank.

Filtering scheme. We find that certain terms such as $1/1$ or $\tau/1$ are unlikely to occur in mathematics texts. We formally define these as unlikely and filter out candidate terms containing unlikely terms as subterms.

Definition 4 (Unlikely terms)**.** *Let τ be a term in a grammar G'. We call τ **unlikely**, if it contains one of the following terms as a subterm (here $\tilde{\tau}$ is an arbitrary subterm): $\tilde{\tau} \,/\, 1, 1 \cdot \tilde{\tau}, \tilde{\tau} \cdot 1$, $0 \pm \tilde{\tau}$, $\tilde{\tau} \pm 0$, $\texttt{const} \pm \texttt{const}$, $\texttt{const} \cdot \texttt{const}$, $\tilde{\tau} \pm \tilde{\tau}$, $\tilde{\tau} \cdot \tilde{\tau}$, $\tilde{\tau} \,/\, \tilde{\tau}$, $\sqrt{\tilde{\tau}^{2k}}$, $\left(\sqrt{\tilde{\tau}}\right)^2$, $\sqrt{s}$, where s is of the form $\texttt{const}^2$.*

Algorithm. Given a sequence of tokens σ, we would like to enumerate all terms τ that can be generated from the grammar G, such that the sequence of tokens in τ is the same as σ, calculate the rank of each such term, and suggest the top few terms with highest ranks as possible alternatives to the user. However, this approach does not scale for larger sequences of tokens, since the number of possible paranthesizations explode exponentially with the length of the sequence.

We observe that we can perform an optimization by utilizing the hierarchical structure inherent in the grammar G. We create a dependency graph between nonterminals in any grammar, by introducing an edge from nonterminal U to nonterminal V if there is some production in the grammar with U on the LHS and V on the RHS. Then, we can calculate the strongly connected components of this dependency graph, and define a partial order $\preceq$ between nonterminals as follows: for two nonterminals U and V, define $U \preceq V$ iff the strongly connected component containing U is ordered before the strongly connected component containing V. Specifically, we get the following total order for the nonterminals in G: $Eqn \preceq Expr \preceq Poly \preceq Trig \preceq Arg$

Our optimized algorithm follows this partial order, and first processes the input sequence σ to find candidate terms with highest rank that can be generated by the highest nonterminal in this partial order (which is Arg in our grammar G). Then, it replaces the subsequence corresponding to such terms with the nonterminal Arg and proceeds to find candidate terms with highest rank that can be generated by the next highest nonterminal (which is $Trig$ in our grammar G). Proceeding in this layered fashion greatly reduces the number of paranthesizations we need to search and improves the speed of the algorithm. As we show later, our implementation is able to process most expressions from sample mathematics texts in a few seconds, and the terms with the highest rank is usually the one with the desired paranthesization.

Sequence Prediction Algorithm

We reduce the sequence prediction problem to learning a term transformer from a set of input-output examples, and present an algorithm for the latter problem.

Before we present the reduction, we note that sequences arise in many syntactic contexts in mathematics texts. For instance, in Equation 2 both the LHS and RHS of the equation correspond to the sequence $\langle \tan A, \tan 2A, \tan 3A \rangle$ with different separators ($\cdot$ in the LHS and $+$ in the RHS). In Equation 3 and Equation 4, we identify that each row of the matrix as a sequence. We perform pre-processing on the input stream to identify such sequences, and the remainder of this section operates on such sequences ignoring where they came from, be it from commutative and associative operators such as $\cdot$ and $+$, or from rows of matrices, etc.

From sequence prediction to term transformation. We detail two sequence prediction scenarios that occur commonly in mathematics texts, and show that both these scenarios can be reduced to term transformation.

Scenario 1 (Sequence generation)**.** *The user wants to enter a sequence σ of terms $\tau_1, \ldots, \tau_n$, such that:*

$$\tau_i = F(\tau_{i-1}), \quad i > 1$$

Given τ_1 and τ_2, we desire that our algorithm learns the transformer F and suggest the remaining terms of the sequence σ automatically.

For instance, the LHS of the equation 2, which is the sequence $\sigma = \langle \tan A, \tan 2A, \tan 3A \rangle$, has the term transformer "Increase argument coefficient by 1". The term transformer corresponding to the each row of the symbolic matrix in Equation 3 is "Increase exponent by 1", and the transformer corresponding to rows of the symbolic matrix in Equation 4 is "Change variables according to the pattern $x \to y \to z \to x$".

Scenario 2 (Sequence transformation)**.** *The user has input a sequence $\tilde{\sigma} = \langle \tilde{\tau}_1, \ldots, \tilde{\tau}_n \rangle$. Next, she inputs a new sequence $\sigma = \langle \tau_1, \ldots, \tau_n \rangle$. Each of the terms of the new sequence σ is a transformation of the corresponding term of the previous sequence $\tilde{\sigma}$:*

$$\tau_i = F(\tilde{\tau}_i), \quad i \geq 1$$

Given the first sequence $\tilde{\sigma}$ and the first term from the second sequence τ_1, we desire that our algorithm learns the transformer F and suggest the remaining terms of the second sequence σ automatically.

$$\begin{aligned} VarExpr &:= Sub(VarExpr, ConstExpr) \\ &\quad \mid Loop_{g,B} \mid Cycle_{g,B} \mid CVar(\texttt{var}) \\ ConstExpr &:= Linear_{A,B} \mid CConst(\texttt{const}) \end{aligned}$$

Figure 2: Pattern grammar G_P.

The RHS of the equation 2 can be produced from the LHS of the same equation using the identity transformation (since sequence separators are dealt with separately, and are not part of the sequence in our system). In the symbolic matrices in Equations 3 and 4, the first row can be transformed to produce each of the subsequent rows. The transformation for Equation 3 is "Increase subscript index if possible", and the transformation for Equation 4 is "Change variables according to the pattern $x \to y \to z \to x$". It is a coincidence that in Equation 4 the term transformer for the first row (corresponding to sequence generation) and the term transformer for the second row (corresponding to sequence transformation) are the same.

In both these scenarios, the sequence prediction problem can be reduced to the learning term transformers from input-output examples. Though all the above instances require only one input-output pair, in general there could be multiple term transformers that map an input term to an output term, and we may have to find a transformer that satisfies several input-output pairs. Also, it is convenient to provide the term index in the sequence as an input to the term transformer. Thus, we arrive at the following formulation:

Problem 3. *Given a set of triples of the form $\langle \tilde{\tau}_1, i_1, \tau_1 \rangle, \langle \tilde{\tau}_2, i_2, \tau_2 \rangle, \ldots$, where $\tilde{\tau}_j$ and τ_j are terms generated by grammar G, i_j are indexes of terms in the sequence, learn the most general transformation $T\colon \langle \tilde{\tau}, i \rangle \mapsto \tau$ that produces the third component of each triple from its first two components.*

In the sequence generation scenario, the input-output pairs are consecutive pair of elements of the sequence. In the sequence transformation scenario the input-output pairs are the corresponding elements of the context sequence and the new sequence.

Transformation language. Next, we define a term transformation language, where each transformation is expressed as a set of rewrite rules. Each rewrite rule specifies an input pattern that should be matched in the input term $\tilde{\tau}$, and an output pattern to transform the term that matches with the input pattern. These patterns depend on free variables, element index i in a sequence, and predefined set of commonly used global sequences (such as "A, B, C", "x, y, z", "α, β, γ"). We extend our grammar G with such patterns and produce the grammar G_P shown in Figure 2. G_p contains two new nonterminals $VarExpr$ and $ConstExpr$ which replace $\texttt{var}$ and $Const$ in grammar G. The remaining new nonterminals in G_p are described below: (1) $Sub(VarExpr, ConstExpr)$ is a subscripted variable.

That is, a variable subscripted with a constant. (2) $Loop_{g,C}$ is a $(i + C)$-th element of a global sequence g, where i is the index of current element in the current sequence. (3) $Cycle_{g,C}$ is a $((i + C) \bmod n)$-th element of a global sequence g of total length n, where i is the index of current element in the current sequence. (4) $Linear_{A,B}$ is a constant equal to $A \cdot i + B$, where i is the index of current element in the current sequence. (5) $CVar(\texttt{var})$ and $CConst(\texttt{const})$ are constant expressions with a certain variable and constant, respectively.

Definition 5. *A **rewrite rule** R is a pair*

$$\vartheta_1(e_1, \ldots, e_k) \rightarrow \vartheta_2(e_1, \ldots, e_k)$$

where $\vartheta_1(e_1, \ldots, e_k), \vartheta_2(e_1, \ldots, e_k)$ are terms in grammar G_P, extended with free variables $e_1, \ldots, e_k$.

A term transformer T is a set rewrite rules

$$\{\vartheta_{1,j}(e_1, \ldots, e_k) \rightarrow \vartheta_{2,j}(e_1, \ldots, e_k)\}_{j=1}^{l}$$

When applied on the input pair $\langle \tau, i \rangle$, the transformer T operates as follows: wherever the subterm of an input term τ matches the input pattern $\vartheta_{1,j}$ of a rule R_j, replace it with the output $\vartheta_{2,j}$, substituting actual values of free variables $e_1, \ldots, e_k$ correspondingly. If $\vartheta_{1,j}$ and $\vartheta_{2,j}$ contain terminals that are dependent upon the element index, the actual index i is substituted from the input to T.

A transformation T may contain conflicting rules, meaning that on some particular argument ϑ two rules $R_1 \in T$ and $R_2 \in T$ may trigger a replacement in intersecting positions. In this case we give priority to the rule with higher index (arbitrarily and deterministically).

Consider the symbolic matrix from Equation 4. The first row of the matrix can be described using the following pattern:

$$\vartheta_1 = Cycle_{g,1} * Cycle_{g,2} - (Cycle_{g,0})^2$$

where g is the global sequence "x, y, z". The second row can be produced from the first with the following transformation:

$$T = \{Cycle_{g,k} \rightarrow Cycle_{g,k+1} \mid k = 0, 1, 2\}$$

This produces the new pattern for the second row:

$$\vartheta_2 = Cycle_{g,2} * Cycle_{g,3} - (Cycle_{g,1})^2$$

The third row can be similarly produced from the second.

Consider the Vandermonde matrix from Equation 3. Each row of the matrix can be can be obtained from the previous row with the following term transformer:

$$\{Sub(CVar(\text{“x”}), CConst(j-1)) \rightarrow Sub(CVar(\text{“x”}), CConst(j))\}$$

Learning Algorithm. Our learning algorithm is based on the inductive synthesis methodology proposed recently for doing string transformations (Gulwani 2011). It makes use of two key procedures: (i) `GenerateTrans`, which generates the set of all transformations T that are *consistent* with a given input-output example (i.e., transformation T maps the example input into the example output). (ii) `Intersect`, which intersects the sets of transformations, where each set corresponds to a specific example. The number of transformations that are consistent with a given input-output example can be huge and requires efficient data-structures for succinct representation and manipulation of such sets. Next, we describe the specifics for our case of term transformations.

Data-structure. Instead of explicitly representing the set of all transformations that are consistent with a given example, we represent such a set succinctly by a single saturated set S of rewrite rules each of which is compatible with the example. We say that a rewrite rule R is *compatible* with an example if application of R to the example input makes modifications that are compatible with the example output. Such a set S represents any transformation that comprises of any subset S' of S such that S' covers the example. We say that a set of compatible rewrite rules *covers* an example if application of the rewrite rules transforms the example input into the example output.

Procedure `GenerateTrans`. It compares the structure of the ASTs of the input term $\tilde{\tau}_1$ and the output term τ_1, and infers the set of all candidate rewrite rules that can transform parts of $\tilde{\tau}$'s AST into corresponding parts of τ's AST. At the non-terminal level, for each unmatched node we find a matching set of free variables within the distance of k tree levels from the unmatched position (A choice of k=2 turns out to be efficient and sufficient in practice). The unmatched part forms the essential replace nodes. We track the position π_j of the AST, from which each particular rewrite rule R_j was derived. At the terminal level (for the first element of the sequence, or for sequence generation scenario) we can use a set of heuristics that describe the common transformations that take place in real life examples: change the global sequence g to g', make a shift by one in global sequence g, increase or change a constant, change a variable, and so on. Each of the candidate rewrite rule that is compatible with the example is put into the solution set.

Procedure `Intersect`. It intersects the set of all saturated sets of rewrite rules for each example, followed by filtering all those rules that are incompatible with any example. A final check is performed to see whether the resultant set covers each of the examples. If so, the resultant set of rewrite rules is returned as a transformation that is consistent with each of the examples. Otherwise, the algorithm signals failure.

Evaluation

We have implemented both the structure prediction and sequence prediction algorithms in C#. In this section, we evaluate our implementation of these algorithms on on benchmark examples collected from high school text books (Loney; Khanna). All experiments are performed on Intel Core 2 Quad 2.5 Ghz machine.

Structure prediction. We collected 85 benchmarks from the above text books to evaluate structure prediction. We first process each benchmark with a baseline algorithm which uses default parsing order called PEMDAS (Parentheses, Exponents, Multiplication, Division, Addition, Subtraction), with left-associativity, and associating each function with the first monomial to the right of a function name as its argument. We define **usefulness measure** for a benchmark as number of parenthesis in a term which are not implied by the baseline algorithm. This is the room available for any algorithm (not just ours) to improve.

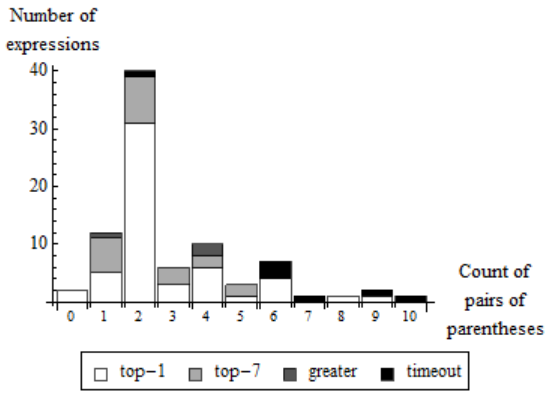

Figure 3: Structure prediction results.

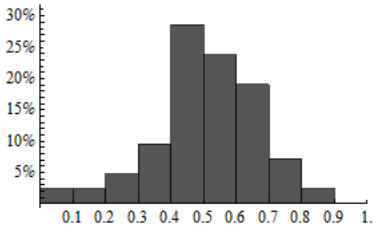

Figure 4: Sequence prediction effectiveness.

Next we evaluate how many of these non-trivial parentheses are correctly predicted by our structure prediction algorithm. These results are shown in figure 3, which is a histogram showing how often the desired term was ranked within top-1, how often it was ranked within top-7, greater than top-7, or failed because of 10 sec timeout, specifically for each value of usefulness measure. For most examples, the correct result is found in less than 0.1 sec, and the correct result is ranked close to the top.

Sequence prediction. We collected 45 benchmark sessions (arbitrarily chosen) from textbooks for this evaluation, which contained 186 general sequences. For each of our benchmarks, we first measure the usefulness of our pattern language: how many transformers that need to be inferred for correctly predicting the completion of the sequence can be expressed by our pattern language G_P. Of the 186 benchmarks, we find that for 114 benchmarks, the correct transformer can be expressed using the pattern language G_P. Thus, the usefulness of our pattern language is 61.29%.

For each of these 114 benchmarks, we measure the ratio of the number of keystrokes saved by user thanks to our algorithm to the total number of keystrokes in the sequence. The latter is the **effectiveness** of our algorithm. Our implementation showed top 5 predictions, and we considered our algorithm to be successful it showed the correct completion as one of the top 5 predictions. Figure 4 shows the results of this evaluation, as a histogram with effectiveness on the x-axis (with bin width 0.1) and percentage of benchmarks on the y-axis. Overall 52% of benchmarks save 50% or more of typing time. All these benchmarks are processed by our implementation in less than 1 second.

Related Work

There has been work on inferring structure from unstructured data in the context of several domains including databases (Buneman et al. 1996), web data (Mendelzon, Mihaila, and Milo 1996), and programming (Little and Miller 2009). We develop a novel structure prediction algorithm for a novel domain, namely parenthesis insertion in mathematical text. Our approach is based on the hypothesis that natural mathematical structures have low entropy with respect to a natural definition of symmetry unlike past work that leverage statistical inference.

Real-time auto-completion has been a popular research topic in several domains including web query expansion (Efthimiadis 1993; Paek, Lee, and Thiesson 2009), mobile text entry (Mackenzie and Soukoreff 2002), and entry of programming text. We develop a novel algorithm for real-time auto-completion for a novel domain, namely term prediction in mathematical text. Our algorithm is based on the hypothesis that there is a lot of repetitiveness in mathematical text and hence can be predicted from simply the prefix and immediately surrounding context. In contrast, past techniques for free-form text leverage statistical inference.

Our algorithm for term prediction is based on inductive synthesis (i.e., synthesis by examples (Lieberman 2001; Cypher 1993)) of term transformations. We use a similar methodology as proposed recently by Gulwani (2011) for learning spreadsheet string transformations, wherein the set of all consistent transformations is computed separately for each example, followed by an intersection of such sets. However, the specifics of the underlying domain specific language for transformations, the two key procedures for generating a set of consistent hypothesis and for intersecting such sets, the data-structure for succinct representation of such sets, are all different and novel.

Conclusion and Future Work

With increasing educational cost and classroom sizes, there is an increasing need to automate repetitive tasks such as solution/hint generation, problem generation, and automated feedback. Recent technological advances in the form of better algorithmic techniques for logical reasoning, online social educational sites such as KhanAcademy, and availability of new multi-modal devices with variety of form factors have created a favorable setting. However, challenges in entering structured data can be a stumbling bottle-neck to realize this vision. This paper provides the core building blocks for making it easy to enter structured mathematical content. Our techniques rely on the observation that natural mathematical text is symmetric and repetitive, and allows us to borrow methodologies for automating program synthesis (Gulwani 2010) for repetitive tasks (Gulwani 2011; Harris and Gulwani 2011). The next step is to put our content creation system online in conjunction with some other components that we are building (namely solution checking, solution generation, and problem generation) to create a usable intelligent tutoring system for algebra.

References

Buneman, P.; Davidson, S.; Hillebrand, G.; and Suciu, D. 1996. A query language and optimization techniques for unstructured data. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, volume 25, 505–516. ACM Press.

Cypher, A., ed. 1993. *Watch What I Do – Programming by Demonstration*. MIT Press.

Efthimiadis, E. N. 1993. A user-centered evaluation of ranking algorithims for interactive query expansion. In *SIGIR*, 146–159.

Gulwani, S. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *38th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Revised version to appear in CACM Research HighLights.

Harris, W. R., and Gulwani, S. 2011. Spreadsheet table transformations from examples. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Revised version to appear in CACM Research HighLights.

Khanna, M. L. *IIT Mathematics*.

Lieberman, H. 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann.

Little, G., and Miller, R. C. 2009. Keyword programming in java. *Automated Software Engineering* 16(1):37–71.

Loney, S. L. *Plane Trigonometry*. Cambridge University Press.

Mackenzie, S. I., and Soukoreff, W. R. 2002. Text Entry for Mobile Computing: Models and Methods,Theory and Practice. *Human-Computer Interaction* 17(2 & 3):147–198.

MathWorks. Commonly encountered error and warning messages.

Mendelzon, A. O.; Mihaila, G. A.; and Milo, T. 1996. Querying the world wide web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS)*, 80–91.

Paek, T.; Lee, B.; and Thiesson, B. 2009. Designing phrase builder: a mobile real-time query expansion interface. In *Mobile HCI*.