Template-based Formula Editing in Kaava

Ken Rimey*
rimey@cs.hut.fi
Department of Computer Science
Helsinki University of Technology

Abstract

This paper describes a user interface for entering mathematical formulas directly in two-dimensional notation. This interface is part of a small, experimental computer algebra system called Kaava. It demonstrates a data-driven structure editing style that partially accommodates the user's view of formulas as two-dimensional arrangements of symbols on the page.

1 Introduction

Before someone with a mathematical problem to solve can benefit from having a workstation on his desk, he has to enter the relevant formulas onto the screen. All the major computer algebra systems have expected him to use a Fortran-like expression syntax for this. Although we have grown accustomed to using this syntax for the kinds of expressions we put into Fortran programs, the richer, more concise, and easier-to-read notation that a scientist or applied mathematician uses in problem solving does not take the form of strings of characters. Although the major computer algebra systems use character strings for input, all support two-dimensional mathematical notation for output. It would be better to use the same language for both input and output.

This has been done, both in small algebra systems and in add-on interfaces to major algebra systems. Neil Soiffer's PhD thesis surveys these efforts and provides a comprehensive discussion of strategies for directly entering two-dimensional notation [8]. The key issue seems to be a conflict between the hierarchical structure of mathematical expressions and our tendency to read and write from left to right.

Consider the expression

$$ax^2 + bx + c$$

In reading it aloud, or in writing it, most people would begin with the a. Even the purest mathematician is aware of the left-to-right arrangement of the symbols on the page. The natural basis for representing the expression in a computer, on the other hand, is its hierarchical structure as a sum of three terms, two of which are products, and so on. Designing an editor that represents expressions as trees and yet allows the user his visual bias in entering them is a hard problem.

Soiffer favors retaining an underlying string model as the basis for the editing operations:

```
a * x ^ 2 + b * x + c
```

*Author's current address: Unda Oy, Ahventie 4A, SF-02170 Espoo, Finland.

This string reflects the left-to-right layout of the displayed expression and is second nature for a programmer to type. Soiffer discusses two alternative ways of keeping the expression tree up to date as the user, in effect, edits a string: incremental parsing and the simulation of parsing through tree manipulations. The latter was first considered by Kaiser and Kant [4].

This paper focuses on the contrasting approach that completely abandons Fortran-like expression languages. The small, Macintosh-based algebra system, *Milo*, has demonstrated how this can be done without the clumsiness generally associated with structure editing [5]. One of the key ideas is allowing the selection of an insertion point. This seemingly minor user interface issue is trickier and more important than it may at first seem. Milo has recently been enhanced and incorporated into the *FrameMaker* desk-top publishing system [2, 3].

The experimental small algebra system, *Kaava*,[1] takes a similar approach. In designing its formula editor, I have endeavored to create a data-driven mechanism that can be exposed to the user. The remainder of this paper describes the result. After some comments on the representation of the expression trees and on the user extensibility, I describe the editing strategy in three stages with progressively fewer restrictions on what can be selected: placeholders only, arbitrary expressions, and finally arbitrary expressions and insertion points.

Kaava is written in Common Lisp and runs under the X window system. Harri Pasanen provides an overview of the system at an early stage of development in his master's thesis [6].

2 Representation

Kaava represents each formula displayed on the screen as an expression tree. In fact, it represents the user's entire working document (containing formulas and paragraphs of explanatory text) as one big tree. Each node of the tree contains the following information:

- An operator. Leaf nodes have operators like *number* and *variable*.
- A list of child nodes, i.e. arguments.
- Additional information, depending on the operator (number's value, variable's name, etc.).

Although the objects that represent these abstract nodes in the program have other information attached to them, including pointers to a data structure describing the layout of the symbols on the screen, this is unimportant in understanding the program's behavior. The user edits an expression tree.

The task of choosing a vocabulary of operators is complicated by the conflicting needs of editing and algebraic manipulation. Instead of compromising, Kaava uses different representations for the two purposes, converting expressions freely back and forth. The editing representation is the primary one, and the only one of concern in this paper. The other is used in performing mathematical commands.

Whereas a small operator vocabulary is most conducive to algebraic manipulation, Kaava's editing representation is designed to correspond closely to what the user sees on the screen. For example, rather than abandon the division operator in favor of negative exponents, Kaava instead represents the expression

$$\frac{xy}{2ab}$$

straightforwardly as a quotient of two products. Rather than abandon unary minus in favor of multiplication by -1, Kaava uses the unary minus operator wherever a minus sign appears in an

[1] *Kaava* rhymes with *java* and is the Finnish for *formula*.

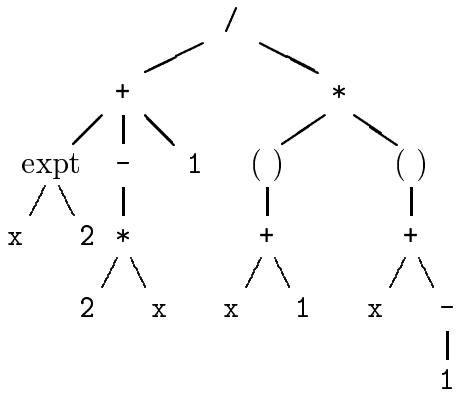

Figure 1: An expression tree.

expression, as in $-x$, $-2x$, and even -3. A subtraction operator, on the other hand, would be undesirable because of the unwanted structure it would impose on an expression such as the following:

$$a + b - c - d$$

Kaava represents this as a sum of four terms, and by the way, has no preference as to their order.

Figure 1 illustrates these conventions using the tree for the expression

$$\frac{x^2 - 2x + 1}{(x + 1)(x - 1)}$$

It also illustrates our use of an explicit parenthesizing operator. After each editing operation, Kaava inserts instances of this operator wherever it is needed to make the expression tree legal. It also *flattens* nested instances of certain operators.

Legality is determined by a *structural schema*. This defines how many arguments an operator can have and what the operators of these arguments can be.[2] For example, a child of a product node may not be a + node, but it may be a parenthesizing node.

We use a combination of mechanisms to specify the schema. First, we assign each operator to one or more operator classes and specify the classes to which its arguments may belong. Second, we give each operator a precedence value and specify minimum precedence values for its children. The operator-class mechanism is useful for enforcing basic sanity rules. For instance, it is important to insure that the argument of a product is never a paragraph of text. If, as in this case, flattening and adding parentheses does not make the final tree legal, the program refuses to perform the editing operation.

The purpose of flattening is to facilitate the entry of operators that take arbitrary numbers of arguments. Without it, substituting $c + d$ for the c in

$$a + b + c$$

would produce

$$a + b + (c + d)$$

The actual result is

$$a + b + c + d$$

[2]The general idea of a structural schema appears to be well-known in the structure editing community. The variant used in Kaava is relatively weak insofar as it does not have the power of a context-free tree grammar.

Kaava flattens not just associative mathematical operators, but certain nonmathematical operators as well. For example, there is a *comma* operator that takes two or more arguments and appears, for instance, in the representation of

$$f(a, b, c)$$

A command to replace the c by "c, d" results in the expression

$$f(a, b, c, d)$$

3 Pasting

In a sense, there is only one important command in Kaava: *pasting*. It can be invoked either with the mouse or from the keyboard. The pasted expression determines the semantics of the operation.

In the simplest case, pasting replaces the *selection* by the expression being pasted. Consider again the example of the sum $a + b + c$. Selecting the c

$$a + b + \boxed{c}$$

and pasting $c + d$ results, after flattening, in the expression

$$a + b + c + d$$

The initial selection can be done with the left mouse button. The pasting can then be done by selecting any $c + d$ in the document with the right mouse button. The left button controls the *primary selection* (the destination), while the right button controls the *copy selection* (the source). Only the primary selection is persistent; the copy selection disappears when the mouse button is released.

We call pasting done with the mouse in this way *copy-pasting*. It combines two operations from the standard cut-copy-paste trio, eliminating the cut buffer. Although the user interface is spartan, the effect of copy-paste is the same as that of an intra-application drag-and-drop.

The user can paste an expression from the keyboard by hitting a key bound to the expression. Keys are bound to expressions by including *keybindings* in the document. For example, the key Control-A can be bound to the expression $c + d$ by putting the keybinding

$$\texttt{C} - \texttt{A} : c + d$$

anywhere in the document. Then the pasting operation of the previous example can be performed by hitting that key. The default initial document predefines many keybindings, including, for instance, bindings of the alphabetic keys to their corresponding symbols:

$$\texttt{a} : a \quad \texttt{b} : b \quad \texttt{c} : c \quad \ldots$$

The simple replacement semantics generalize to the semantics of pasting a *template*. The initial document includes, for example, the following:

$$+ : .?+?. \qquad * : .??. \qquad / : \frac{.?}{?.}$$

Template pasting semantics is the topic of sections 4–6.

Unfortunately, the behavior that users expect of the "-" key is too complicated to be specified in this way. Sometimes it should paste one template ($-?.$), and sometimes another ($.?-?.$). It actually invokes a built-in command, represented in the document by its quoted name:

$$- : \text{“minus”}$$

Sometimes the effect of pasting a built-in command may have nothing to do with pasting templates:

$$\texttt{C} - \texttt{Q} : \text{“quit”}$$

Kaava's as-yet limited mathematical facilities are invoked through the pasting of rewrite rules onto expressions to be transformed. Here it makes sense to have many bindings for a single key, as in the expand command:

$$\texttt{C} - \texttt{X} : \boldsymbol{a}(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{a}\boldsymbol{x} + \boldsymbol{a}\boldsymbol{y}$$
$$\texttt{C} - \texttt{X} : (\boldsymbol{x} + \boldsymbol{y})^{\boldsymbol{n}} = \text{expand-expt}\left((\boldsymbol{x} + \boldsymbol{y})^{\boldsymbol{n}}\right)$$

Most everything that can be done with keybindings can be done as well with copy-pasting. Collecting the standard keybindings neatly at the head of the document produces a kind of palette from which the user can copy-paste as an alternative to the keyboard.

The special behavior of rules and built-in commands makes them tricky to manipulate. Binding the middle mouse button to a *literal copy-paste* operation with only simple replacement semantics solves this problem.

4 Placeholders

Placeholders enable the representation of incomplete expressions. They appear in expressions that are in the process of being entered:

$$ax^2{+}?$$

They also appear in templates (expressions meant for pasting):

$$+ :?{+}?$$

Kaava displays ordinary placeholders as question marks.

If we select only placeholders, the pasting of a template is purely a matter of replacement:

$$ax^2 + \boxed{?} \xrightarrow{?+?} ax^2{+}?{+}?$$

Here the template is ?+?.

Arbitrarily complex expressions can be entered in this way. For example, the expression

$$ax + b$$

can be entered as follows:

$$\boxed{?} \xrightarrow{?+?} \boxed{?}{+}? \xrightarrow{??} \boxed{?}?{+}? \xrightarrow{a} a\boxed{?}{+}? \xrightarrow{x} ax + \boxed{?} \xrightarrow{b} ax + b$$

With appropriate keybindings, what the user types is

```
+ * a x b
```

Unfortunately, successive placeholder replacement amounts to using prefix syntax for input.

Another problem is that the user apparently needs to select a placeholder before each keystroke. In practice, having to switch back and forth between the mouse and keyboard in entering an expression is unacceptable; the system must always leave something selected after each operation. Kaava does this, but it does not always select a placeholder. Whenever there are placeholders in the replacement expression, however, it automatically selects one of them.

5 Selections

The opportunity to go beyond replacement semantics makes the selection of arbitrary expressions attractive. The user establishes a selection by depressing the left mouse button, moving the mouse, and then releasing the button. The selected expression is the smallest one whose bounding box includes both endpoints.[3]

The user can adjust the selection with various built-in commands. With the standard keybindings, the four arrow keys move the selection up, down, left, and right in the expression tree: Up-arrow selects the parent; Down-arrow, the first child; Left-arrow and Right-arrow, a sibling. The space bar expands the selection like Up-arrow. The tab key moves it to a nearby placeholder. (These keybinding conventions come from Milo.)

When a template containing at least one placeholder is pasted onto a target expression, the replacement expression is constructed by incorporating the target into the template in place of one of the placeholders. The authors of CaminoReal call this *wrapping* [1]. Soiffer would call the template an *overlay* [8]. We have so far been satisfied with the choice of the first placeholder in the template as the one to replace, but it would be an improvement to have a distinguished type of placeholder that could be incorporated into the template to control the choice.

The template pasting procedure thus far is as follows:

1. If there are placeholders in the template, choose one to be replaced.
2. If there are yet other placeholders in the template, choose one to be selected.
3. Replace the chosen placeholder, if any, with the target expression.
4. Substitute the resulting replacement expression for the target expression in the document.
5. Select the chosen placeholder, if any.

There remains the question of what to do in the last step if no placeholder was chosen to be selected. Remember, the critical issue is that *something* must be selected. One natural choice is as follows:

5′. Select the chosen placeholder. If none was chosen to be selected, select the replacement expression.

This allows the user to enter, for example, the expression

$$ax + b$$

entirely from the keyboard as follows:

$$\boxed{?} \xrightarrow{\texttt{a}} \boxed{a} \xrightarrow{\texttt{*}} a\boxed{?} \xrightarrow{\texttt{x}} a\boxed{x} \xrightarrow{space} \boxed{ax} \xrightarrow{\texttt{+}} ax + \boxed{?} \xrightarrow{\texttt{b}} ax + \boxed{b}$$

The required sequence of keystrokes

`a * x` space `+ b`

bears at least some resemblance to the expression being entered. Only the `*` and the space seem superfluous.

The version of rule 5 that we actually use is the following:

[3]Using a simple bounding box would not be appropriate if Kaava broke long expressions onto multiple lines.

5″. Select the chosen placeholder. If none was chosen to be selected, select the insertion point to the right of the replacement expression.

When combined with the next section's pasting procedure for insertion points, this yields extremely natural entry of linear notations for unary and binary operators. Nonlinear notations, such as exponents and fractions, are only slightly more troublesome to enter. On the other hand, operators that require more than two arguments unavoidably force a top-down style of entry. For example, although the user can wrap a definite-integral template around a previously entered integrand,

$$\boxed{x^2} \longrightarrow \int_{\boxed{?}}^{\boxed{?}} x^2 d\boxed{?}$$

he will then have to select and replace the remaining placeholders in turn.

6 Insertion points

Every insertion point is either the left- or the right-hand side of an expression. For example, the expression $ax + b$ has five insertion points:

$$|a|\ x|\ +\ |b|$$

The leftmost insertion point is the left-hand side of three different expressions: a, ax, and $ax+b$. The next is the right side of a and the left side of x. The insertion points divide the displayed expression roughly as if it were a linear string of symbols.

It is not always the case that an insertion point neighboring an expression is identified with an insertion point neighboring the expression's first or last child. This can be because the two insertion points are at different horizontal positions on the screen, or because the expression's main operator arranges its arguments in a nonlinear way. Both cases are illustrated by the following:

$$|f|\ (|x|)| - \left|\frac{|1|}{|2|\ x|}\right|$$

The right side of $f(x)$ and the right side of its second argument, x, correspond to distinct insertion points because they are at different positions. The fraction does not at all resemble a string of symbols, so it is treated as if it were itself a single symbol. The overall expression thus forms a string of six symbols:

$$|\ |\ |\ |\ |\ \Big|\ \ \Big|$$

The numerator and denominator of the fraction form symbol strings of their own.

The user selects an insertion point by positioning the mouse and then clicking the left button. The system uses bounding boxes to determine the relevant symbol string:

$$\boxed{f(x) - \frac{\boxed{1}}{\boxed{2x}}}$$

It then scans the sequence of insertion points for that string to find the one nearest the click position. The user can step the selection back and forth in the sequence with the left and right arrow keys.

All of this creates the illusion of linear expression structure for the user, but this structure is a fiction. The system must relate insertion points to the expression tree.

Internally, Kaava represents an insertion point as a combination of a path in the expression tree (a sequence of child indices starting from the root) and an indication of *left* or *right*. We have chosen to make the representation unique by preferring *right* over *left*, and a smaller expression over any expression containing it.

Externally, an insertion point identifies a node of the expression tree only ambiguously. This ambiguity makes insertion points quite awkward from a programming perspective. The same ambiguity is, in my opinion, the main justification for allowing the user to select insertion points.

Consider the insertion point to the right of the product xy:

$$x\ y|$$

If the user types a `+`, he will expect xy to become the left-hand argument:

$$x\ y| \xrightarrow{+} xy + \boxed{?}$$

If he types a `^` to indicate exponentiation, he will expect y to become the base:

$$x\ y| \xrightarrow{\hat{}} xy^{\boxed{?}}$$

Both cases involve wrapping a binary template around an expression, but the expressions is xy in the first case and y in the second. In each case, the choice happens to be the only one that directly produces a tree satisfying the structural schema, without the addition of parentheses.

So that the user can conveniently take control of the choice, hitting the space bar expands the selection from an insertion point to the smallest neighboring expression, preferably one on the left.

$$x\ y| \xrightarrow{space} x\boxed{y} \xrightarrow{+} x(y + \boxed{?})$$

One can hit it again to select the next larger expression.

$$x\ y| \xrightarrow{space} x\boxed{y} \xrightarrow{space} \boxed{xy} \xrightarrow{\hat{}} (xy)^{\boxed{?}}$$

Whether a template should be wrapped around anything at all when it is pasted at an insertion point is a matter of how the template appears to the user. Kaava leaves the decision up to the template designer. It replaces a placeholder with an expression to the left of the insertion point only if he has made the placeholder *left-sticky*. It replaces a placeholder with an expression to the right only if the placeholder is *right-sticky*. Stickyness is shown as a period on the left or right side of the question mark, as in the standard `+` template:

$$.?+?.$$

It is natural to make the base of an exponential left-sticky, but how to treat the exponent is a matter of taste. Both the base and the exponent are sticky in our standard exponential template:

$$.?^{?.}$$

In contrast, an integral template might include no sticky placeholders at all:

$$\int ?d?$$

Making the integrand right-sticky becomes an option, though, if we bind the template to a key that we expect the user to associate with the integral sign:

$$\texttt{\$} : \int ?.\ d?$$

When the template does not include an appropriately sticky placeholder, the system will have to actually insert it at the insertion point. Kaava does this with the aid of ordinary multiplication. It multiplies the original template by a sticky placeholder on either the left or the right to produce the effective template. In effect, it transforms the template x into one of the following:

$$.?x \qquad x?.$$

Putting all of these ideas together into a pasting procedure is not easy. One of the compromises in Kaava is that the decision as to whether to wrap the template around an expression to the left or around an expression to the right (though not the choice of the particular expression) is made at the outset without considering the form of the template. If there is any neighboring expression at all on the left, Kaava chooses the left side. Here is the procedure for pasting at an insertion point:

1. If there is a placeholder that is sticky on the given side, mark it for replacement.
2. If there are yet other placeholders, choose one to be selected.
3. If no placeholder was marked for replacement, multiply the template on the given side by a placeholder marked for replacement.
4. Let the *target expression* be the smallest neighboring expression on the given side.
5. If the target expression is not the largest of the neighboring expressions on the given side, and if wrapping the template around it would not directly produce a document tree satisfying the structural schema, make the target expression's parent the new target. Repeat as many times as necessary.
6. Replace the placeholder marked for replacement with the target expression.
7. Substitute the resulting replacement expression for the target expression in the document.
8. If a placeholder was chosen to be selected, select it. Otherwise, depending on whether the given side is **L)** the left side or **R)** the right side, select the insertion point

   L) to the right of the replacement expression, or

   R) to the left of the placeholder that was replaced.

A few examples will wrap things up. The expression $ax + b$ of the previous section can now be entered as follows:

$$\boxed{?} \xrightarrow{\texttt{a}} a| \xrightarrow{\texttt{x}} a\ x| \xrightarrow{\texttt{+}} ax + \boxed{?} \xrightarrow{\texttt{b}} ax + b|$$

Let us say we want to start with a copy of this and produce the following quadratic:

$$ax^2 + bx + c$$

We select the insertion point to the right of the x and enter the exponent:

$$a\ x| + b \xrightarrow{\texttt{\^{}}} ax\boxed{?} + b \xrightarrow{\texttt{2}} ax^{2|} + b$$

Then we select the insertion point to the right of the b and enter the additional x and the third term:

$$ax^2 + b| \xrightarrow{\texttt{x}} ax^2 + b\ x| \xrightarrow{\texttt{+}} ax^2 + bx + \boxed{?} \xrightarrow{\texttt{c}} ax^2 + bx + c$$

We could also have entered the quadratic from scratch, in which case the sequence of keystrokes would have been as follows:

`a x ^ 2` space space space `+ b x + c`

7 Conclusion

Usable formula editing interfaces are surprisingly complex. Only by distilling the essential ideas from our design attempts can we save these interfaces from creeping featurism.

This paper recommends the use of a literal expression representation corresponding closely to what the user sees on the screen. It also recommends the uniform, data-driven treatment of editing operations as template pasting. My recommendation regarding insertion points would depend on the available programming resources. The ambiguity that makes them useful in the first place also makes them rather difficult to implement.

The idea of sticky placeholders is one that I have not seen elsewhere. Although the semantics of pasting at an insertion point needs further refinement, Kaava's template notation will be a good foundation for this work.

Finally, I would like to reiterate that structure editing as described in this paper has a competitor: incremental parsing. I hope that the paper is a step towards an eventual understanding of the tradeoffs between these two approaches.

Acknowledgments

This work was supported by the Academy of Finland. Kim Nyberg, Harri Pasanen, Kenneth Oksanen, Tero Mononen, and Harri Hakula assisted in developing Kaava as it now stands. I especially want to thank Harri Pasanen, whose thesis has significantly influenced the exposition in this paper.

References

[1] Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger. CaminoReal: An interactive mathematical notebook. Technical Report EDL-89-1, Xerox PARC, 1989.

[2] Ron Avitzur. Suggestions for a friendlier user interface. In *Proc. DISCO '90: Symposium on the Design and Implementation of Symbolic Computation Systems*, pages 282–283, 1990.

[3] Frame Technology Corporation, San Jose, California. *Using FrameMath*, 1989.

[4] Gail E. Kaiser and Elaine Kant. Incremental parsing without a parser. *Journal of Systems and Software*, 5:121–144, 1985.

[5] Paracomp, San Francisco, California. *Milo User's Guide*, 1988.

[6] Harri Pasanen. Highly interactive computer algebra. Master's thesis, Helsinki University of Technology, Department of Computer Science, 1992. Available as Technical Report TKO-C52.

[7] Carolyn Smith and Neil Soiffer. MathScribe: A user interface for computer algebra systems. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 7–12. Association for Computing Machinery, 1986.

[8] Neil Soiffer. *The Design of a User Interface for Computer Algebra Systems.* PhD thesis, University of California at Berkeley, 1991.