J. Symbolic Computation (1995) **11**, 1–000

The design of a customizable component to display and edit mathematical formulas

Olivier Arsac and Stéphane Dalmas and Marc Gaëtano †

INRIA Sophia Antipolis, projet SAFIR

2004 route des lucioles BP 93

06902 Sophia-Antipolis CEDEX

France

E-mail: olivier.arsac@sophia.inria.fr
stephane.dalmas@sophia.inria.fr
marc.gaetano@sophia.inria.fr

(Received 3 April 1995)

This paper describes the design and implementation of , a customizable component to display and edit mathematical formulas. can be easily embedded in other applications and uses a Lisp language as its extension language, in the same spirit as the text editor. To insure a good responsiveness, the editor is multithreaded and normally runs as separate process.

1. Introduction

This paper describes the design and implementation of *Emath*, a customizable formula editor that can be easily embedded in other applications. Much of the originality and power of *Emath* come from the fact that it uses a Lisp language as its extension language (in the same manner as the popular GNU Emacs text editor uses Emacs-Lisp). Customizing through a Lisp language is a natural choice because of the success of *Emacs* and because the symbolic computation community is quite familiar with Lisp anyway. To insure a good, guaranteed, responsiveness, the editor is multithreaded and normally runs as a separate process. *Emath* can be used on most workstations with the X window system and can handle all common mathematical notations, including matrices and commutative diagrams. *Emath* was not designed to enter and display whole mathematical texts but to be included in an application as a tool to manage the formulas. Nevertheless, it can also display texts and pixmaps (bitmaps with color information) together with formulas.

There is a clear need for a good equation editor: to be used in interfaces to symbolic computation systems, in WYSIWYG document preparation systems with mathematical facilities or in electronic or interactive books to display and modify formulas. Several

† also I3S, CNRS URA 1376,Université de Nice Sophia–Antipolis

0747–7171/90/000000 + 00 $03.00/0

equation editors have been built (see, for example, [YW87], [SS86], [vEHvV89]). Most (if not all of them) are deeply linked with a whole interface. Building such an editor from scratch for each application that needs equation editing or displaying capability is certainly a hard task. For example, the *FrameMaker* document creation system uses *Milo* (written by Ron Avitzur) as an embedded equation editor (that includes some mathematical facilities). As we have text widgets in X toolkit, it is natural to think of building a formula editor widget (in fact, for practical reasons, *Emath* is not yet a widget).

Our approach to build this component was to enforce a clear separation between edition and mathematical manipulations on one hand and interface issues and edition/manipulation on the other hand. *Emath* has as little mathematical knowledge as possible, while keeping the burden of interfacing with an application as low as possible. Our editor does not attempt to do clever things like (most) elisions and line cuttings. It relies on the application for nearly all mathematical decisions.

Emath is largely adaptable to the needs of the end user as well as those of the programmer of the application that uses it. Almost everything (input model and parsing, formatting, cut and paste behaviors...) can be changed or customized through Lisp code. We feel this is really necessary. We believe that we don't have yet a sufficient experience with interfaces dealing with the manipulation of formulas. Some time ago, we conducted a little study with people using computer algebra daily for applications that clearly shows they expect more than they can get now. We don't know yet what is better from a human interaction point of view. Thus, we need and want to experiment with different solutions. *Emath* should be a very good platform for experimenting (as we can drastically change the behavior of the editor dynamically, without any recompilation). The fact that this editor could be used in several kinds of application also asks for good customization possibilities.

2. The architecture of *Emath*

The problem of a user interface designed specifically for computer algebra systems has already received some attention ([Avi90], [YW87], [Soi91]). A good user interface is a substantial piece of software that requires significant efforts for its design and implementation. Thus, it stands to reason that the design of a user interface should rely on an existing and extensible equation editor that provides basic editing/displaying capabilities. The same remark applies to software for document formatting (such as *FrameMaker*). Although *Emath* offers many useful operations, including multiple overlapping windows, incremental 2-dimensional editing, mouse assisted cut and paste and selection operations, etc... *Emath* is *not* a user interface but a parameterized component designed to be integrated in any application that needs to perform formula editing. Figure 1 pictures the architecture of a graphical user interface (GUI) that uses *Emath*. The interface can be used to access a computer algebra system (CAS). A library (lib) linked to the application allows to start *Emath* as a server process. When *Emath* is running, the application is able to delegate one or more windows to the server. Once a window is under the control of *Emath*, all the X events associated to this window are managed by *Emath* and no more by the application. Editing formulas is then possible in this window. If the basic functionalities of *Emath* are enough, no extra work needs to be done by the application but writing functions that map between the representation used by *Emath* and the representation used by the application. The application and the server can exchange information

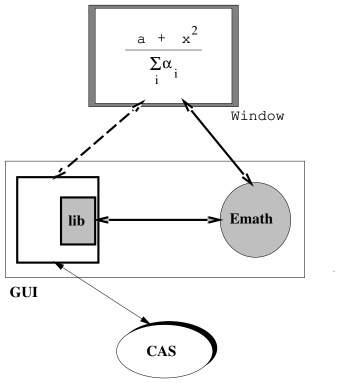

Figure 1. Using in an interface

in Lisp format. Actually, a whole Lisp interpreter is embedded in *Emath* and provides primitive operations for building expression trees and functions for displaying them on a window. Thus, changing the default behavior of *Emath* is achieved by writing and evaluating some Lisp code which makes customization rather simple. If needed, it is possible to modify the way *Emath* reacts to the X events that occur on the windows it manages. For example, the application can ask *Emath* to forward some special events instead of leaving *Emath* treating them by itself. This mechanism achieves a high level of flexibility and parameterization, and allows to implement sophisticated behaviors *outside Emath*. Some particular features (like the "semantic displacement" of THEORIST and *Milo*) which are usually built-in in the interface can be implemented without knowing how *Emath* itself is implemented.

3. Using *Emath*

Usually *Emath* interacts with both an application and a user. The issue of how the editor and the application communicate is especially important because we want to separate them as much as possible to favor the reusability of *Emath*.

Basically, an application interacts with *Emath* by sending and receiving Lisp s-expressions. These s-expressions are in fact evaluated by a Lisp interpreter embedded inside the editor. The user interacts directly with *Emath* through the mechanism of events (key pressed, mouse buttons...). These events are generated by the X server and sent to the windows managed by the editor.

3.1. *Emath* AND THE APPLICATION

The application can ask arbitrary Lisp code to be executed by *Emath*. The editor returns s-expressions to the application. From a practical point of view, a simple library is provided for the communication of these s-expressions.

The Lisp interpreter in the editor provides primitive operations that build the trees that represent formulas. For example, sending the two s-expression

```
(setq test-formula (mkt '(/ (+ a b) c)))
(display-formula test-formula current-window)
```

causes the formula $(a+b)/c$ to be built in the editor, assigned to `test-formula` and then displayed in `current-window`, where `current-window` is bound to a window managed by *Emath*.

Conversely, *Emath* sends back notification of events to the application. These events, that are also transmitted as s-expressions, can include the X events but also higher level events, internal to the editor: creation of a formula, deletion of a sub-formula, selection... A simple occurrence notation can be used to identify sub-formulas. For example, the selection of $a+b$ in $c(a+b)$ is typically expressed as `(selection-event (path 1))`.

Emath can in fact display more than one "logical" formula in one window. Several formulas can be grouped (together with text or pixmaps) by structuring operators (horizontal or vertical lists, for example). The application can then simply keep track of the occurrences of the formula thanks to the power of having a Lisp interpreter inside *Emath*. All interesting occurrences can be given a name or can be stored in an array and the application can reference this name (or index) later.

3.1.1. Formulas, trees and attributes

The formulas represented inside *Emath* are in fact abstract syntax trees. Some "concrete" details of the formulas are not explicitly given. These details are, for example, the parenthesis used to indicate the precedence of infix operators. This means that the editor really has some mathematical knowledge built in. This knowledge is contained in a data structure that holds information about each "known" operator: status (infix, postfix, prefix), precedence, formatting method... There are functions to enter and modify this information (see section 4 for further discussion).

Each node of a formula can carry attributes. Attributes are used by *Emath* to specify some display parameters (foreground or background color, font) or for managing events. Attributes can be specified, consulted and modified by the client. The client can also add its own attributes. For example, when a formula is created, the client can annotate each node with an attribute that identify the node. When a selection is performed, the attribute at its root can be consulted for the client to identify the sub-tree.

Attributes can also be used by the application to distinguish between similar operators with different meanings (overloading).

3.1.2. Managing events

The *Emath* server handles all X events associated with the windows it manages. Its behavior with respect to these events is described by an automaton that implements an extension of the translation mechanism of the X toolkit (see 3.2.1). *Emath* can also generate and manage other events: creating a node in a formula tree, selecting a sub-formula, moving in a formula... All events can be treated locally by the *Emath* server (this is the default), or forwarded to the application for a more clever treatment. For example, we can implement a "semantic displacement" where $a-b=0$ becomes $a=b$ when the b is dragged to the other side of the equation ([Avi90]). This can be done by forwarding the corresponding event to the application that can solve the equation and send back the result. For some events, the round-trip time (from *Emath* to the application back to *Emath*) may be too large. In these cases, it is possible for the application to install special Lisp code for handling the events inside *Emath*.

3.2. *Emath* AND THE USER

3.2.1. ENTERING FORMULAS

How formulas can be entered in an editor is more complicated than it seems. Several models have been proposed and implemented: the classical one-dimensional style, template-based models and overlay models (see the discussion in [Soi91]). *Emath* does not directly implement a particular model, it just gives the basic mechanisms to support all of these models and to define new ones. In fact, *Emath* treats uniformly all inputs, whether they are associated with the input of a formula or with moving around a (partially) constructed formula, or selecting with mouse buttons.

The management of inputs in *Emath* is thus entirely based on events. Such events are associated with keyboard or mouse actions (moving, pressing or releasing a button). The idea is to associate some actions, described as Lisp functions, to the occurrences of an event. The choice of which action will be triggered by an event is done with respect to a current state. Events are thus managed by a finite-state automaton that changes states as a result of a keystroke or mouse action. Such an automaton is described by what we call an *extended translation table*. This can be seen as an extension of the translation tables used by the X toolkit. Here is an example of what can be in a typical extended translation table:

state	event	action	next state
`selection-activated`	`<key>LeftArrow`	`move-selection-left`	
`selection-activated`	`<key>Space`	`extend-selection`	
`edition`	`<key>LeftArrow`	`move-cursor-left`	
`edition`	`<key>Delete`	`delete-current-node`	

What happens when the left arrow key is struck depends on whether we are in the `selection-activated` or the `edition` state. For these two events, the automaton does not change state.

One might argue that this is a relatively low level of abstraction that is closer to lexical analysis than parsing. For example, it is not clear how to easily check if the parenthesis are balanced in a formula. In fact, it is relatively straightforward to implement a template-based input model with this mechanism where it is only possible to construct legal formulas.

For more complicated input models, we really need a parser to check the correctness. That is why an extensible precedence parser that can work incrementally is included in *Emath* (based on the works of [JG86] and [Soi91]) and can be called in actions. For the moment, this parser is written in Lisp.

The advanced user can also entirely program its own input model by associating each event to its own parser, written as a Lisp function.

We thought of implementing a more complex mechanism where the actions are not only dependent on a state but also on a tree pattern that describes the "shape" of the current formula. In a certain way, this is more natural and can save some states (that are typically used to keep track of certain tree patterns). Unfortunately, this is not very easy to implement. We might make experiments with such a specification in the future.

3.2.2. Moving and selecting

We can perform displacements based on the syntactic structure of the formula as well as displacements based on its "spatial" bidimensional structure. These displacements can be bound to keys like CTRL-F or CTRL-E in the spirit of *Emacs*.

Moving and selecting are parameterized through the above mechanism of extended translation tables. The primitive operations for moving are syntactic displacements on the tree representing the formula (up or down one level, going to the next son of an operator...) and spatial displacements (going left, right, up or down in the two dimensional representation of the formula...).

Syntax is nice to extend a selection. Syntactic displacement may be more awkward because the syntactic structure of a formula does not directly reflect its spatial structure. So, it can be a bit hard for the user who tends to think in term of where he wants to go in the spatial structure rather than in the underlying, astract tree structure. Hence, *Emath* also supports spatial displacements inside formulas. This is similar to what is found in text editor. A clever algorithm is used to obtain a natural behavior.

For selecting, *Emath* has only one primitive operation. *Emath* supports ordered multiple selections through the use of a special attribute. Several selections can be active at once and each selection has an associated "time stamp" (contained in the value of the selection attribute). Of course, *Emath* allows the classical mouse events (single, double and triple clicks and button holding and releasing (to define a rectangular region) to be associated with selection actions. It is also possible to use the keyboard alone to perform selections. What exactly is selected is defined by some Lisp code associated with the selection action. We have a small selection library that implements two models of selection corresponding to the biggest enclosed expression and the smallest containing expression. This library also contains functions to extend the selection.

3.3. Integrating *Emath* in an interface

The editor runs as a server process. Once the server has started, the application can communicate with *Emath* by sending it some Lisp code. It must delegate one or more windows to the server for displaying and editing formulas. This delegation can occur at any time (not just when *Emath* starts) and is performed by executing the Lisp function `manage-window` that takes as its argument the X id of the window to use.

Practically, the programmer of an application that needs equation editing or displaying should follow these steps to use *Emath*:

1 create a wigdet hierarchy in its application (that can include menus, scrollbars, text, buttons...) with one or more neutral widget (for example drawing area widgets) where the formulas will be displayed and manipulated. Then delegate the windows associated with these widgets to the *Emath* server. *Emath* will take charge of the management of all X events received by these widgets (refresh, cursor displacements, mouse clicks).

2 if necessary, write some Lisp code that will customize *Emath* to its need. This includes writing the extended translation tables used to manage inputs (if *Emath* is used for entering formulas, not just for displaying) and the information for the formatting process. We provide some sensible default behaviors and a library of

Lisp functions that implements the most common behaviors (for displaying and editing) and that contains useful tools to build special ones.

3 write functions that map between the representation used by *Emath* and the representation used by the application. If *Emath* is used only as a display engine, the work is very easy and should only be done for mapping the application representation to *Emath* trees. If the application can get formulas back, the reverse process must also be implemented.

4. Displaying formulas

From a formula to its image on the screen, three main steps are necessary:

1 decide the format that is used to represent each operator (decide that a quotient is displayed as the numerator above the denominator, separated with a line, that an exponent is written on the right, above and in a smaller font...)

2 compute the images of each characters and symbols that occur in the representation of the formula and their placements in the target window. Usually, computing these images revert to choosing a font and a character size. But in the case of special symbols that have an adjustable size (parenthesis or braces) or the symbols for integral or summation, more work can be involved.

3 map these images at the right places onto the window.

The situation is complicated by the fact that the formatting process can depend on constraints related to the target window (most notably, its size). In *Emath*, the attributes can also drive some of the decisions on formatting.

In *Emath*, the formatting process can be completely under control from the application. As we mentioned in 3.1.1, the application sends what can be considered as abstract syntax trees. *Emath* uses information that can be associated to each operator to know how to format. Some *a priori* knowledge of the basic arithmetic operators is included: associativity, precedence, the equivalence between $a+(-b)$ and $a-b$ or between ab^{-1} and a/b. This was designed to keep the mapping between the representation of formulas in the application and the representation used by *Emath* as simple as possible. Nevertheless, these equivalences can be invalidated. A simple pattern-matching facility (implemented as a Lisp library) is available to make translations to define new equivalences (similar, in spirit, to the translations described in [Soi91]).

Two important issues when dealing with displaying formulas are efficiency and incrementality. Efficiency is a concern because we want to handle large expressions. Ideally, the time spent to draw (format) a formula should depend only on the size of the visible part of the expression. Incrementality is necessary because we would want to be able to modify (edit) the formula and thus redraw only what is necessary.

Like most programs that display formulas, the algorithm used in *Emath* is based on the calculation of a tree of bounding boxes ([Knu84], [Mar71]). Each box contains a sub-formula (a sub-tree), a character string or a special symbol, together with parameters that indicate its position relative to the parent box. The tree of the bounding boxes associated with a formula is computed lazily to avoid constructing the boxes that are not in the visible part of the formula (or near this zone to speed up scrolling).

Emath is a multi-view editor. It is possible to display the same formula in several windows. Different attributes can be used in each view, resulting in different renderings.

Colors, fonts, sizes, display functions... can be changed. A factorization mechanism based on an inheritance hierarchy is used to minimize the memory consumption.

A classical double-buffering mechanism is used to render the formula on the screen. It avoids flickering during scrolling and editing. We are also experimenting with bitmaps caching to reduce the computing time when local modifications are performed. This consists of keeping part of the formula as a bitmap (or pixmap) that can be just translated to redraw the formula. For example, if we want to add 1 in front of a large, complicated formula, we can just draw 1+ and translate bitmap image of the formula to the right, after the + sign.

4.1. Elision and line cutting

Two big (and sometimes controversial) issues are how to cut long expressions and how to hide parts of a formula to reduce its size while making its structure more apparent (elision). Unfortunately, it seems that interesting elisions or clever line cuttings require a good deal of mathematical knowledge, much more than a tool like *Emath* can have. The philosophy of *Emath* is thus to delegate these kinds of decisions to the application. However, we need to be more flexible, for applications that don't want to deal with these issues. The mechanism for elision is available in *Emath* through a special attribute. This attribute can be set by the application. The application can also decide to send a special symbol indicating elision instead of a sub-term. Line cutting is more problematic. It is an non-essential feature because *Emath* can scroll (both horizontally and vertically) along a formula. This can be linked to some keyboard sequence or the actions on scroll-bars managed by the application around an *Emath* window. If we really want to avoid scrolling, the simplest tolerable solution is certainly to introduce names for intermediate expressions and make references to these names in the original expression. In an editing mode, *Emath* allows the user to explicitly break its expressions into several lines.

4.2. Characters, symbols and fonts

We directly use the fonts in the X server (scalable fonts when available) and draw resizable symbols (integral and sum symbols, parenthesis...). We make experiments to dynamically generate these symbols from a outline description with Bezier curves, using optical rescalling. The results are not yet completely satisfactory.

We plan to make experiences with using anti-aliasing and grey scale fonts to improve readability. This is relatively expensive because of the cost associated with using pixel based operations in X. However, this can become a viable alternative together with bitmap caching.

5. Implementation

Emath is approximately 14000 lines of C code and 1500 lines of Lisp for the core facilities.

5.1. Klone

One of the major capabilities of *Emath* is its flexibility. *Emath* has been designed to make the modification of all its behaviors (including displaying of formulas and managing

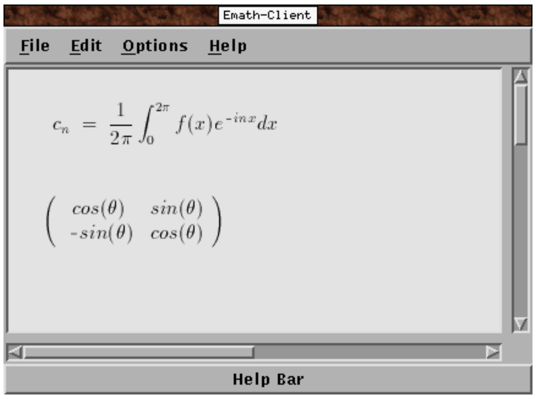

Figure 2. A simple interface built with

of X events) very easy. This flexibility is achieved by using an extension language. For this purpose, we use KLONE, a Common Lisp dialect. KLONE has been designed to be embedded in C applications. Calling a C function or accessing a C structure from the KLONE interpreter is straightforward. KLONE is among the most efficient embeddable Lisp interpreter and still a very small process (about 200K). Thus, *Emath* can be viewed as a kernel of C primitives acting on the basic structures of the editor and callable from the KLONE interpreter. This architecture has many advantages. First, it ensures extensibility for the user as well as for the developper of applications that embeds *Emath*. Second, it allows the implementors of *Emath* to experiment in a dynamic easy-to-use environment. Actually, nothing prevents to extend KLONE with other primitives that could be useful for the design of software that uses *Emath*. We experimented such an extension by providing KLONE with a complete interface to *Motif*. Then it is possible to build a prototype of a complete graphical interface in a very short time. Such a prototype is under development. Figure 2 shows a screen dump of this interface.

5.2. Reactivity

Constructing a tree, computing all the bounding boxes and the display itself can take time, in spite of the efforts to make everything incremental. Our decision to favor customization forbids some optimizations. Under a classical model: treat an event and then redisplay. A good reactivity is lost: during the time to treat an event, other events are blocked, refresh is a blocking operation. Then, we must detach refresh from treating events. In *Emath*, events are treated with the maximum priority. Displaying occurs only when there is nothing else to do and is interrupted when an event occurs.

This behavior is possible using the X toolkit “workprocs”. But in this case, the refresh function must be cut in pieces whose duration don’t exceed the needed level of reactivity.

To preserve enough reactivity and to keep the code as simple as possible, we use *threads*. The details of the multithreading mechanism we use in *Emath* is out of the scope of this paper. Roughly speaking, multithreading allows to run several functions concurrently within the same process. In a single processor machine, the concurrency is achieved by a scheduler which performs time-sharing on the parallel subprocesses. We used the package [Kep93] to implement a special non-premptive scheduler optimized for our needs.

Two threads are associated to each window managed by *Emath*. The one that manages

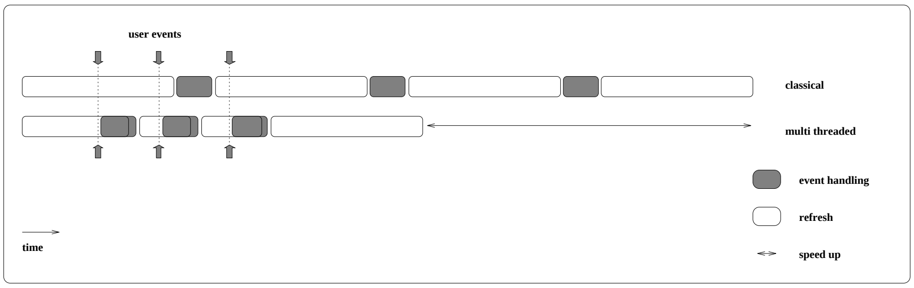

Figure 3. Refreshing with threads

events and the one that takes care of refresh. Giving a higher priority to the threads that control the events, we ensure that these events are treated as soon as they are detected. This may lead to interruptions during the refresh process, but as shown in figure 3 the complete display should be faster than the one used in more classical model. This solution is even better with muti-views where we can decide that the thread associated to the window that owns the focus is the one with the highest priority. So that the window where formulas are entered is the one that will be refreshed first.

5.3. Memory usage

For each node in a formula, there exist a great number of attributes. Some attributes are predefined and the user can add new ones. In a subexpression, the attributes are inherited from the outer expression. Then, if all the nodes of an expression tree share the same value for a given attribute, only the root of the tree needs to be attributed. Then, looking for an attribute value implies a partial tree traversal. An inheritance mechanism is used to share (factor) as much as possible.

The structures related to displaying (trees of bounding boxes or bitmap caches) are kept (constructed) only for the part of the formula corresponding to its visible part. So that the memory used for that does not vary with the size of the formula. The current implementation uses approximately 30 bytes for each node in a formula. The startup size of an *Emath* process is about 1M.

6. Conclusion

Emath provides an original solution to the problem of formula editing. More than a YAGUI ("Yet Another Graphical User Interface") for computer algebra, *Emath* is a powerful and highly extensible equation editor kernel that can be embedded in any application that needs displaying/editing of mathematical formulas. These include high level graphical user interface for computer algebra systems as well as document formatting systems or electronic books. It is hoped that the basic design of *Emath* provides a flexible framework with which to experiment with various applications, particularly out of the scope of computer algebra. Tools like *Emath* provide a base for additional work on the more fundamental issues involved in creating the ideal user environment for scientific computation.

References

ABMW88 Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger. Caminoreal: an interactive mathematical notebook. In *EP'88 International Conference on Electronic Publi shing, Document Manipulation and Typography*, Nice, France, April 1988. Cambridge University Press.

Avi90 Ron Avitzur. Suggestions for a friendlier user interface. In A. Miola, editor, *LNCS 429, DISCO'90*, pages 282–283. Springer-Verlag, 1990.

JG86 Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 196–206. ACM, 1986.

Kep93 David Keppel. Tools and technniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

Knu84 Donald E. Knuth. *The TEX book.* Addison Wesley, 1984.

KS94 N. Kajler and N. Soiffer. Some Human Interaction Issues In Computer Algebra. *ACM SIGSAM Bulletin*, (28):18–28, March 1994. Also in ACM SIGCHI Bulletin 28(1).

Mar71 William A. Martin. Computer input/output of mathematical expressions. In *Second Symposium on symbolic and algebraic manipulation*, pages 78–89. ACM, 1971.

Soi91 Neil Soiffer. *The Design of a User Interface for Computer Algebra Systems.* PhD thesis, University of California, Berkeley, 1991. Report UCB/CSD 91/626.

SS86 Carolyn Smith and N. Soiffer. Mathscribe: A user interface for computer algebra systems. In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, pages 7–13, July 1986.

vEHvV89 S. van Egmond, F. C. Heeman, and J. C. van Vliet. INFORM: an interactive syntax-based formulae editor. *Journal of Systems and Software*, 9:169–182, 1989.

YW87 Douglas A. Young and Paul S. Wang. GI/S: A graphical user interface for symbolic computation systems. *Journal of Symbolic Computation*, 4:365–380, 1987.